Transparencias Procesadores de Lenguaje

Elena Díaz Fernández

29 de septiembre de 2004

Tema 1

Introducción al proceso de traducción

Bibliografía:

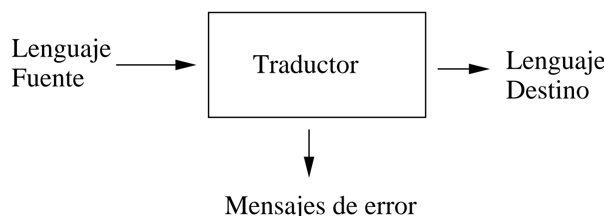
- Aho, A.V., Sethi, R., Ullman, J.D. (1990), *Compiladores: principios, técnicas y herramientas*, Tema 1, páginas: 1- 25, 743-747.
- Louden, K.C. (1997), *Compiler Construction: Principles and Practice*, Tema 1, páginas: 1- 27.

Contenido:

1. ¿Qué es un traductor?. Tipos de traductores.
2. Fases del proceso de traducción.
3. Especificación de los lenguajes de programación.
4. Programas del sistema relacionados con los traductores.
5. Principales estructuras de datos en un traductor.
6. Mil lenguajes y mil máquinas, ¿un millón de traductores?:
   a) *Front-end* y *Back-end*. Traducción cruzada.
   b) *Bootstraping*.
7. Herramientas de ayuda a la traducción.
8. Aplicaciones de los sistemas de traducción.
9. Relación con otras áreas.

1.1. ¿Qué es un traductor?. Tipos de traductores.

Traductor: programa que traduce un lenguaje (el lenguaje fuente) a otro lenguaje (el lenguaje objeto). Informa de posibles errores.



En el contexto de la compilación de programas, el lenguaje fuente es normalmente un lenguaje de alto nivel, y el lenguaje destino u objeto es el lenguaje máquina.
Gran diversidad de lenguajes fuentes y lenguajes objetos implica gran diversidad de traductores. Algunos ejemplos:

lenguaje fuente	lenguaje objeto	
C	código máquina	// compilador de C
C	código ensamblador	
C	Pascal	
C++	C	
código ensamblador	código máquina	// ensamblador
Latex	Html	
Html	Xml	
Inglés	Castellano	
Imagen pgm	Imagen bmp	

Breve introducción histórica:

- **1940-50:**
    - primeros ordenadores digitales: programados en lenguaje máquina (códigos numéricos que representaban las operaciones máquina). Mucho tiempo de programación y tedioso.

      C7 06 0000 0002 (mover el número 2 a la dirección 0000).

  - código ensamblador: las instrucciones y posiciones de memoria se dan simbólicamente MOV X, 2 (X tiene la dirección 0000). Reducción del tiempo de escritura y de la corrección de programas. Todavía difícil de leer y escribir, dependiente de la máquina (código debe ser reescrito si se cambia de máquina).
  - necesidad de lenguajes que permitan escribir los programas de forma concisa, similar a una notación matemática, y que se pudieran traducir a código ejecutable para una máquina dada. (X=2).
  - primeras impresiones: sería extremadamente difícil (casi imposible), generarían código muy ineficiente (en aquella época escasos recursos de computación). Existia un escepticismo generalizadp.

- **1950-60:**
  - El término *compilador* fue introducido por primera vez en 1950 por G. M. Hooper. La traducción de programas era entonces vista como la *reunion* de una secuencia de subprogramas de una librería.
  - aparecen los primeros trabajos sobre compiladores relacionados con la traducción de formulas aritméticas a código máquina.
  - John Backus lideró un grupo de trabajo en IBM para la realización de un traductor a código máquina de fórmulas matemáticas. Resultado con gran éxito: especificación de un lenguaje de alto nivel (FORTRAN, FORmule TRANslation) y la realización de un traductor para una máquina (IBM 704). Se tardo el trabajo de 18 personas en un año. Fué un compilador hecho “ad-hoc”, no existia una teoria formal, sino que se iban resolviendo las construcciones una a una, para cada situación particular.
  - Chomsky comienza estudios sobre la estructura del lenguaje natural. Clasificación de los lenguajes (jerarquía de las gramáticas) y estudios sobre los algoritmos de re-

conocimiento (susceptibles de ser automatizables y de forma eficiente).

- en Europa surgió una corriente más universitaria, que pretendía que la definición de un lenguaje fuese independiente de la máquina y donde los algoritmos se pudieran expresar de forma más simple. Influida por los trabajos de Chomsky. Surgió el grupo de Bauer en Alemania para definir un lenguaje de usos múltiples. Surgió el ALGOritmitc Language (ALGOL). La sintaxis fue completamente especificada por primera vez con la notación BNF (Backus-Naur-Form).
- ALGOL 60, primer lenguaje de propósito múltiple que gozó de amplio uso. Se introducen las características de los lenguajes actuales: definición de la sintaxis en notación BNF, formato libre, declaración explícita de tipos, estructuras iterativas, recursividad, paso de parámetros por valor y nombre y estructura de bloques, con el consiguiente ámbito de las variables.

- **1960-70:**
  - Diseño del lenguaje de alto nivel LISP. En un principio, el código LISP se traducía manualmente a código máquina (también para el IBM 704). Se escribió en LISP un programa capaz de interpretar programas LISP, que se tradujo manualmente a código de máquina, construyendo de este modo un intérprete ejecutable de LISP.
  - Knuth definió las gramáticas LR y determinó cómo construir la tabla de análisis sintáctico ascendente LR. Se definieron las gramáticas LL para el análisis descendente.
  - primeros lenguajes, como FORTRAN y ALGOL 60, los tipos de datos eran muy sencillos y su verificación también. Con el ALGOL 68 se empieza a aplicar la coerción de tipos y la equivalencia estructural y por nombre.
  - entornos de ejecución estáticos: no recursividad y no almacenamiento dinámico de memoria.

- optimización: eliminación de subexpresiones comunes, identificación de código muerto, sustitución de operaciones aritméticas, propagación de constantes y cálculo previo de constantes, variables de inducción, propagación de copias o código inalcanzable, elección de determinadas instrucciones, registros,etc

- **1970-80:**
  - aparecen los generadores automáticos de analizadores léxicos o sintácticos, como *Lex* o *Yacc (Yet another compiler-compiler).*
  - se multiplican los lenguajes de programación (torre de Babel) debido a los avances teóricos en la formalización de los lenguajes y a la sistematización de las herramientas para la generación de reconocedores.
  - PASCAL diseñado por N. Wirth, del Instituto Politécnico Zurich. Pensado para la enseñanza. Goza de ser buen progenitor de posteriores lenguaje (Modula-2, ADA). Representación intermedia: código P. Separar el proceso de traducción en dos fases: el *front-end* encargada de analizar el programa fuente (operaciones dependientes sólo del lenguaje fuente) y el *back-end* encargada de generar el código para la máquina objeto (operaciones dependientes sólo del lenguaje fuente). Front-end codificado a mano usando un analizador descendente predictivo recursivo.
  - la compilación pasa a ser una materia con un cuerpo de conocimiento propio. Aparecen los primeros libros sobre compilación.
  - entornos de ejecución basados en pila y montículo: recursividad, gestión dinámica de memoria, métodos de paso de parámetros, acceso a variables no locales (ámbitos de referencia anidados).

- **1980-90:**

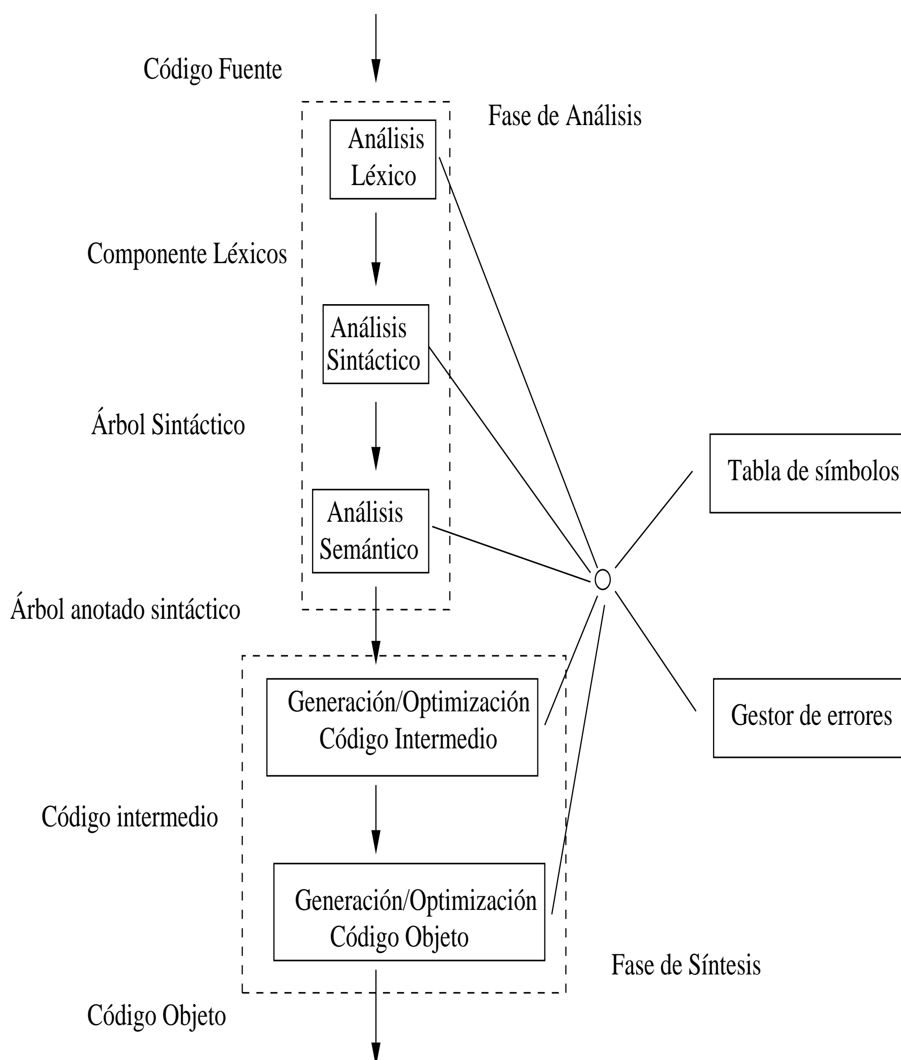
- gramáticas de atributos para la especificación semántica del lenguaje. No es un punto de vista tan formal como para el análisis léxico y sintáctico (excepto la semántica denotacional para lenguajes funcionales). Son comprobaciones "ad-hoc" implementadas a mano.
- estudios sobre la optimización de código basado en análisis de flujo de control y de datos. Tarea muy compleja, investigación hasta la fecha. Análisis de control de flujo y de datos, predicción de ramificaciones, ... Optimizaciones dependientes de la máquina: jerarquía de memoria, encauzamientos, número de registros, etc.
- entornos completamente dinámicos de ejecución: compilador gestiona la asignación y desasignación de memoria en tiempo de ejecución, recolección de basura y referencias desactivadas (entornos de ejecución basados en montículos *heap*).

- **1990-:**
  - se consolida la programación orientada a objetos.
  - compiladores en un entorno de desarrollo junto con editores, enlazadores, cargadores, depuradores, gestores de proyecto, analizadores de rendimiento.
  - herramientas para la generación automática de código.
  - no hay cambios en el análisis léxico y sintáctico, conceptos y métodos persistentes, sólidamente establecidos.
  - se vuelve a poner de moda la compilación a un *código intermedio* (código-byte) que pueda ejecutarse en cualquier máquina, bien mediante un intérprete de máquina virtual: Java (JVM, Java Virtual Machine), Perl, Python, bien mediante compilación *just-in-time*: Java, Oberon. El código es ejecutado por un intérprete normalmente incluido en un navegador de Internet.
  - los lenguajes interpretados de muy alto nivel (*lenguajes de script*), (re)aparecen con fuerza (Python, Perl,

Tcl/Tk, shells, awk), permiten construir y prototipar rápidamente programas complejos, interfaces de usuario, implementar filtros para tratamiento de información textual, etc.

1.2. Fases del proceso de traducción.

Conveniente desde un punto de vista conceptual ver un compilador como una serie de fases con diferentes fines (en la realidad esta separación es menos estricta, los módulos se solapan entre sí).



- *el analizador léxico:* lee la secuencia de caracteres de izquierda a derecha del fichero fuente y los agrupa en unidades con significado propio (los componentes léxicos): palabras clave, identificadores, operadores, constantes numéricas, signos de puntuación como separadores de sentencias, llaves, paréntesis , etc. Además de eliminación de comentarios, procesamien-

to de macros o inclusión de ficheros. La estructura léxica la modelizaremos mediante expresiones regulares.

- *el análisis sintáctico:* determina si la secuencia de componentes léxicos sigue la sintaxis del lenguaje y obtiene la estructura jerárquica del programa en forma de árbol, donde los nodos son las construcciones de alto nivel del lenguaje. Se determinan las relaciones estructurales entre los componentes léxicos. La estructura sintáctica la definiremos mediantes gramáticas independientes del contexto (GIC).

- *el análisis semántico:* realiza comprobaciones necesarias sobre el árbol sintáctico para el correcto significado del programa: verificación e inferencia de tipos en asignaciones y expresiones, declaración antes de uso (variables, funciones, tipos), correcto uso de operadores, ámbito de variables, correcta llamada a funciones ... Nos limitaremos al análisis semántico estático (en tiempo de compilación). Necesidad de hacer uso de la Tabla de Símbolos, como estructura de datos para almacenar información sobre los identificadores que van surgiendo a lo largo del programa. El análisis semántico se suele implementar a mano y se suele asociar a construcciones del lenguaje o un subárbol. Por eje: para la produccion $E \to E + E$, deberiamos incluir código para hacer la comprobacion de tipos, validez de los operadores para esos operandos, etc. La semántica la modelizaremos mediante gramáticas de atributos (GIC+atributos que representan propiedades de las construccciones del lenguaje: por ejemplo: tipo, valor, etc).

- *la fase de generación y optimización de código intermedio:* linearización del árbol sintáctico donde ya no aparecen construcciones de alto nivel (similar a un lenguaje tipo ensamblador). Es un código ya no estructurado más fácil de traducir directamente a código ensamblador o máquina: código de tres direcciones (cada instrucción tiene un operador, y la dirección de dos operandos y un lugar donde guardar el resultado). Optimizaciones que sólo dependen del lenguaje fuente (y no de la máquina): eliminación de subexpresiones

comunes, identificación de código muerto, sustitución de operaciones aritméticas, cálculo previo de constantes, variables de inducción, propagación de copias o código inalcanzable. Suele ser una fase lenta y compleja. Existe la "optimizacion de mirilla ("peephole optimization"), en la que sólo se consideran una cuantas instrucciones a cada vez, y se realizan optimizadiones sencillas como eliminar las multiplicaciones por 1, remplazar una secuencia de instrucciones por otra sóla más eficiente (incrementar el valor en una posición de memoria en 1).

- *la generación y optimización de código objeto:* toma como entrada la representación intermedia y genera el código ensamblador o máquina. Optimizaciones dependientes de la máquina, es necesario conocer: conjunto de instrucciones, la representación de los datos (número de bytes), modos de direccionamiento, número y propósito de registros, jerarquía de memoria (espec. la caché), encauzamientos, etc. Suelen implementarse a mano, y son complejos porque la generación de un buen código objeto (máquina) requiere la consideración de muchos casos particulares.

  También se está investigando en la creacción de crear generadores de código automáticos (sorcerer). La idea es automaticamente hacer corresponder una representación intermedia (IR) a plantillas de intrucciones objeto. Permite facilmente *retarget* el compilador a una nueva máquina objeto, se cambiaría el nuevo conjunto de plantillas. Ejemplo (GCC, GNU C compoler). Posee plantillas para mas de 10 arquitecturas más habituales de ordenadores.

- *Tabla de Símbolos:* estructura tipo diccionario con operaciones de insertar, borrar y buscar que almacena información sobre los símbolos que van apareciendo a lo largo del programa: los identificadores (variables y funciones), etiquetas, tipos definidos por el usuario (arrays, records,...). Almacena el tipo de dato, la aridad, método de paso de parámetros, tipo de retorno y de argumentos de una función, el ámbito de

referencia de identificadores y la dirección de memoria. Interacciona tanto con el analizador léxico, sintáctico y semántico que introducen información conforme se procesa la entrada. La fase de generación de código y optimización también la usan.

- *el gestor de errores:* detecta e informa de errores de cada fase que se produzcan durante el análisis. Debe generar mensajes significativos y reanudar la traducción. Encontramos errores:

  - Detectables en tiempo de compilación: errores léxicos (ortográficos), sintácticos (construcciones incorrectas) y semánticos (p. ej. uso de variables no declaradas, errores de tipo, etc).
  - Detectables en tiempo de ejecución: direccionamiento de vectores fuera de rango, divisiones por cero, etc.
  - De especificación/diseño: compilan correctamente pero no realizan lo que el programador desea.

  Se trataran sólo errores estáticos (en tiempo de compilación). Respecto a los errores en tiempo de ejecución, es necesario que el traductor genere código para la comprobación de errores específicos, su adecuado tratamiento y los mecanismos de tratamiento de excepciones para que el programa se continúe ejecutando.

La mayoría de los compiladores son *dirigidos por la sintáxis*, es decir, el proceso de traducción es dirigido por el análizador sintáctico. El análisis sintáctico genera la estructura del programa fuente a través de *tokens*. El análisis semántico proporcionan el significado del programa basándose de la estructura del árbol de análisis sintáctico
Las fases de análisis léxico y análisis sintáctico se pueden automatizar fácilmente, las verdaderas dificultades en la construcción de compiladores son el análisis semántico, la generación y la optimización de código.
Número de pasadas: número de veces que hay que analizar el código fuente. Es función del grado de optimización.

Típicamente: una pasada para realizar el análisis léxico y sintáctico, otra pasada para el análisis semántico y optimización dependiente del lenguaje fuente y una tercera pasada para generación de código y optimizaciones dependientes de la máquina.

C, Pascal: admiten una pasada; Modula-2: dos pasadas; Compiladores optimizadores: 5 o más pasadas.

Programas que se compilan muchas veces y ejecutan pocas veces (entornos de enseñanza), poca optimización (una pasada).

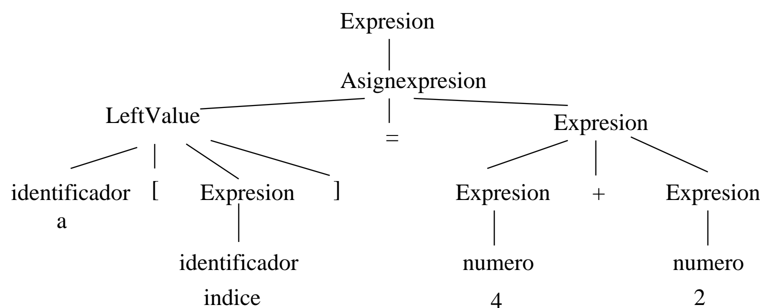
Ejemplo: `a[indice] = 4 + 2 ;`

Componentes léxicos:

a	identificador
[	corcherte apertura
indice	identificador
]	corchete cierre
=	operador asignación
4	número
+	operador suma
2	número
;	punto y coma

Expresion →	Asignexpresion | Expresion + Expresion | identificador | **numero**
Asignexpresion →	LeftValue = Expresion ;
LeftValue →	**identificador** [Expresion]

Arboles de análisis sintáctico y de significado semántico:



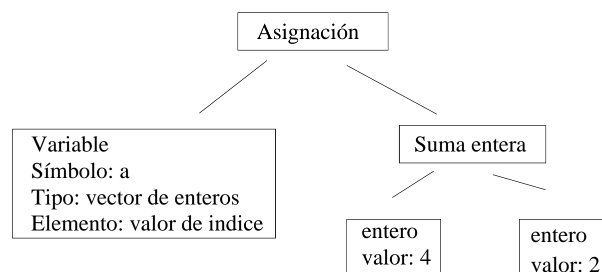


Tabla de símbolos:

...				
a	variable	vector de enteros	tamaño: 10	direcc. memoria
índice	variable	entero	direcc. memoria	
...				

Código intermedio:

No optimizado	Optimizado
t=4+2	a[indice]=6
a[indice] = t	

Código objeto (ensamblador):

```
MOV  R0,   indice // valor de indice a R0
MUL  R0,   2              // multiplica por dos indice (2 bytes para en
MOV  R1,   &a             // dirección de a a R1
ADD  R1,   R0             // añade contenido de R0 a R1
MOV  *R1,  6              // constante 6 a la direccción almacenada en
```

Se ha supuesto que un entero ocupa dos bytes de memoria, &a es la "dirección de ", *R1 implica indirect register adressing.

1.3. Especificación de los lenguajes de programación

Supón que tuvieras que describir un lenguaje de programación. Una manera de hacerlo sería comenzando por describir cuáles son las unidades elementales tales como identificadores, palabras reservadas, operadores, etc. que se encuentran en la entrada. Después podrías describir, cómo se pueden combinar esas unidades en estructuras mayores tales como expresiones, asignaciones, bucles y demás. Finalmente, especificarías una serie de normas que deben cumplirse para que el programa, además de estar "bien escrito", tenga significado (por ejemplo, reglas de que las variables deben declararse antes de usarse, o que los tipos en asignaciones deben ser equivalentes).
Podemos considerar que la definición de un lenguaje consta de tres partes:

- *Descripción léxica*: ¿Cuáles son las "palabras" del lenguaje?. Las palabras de un lenguaje de programación reciben el nombre de *símbolos o componentes léxicos (tokens)*.
- *Descripción sintáctica*: ¿Cuáles son las "frases" del lenguaje?. En un lenguaje de programación, las "frases" son los programas bien construidos sintácticamente.
- *Descripción semántica*: ¿Qué "significa" cada frase del lenguaje?. El significado del programa se describe en términos de lo que éste hace al ser ejecutado.

¿De qué métodos disponemos para especificar el lenguaje?

Estructura	**Especificación**	**Reconocimiento**	**Coste**
Léxica	Expres. regulares	Autómatas Finitos Determ (AFD)	$O(\|x\|)$
Sintáctica	Gram. Indep. Contexto (GIC)	Autómatas a pila	$O(\|x\|^3)$
Semántica	Semántica formal, leng. natural	Reglas "ad-hoc"	

Como puedes observar existe un paralelismo entre especificar un lenguaje de programación y diseñar un traductor para él.

1.3.1. Un ejemplo de la especificación informal de un lenguaje

Supondremos un lenguaje extremadamente sencillo, que llamaremos Micro, que servirá para ilustrar el diseño e implementación de los diferentes modulos que forman un traductor a lo largo del curso:

- El único tipo de dato que existe es entero.
- Los identificadores se declaran implícitamente, empiezan con una letra y están compuestos de letras, dígitos y subrayados.
- Los literales son cadenas de dígitos.
- Los comentarios empiezan por $--$ y terminan al final de la línea actual.
- Las sentencias son:
  Asignaciones:
  $ID := Expresion$, donde Expression se construye a partir de identificadores, literales y operadores de suma (+), resta ($-$), y paréntesis.

  Entrada/Salida:
  read (Lista de Identificadores);
  write (Lista de Expresiones);

- **begin**, **end**, **read**, y **write** son palabras reservadas.
- Cada sentencia se termina con un punto y coma (;). El cuerpo de un programa debe estar delimitado por **begin** y **end**.

1.4. Programas del sistema relacionados con los traductores.

- *Interpretes:* ejecuta las instrucciones del programa fuente inmediatamente en vez de esperar a que esté traducido por completo el código fuente a código máquina. Necesitan menos memoria, pero más lentos (factor de 10) que los compiladores. LISP, BASIC. Históricamente, se pusieron de moda en los primeros años frente porque los recursos de memoria eran escasos. Permiten añadir código dinámicamente durante la ejecución (APL). Desventajas: menor velocidad, peor información sobre los errores al no tener una visión global del programa.

- *Ensambladores:* traductor de lenguaje ensamblador a lenguaje máquina. Es sencillo (correspondencia casi biunívoca de instrucciones).

- *Enlazadores:* en un único fichero une diferentes ficheros objeto compilados separadamente e incluye las librerías necesarias.

- *Cargadores:* normalmente el fichero generado por el enlazador no está completamente preparado para ser ejecutado, las direcciones de memoria son relativas (código relocalizable). El cargador asigna una dirección fija de comienzo y resuelve las diferencias relativas.

- *Preprocesadores:* llamado por el compilador antes del proceso de traducción. Elimina comentarios, incluye ficheros de cabecera, sustituye macros, funciones en línea.

- *Editores:* editores de texto orientados al formato o estructura del lenguaje que llegan a incluir algunas de las operaciones propias del compiladores como informar de errores (Visual C++, Matlab,..)

- *Depuradores:* determina errores de ejecución en un programa compilado. Cuando se ejecuta en modo *debugging* se

mantiene información sobre todo el código, variables en cada momento, procedimientos en la pila de activación, número de línea, ...

- *Analizadores de rendimiento (profilers) :* recoge estadísticas sobre comportamiento del programa durante la ejecución (número de veces que se llama a un procedimiento, porcentaje de tiempo en cada uno, ...). Se pueden reutilizar por el propio compilador para optimizar.
- *Gestores de proyecto:* fusión de ficheros de diferentes programadores y versiones, históricos de cambios producidos ...

Los compiladores se pueden distinguir de acuerdo a la clase cd código objeto que generan:

- *Pure Machine Code*: se genera código para un conjunto particular de instrucciones, sin tener en cuenta la existencia de un S.O. o librerias. El código generado se puede ejecutar directamente sobre el hardware sin ninguna dependencia de software.
- *Augmente Machine Code*: se genera código para una arquitectura dada "aumentada" con rutinas del S.O. y rutinas de soporte al lenguaje (I/O rutinas, rutinas para la reserva de memoria, funciones matemáticas,...) se tienen que cargar junto con el programa. La combinación del conjunto de instrucciones del lenguaje máquina, el S.O y las rutinas del lenguaje se pueden considerar como la definición de una *máquina virtual*, que existe como la combinación de hardware y software.
- Virtual Machine Code: es el caso extremo de una máquina virtual, el código generado se compone enteramente de instrucciones virtuales (aumenta transportabilidad de compiladores, sólo tendriamos que reescribir de la máquina virtual usada por el compilador). Ejemplo sobresaliente: Pascal. El compilador generaba código para una máquina virtual a pila, *P-code*. La escritura de un traductor de P-code a código máquina se puede hacer fácilmente. De ahí, la popularidad de Pascal.

1.5. Diseño de Traductores y Lenguajes de Programación

El diseño de compiladores y de lenguajes de programación se influencian el uno sobre el otro. El estado de las técnicas en el diseño de compiladores influye en el diseño de lenguajes: un lenguaje que no puede ser compilado no se va a usar!. Los lenguajes que son fáciles de compilar presentan muchas ventajas:

- Son fáciles de aprender, de leer y de entender.
- Existirá una variedad de compiladores sobre máquinas dadas. Aumenta su popularidad (p. eje. Pascal).
- Se generará mejor código.
- El compilador será más pequeño, barato, rápido, más realiable y por tanto más usado.

¿Qué influye para que un lenguaje de programación sea compilable? Depende de qué es lo que se puede determinar antes de la ejecución (estático) y que se puede realizar después de la ejecución (dinámico). Preguntas como: Se puede determinar el ámbito y tipo de un identifcicador antes de la ejecución? Se puede cambiar texto del programa o añadir durante la ejecución? (Por ejemplo, LISP permite crear texto durante la ejecución).

1.6. Estructuras de datos en un traductor.

Las principales estructuras de datos que se necesitan en el proceso de traducción y que sirven para comunicarse entre las fases:

- *los componentes léxicos:* estructura tipo registro con dos campos: el tipo de componente léxico que se representa por un tipo enumerado y el lexema como una cadena de caracteres. Una variable global: el símbolo de preanálisis.
- *el árbol sintáctico:* se genera de forma dinámica como una estructura estándar basada en punteros conforme se avanza

en el proceso de análisis sintáctico. Cada nodo es un registro con información obtenida por el analizador léxico (lexema), sintáctico (tipo) y semántico (por ej. valor de una expresión, tipo de una expresión). Esta información depende del tipo de construcción del lenguaje que ese nodo represente.

- *la Tabla de Símbolos:* contiene información sobre los identificadores, funciones, variables, ámbito de referencia de identificadores, constantes numéricas y literales, tipos de datos, o incluso la dirección de memoria. Importante que las operaciones de inserción, búsqueda y eliminación sean de coste casi constante: tabla *Hash*.

- *el código intermedio:* se implementa como una lista de registros, donde cada registro tiene cuatro campos (operador, la dirección de los operandos y del resultado). Es eficiente para mover código para el proceso de optimización posterior. Se puede usar también un fichero texto.

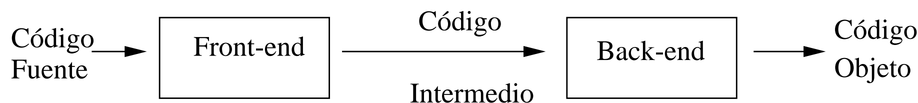
1.7. Mil lenguajes y mil máquinas, ¿un millón de traductores.

¿ Cómo podemos facilitar la transportabilidad de compiladores: varios lenguajes fuente y varias máquinas objeto? **Solución:** División de las operaciones a realizar en dos tipos:

- *front-end o etapa inicial:* operaciones que dependen sólo del lenguaje fuente. Incluye: análisis léxico, sintáctico y semántico, la creación de la Tabla de Símbolos, generación de código intermedio y algunas optimizaciones. Además, del manejo de errores de cada fase.

- *back-end o etapa final:* operaciones que dependen sólo de la máquina objeto. Incluye: generación de código objeto y optimizaciones dependientes de la máquina. Depende de los modos de direccionamiento, conjunto de instrucciones de la máquina, número de registros, arquitectura de la máquina, sistema operativo, ...

La representación intermedia actúa como medio de comunicación entre ellas. Ejemplos de código intermedio son: Código de 3-direcciones (tenemos un operando, las dos direcciones de los dos argumentos y la dirección donde se almacena el resultado) y Código a pila (para una máquina abstracta con un conjunto de instrucciones y de registros y modos de direccionamiento, p-code. (N. Wirth, distribuyó a todas las universidades, el *front-end* de Pascal escrito en p-code para una máquina virtual basada en pila, y esto fue lo que potenció su gran expansión inicial en entornos educacionales, ya que sólo era necesario construir el *back-end* para la máquina con la que se trabajaba).



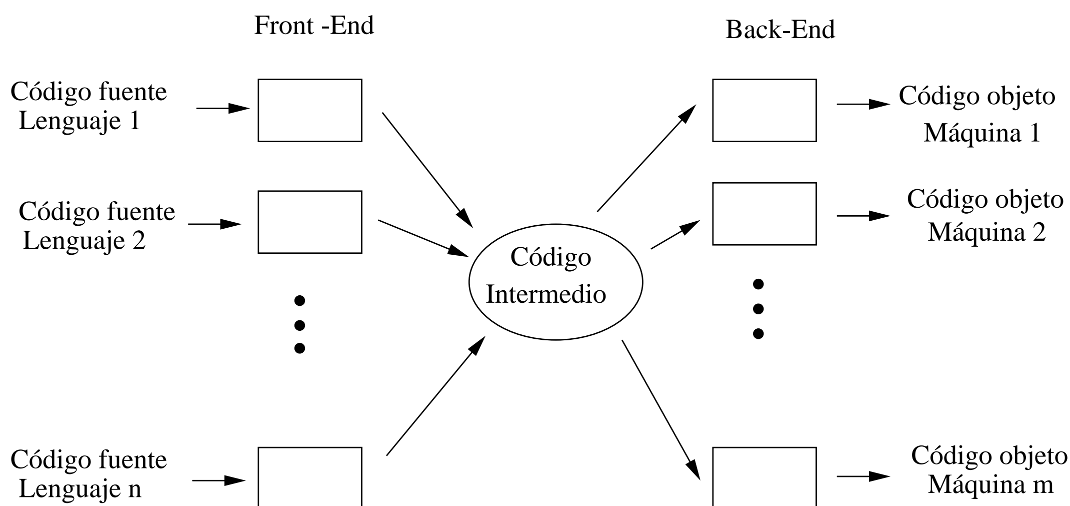
Si se cambia de lenguaje fuente, entonces se reescribe el *front-end*. Si se cambia la máquina objeto, entonces se reescribe el *back-end*. Si aparece una nueva arquitectura, basta con desarrollar un traductor del lenguaje intermedio a esa nueva máquina.

Por ejemplo, la expresión `(A+B)*(C+D)`:

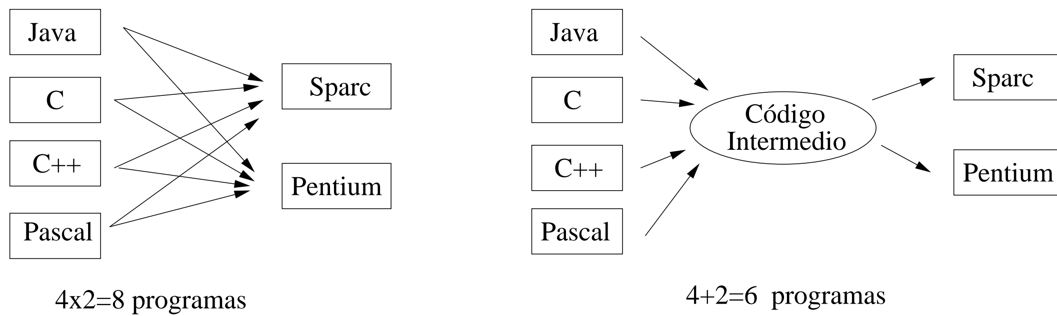
Código de 3-direcciones	
(+, A, B, t1)	suma A y B, coloca el resultado en t1
(+, C, D, t2)	suma C y D, coloca el resultado en t2
(*, t1, t2, t3)	multiplica t1 y t2, y coloca el resultado en t3

Código a pila	
LOAD A	carga el contenido de A en el acumulador
ADD B	añade el contenido de B al del acumulador
STO t1	almacena el contenido del acumulador en la var. temporal t1
LOAD C	carga el contenido de C en el acumulador
ADD D	añade el contenido de D al del acumulador
STO t2	almacena el contenido del acumulador en la var. temporal t2
LOAD t1	carga el contenido de t1 en el acumulador
MUL t2	multiplica el contenido de t2 por el del acumulador
STO t3	almacena el contenido del acumulador en la var. temporal t3

Supongamos n lenguajes fuente y m máquinas objeto.

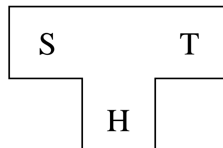


Necesitamos hacer $n + m$ programas en vez de $n * m$.



1.7.1. Compiladores cruzados

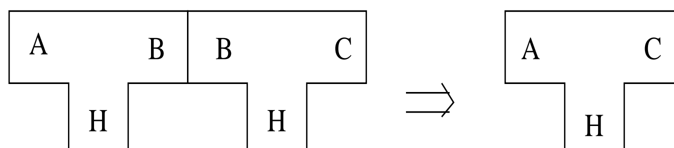
- Definimos los *diagramas en T*. Un compilador escrito en lenguaje H que traduce el lenguaje fuente S al lenguaje objeto T se representa:



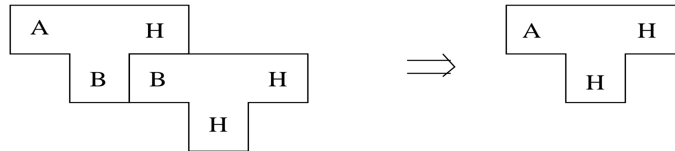
H: lenguaje en el que está escrito el compilador (*host language*), es el lenguaje de la máquina sobre la que se ejecuta.
S: el lenguaje fuente (*source language*).
T: lenguaje objeto (*target language*).

Un *compilador es cruzado* si H es distinto de T (genera código para una máquina diferente en la que se está ejecutando).

- *Escenario 1*: tenemos dos compiladores que corren sobre la misma máquina; el primero traduce el lenguaje A al lenguaje B; el segundo traduce el lenguaje B al lenguaje C. Si los combinamos, obtenemos un compilador que traduce el lenguaje A al lenguaje C.

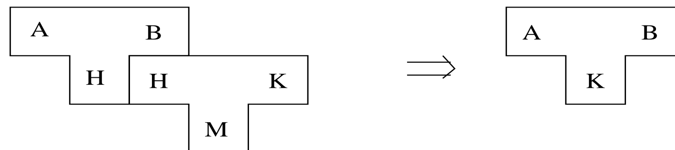


- *Escenario 2*: tenemos un compilador del lenguaje B que corre sobre la máquina H (está escrito en el lenguaje máquina H y genera código para H). Queremos un compilador del lenguaje A que corra bajo H.



  Aumentamos el número de lenguajes fuente para esa máquina.

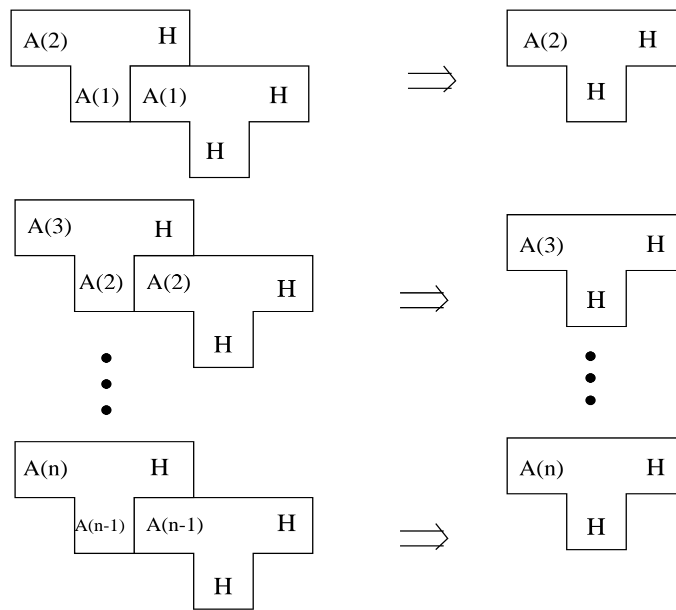
- *Escenario 3*: tenemos un traductor del lenguaje A al lenguaje B que está escrito en H, buscamos un traductor de A a B que esté escrito en K.



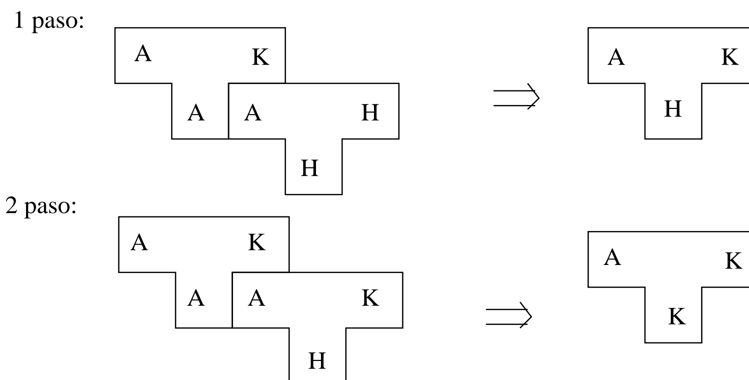
  Aumentamos el número de máquinas donde se puede ejecutar el traductor.

1.7.2. Bootstraping

Se escribe el compilador en el mismo lenguaje fuente, lenguaje A, que se quiere compilar. ¿Cómo? Parece un problema circular. (*Bootstraping*=arranque).
El proceso es como sigue: se define un subconjunto muy reducido de A que llamaremos A(1), y para A(1) se construye un compilador a mano en el lenguaje H, que es el que soporta la máquina. Dado el compilador de A(1) se define una ampliación del lenguaje A(1), que llamaremos A(2), y se construye un compilador escrito en el lenguaje A(1). El proceso se repite n veces hasta obtener un un compilador del lenguaje A (A=A(n)) que se ha escrito en el propio lenguaje que se queria compilar, lenguaje A.



¿Cómo portar un compilador escrito en su propio lenguaje en una máquina H a otra máquina K? Sólo es necesario cambiar el



back-end. Se hace en dos pasos:

- primero *retargeting*: modificar el compilador para producir código objeto para la nueva máquina.
- segundo *rehosting*: modificar el compilador para correr en la nueva máquina.

1.8. Herramientas de ayuda a la traducción.

Herramientas especializadas para implantar los módulos de un traductor. Programas de ayuda en el proceso de escritura de compiladores: *sistemas generadores de traductores.* Se les llama *compiler writing tools, compiler generators, compiler-compilers.*

- *Generadores de analizadores léxicos:* a partir de una especificación basada en expresiones regulares. Generan un AFD. Veremos: *Lex/Flex.*
- *Generadores de analizadores sintácticos:* a partir de una entrada que es la gramática independiente del contexto que representa la estructura sintáctica del lenguaje. Veremos: *Yacc/Bison.*
- *Generadores de código:* con rutinas para la generación del árbol de análisis sintáctico y para su recorrido. En cada nodo se especifican las acciones para su traducción a códigol correspondiente código objeto.
- *Dispositivos para el análisis de flujo de datos y de control:* analizan la forma en que se transmiten los valores de un bloque a otro del programa, útiles en la optimización, predicción de ramificaciones, eliminación de código muerto, eliminación de redundancias, etc.

Incluyen la generación de la Tabla de Símbolos e incluso paquetes para la reparación de errores.

1.9. Aplicaciones de los sistemas de traducción.

Se basan en los mismos conceptos teóricos y técnicas que en el diseño de traductores:

- *Procesadores y formateadores de texto:* entrada cadena de caracteres que incluye el texto a componer y órdenes para indicar capítulos, secciones, párrafos, enumeraciones, figuras, fórmulas, tablas, ... (Latex, Html, ...).
- *Editores de texto:* emacs (búsquedas y sustituciones de patrones de texto).
- *Intérpretes de consulta:* traduce un predicado con operadores relacionales y booleanos a órdenes para buscar en una base de datos los registros que cumplan esos predicados.
- *Reconocedores y conversores de formatos de ficheros:* por ejemplo en UNIX ntroff, troff, eqn (para definir ecuaciones), pic (para dibujar) o tbl (para formatear tablas)
- *Traducción de programas:* programas escritos en lenguajes obsoletos a lenguajes más modernos, o simplemente de un lenguaje a otro (Pascal a C, PHP a ASP,...).
- *Tratamiento de ficheros con información estructurada*: auge de XML.
- *Intérpretes de comandos de sistemas operativos:*
- *Sistemas de impresión y de dibujo de figuras:*
- *Procesamiento del lenguaje natural:*
- *Reconocimiento de patrones en Visión por Computador:*
- *Creacción de circuitos VLSI*, un silicon compiler, debe forzar las reglas de diseño que dictan la feasibilidad de un circuito dado.

1.10. Relación con otras áreas.

Estructura de los Computadores I, II: la representación de los tipos básicos, el lenguaje ensamblador son claves para entender la gestión de la memoria en entornos de ejecución estáticos y dinámicos o la generación y optimización de código, que se trata de aprovechar las características de la máquina: la elección de registros, los modos de direccionamiento, la memoria caché y el uso de instrucciones especiales.

Algoritmos y Estructuras de Datos: para implementar las estructuras básicas: la tabla de símbolos, el árbol de análisis sintáctico y la generación de código intermedio. Los conocimientos sobre la implementación de tablas *Hash*, la construcción y métodos de recorrido de árboles, así como las listas enlazadas son imprescindibles.

Fundamentos de la Programación y Metodología de la Programaci
es necesario facilidad en alguno de los lenguajes de programación, un lenguaje imperativo u orientado a objetos (normalmente C/C++). Los algoritmos de análisis sintáctico y de generación de código suponen un cierto grado de complejidad debido a su naturaleza recursiva.

Teoría de Autómatas y Lenguajes Formales: Establece los fundamentos teóricos en los que se basan las técnicas de análisis de léxico y sintáctico: la construcción de expresiones regulares a partir de una determinada especificación y de los AFD para su reconocimiento, especificación de autómatas mediante diagramas y tablas de estado, la conversión de autómatas no-deterministas a deterministas, la minimización del número de estados, las gramáticas independientes del contexto y los autómatas a pila para su reconocimiento, la notación Backus-Naur-Form y su versión extendida, los diagramas sintácticos para la definición de una gramática, las gramáticas limpias y bien formadas, los tipos de derivaciones de una sentencia, el problema de la ambigüedad en las

gramáticas...

Lenguajes de Programación: La complejidad del lenguaje determina la complejidad del traductor y de este modo las herramientas a utilizar para su construcción. El conjunto de palabras reservadas, su estructura sintáctica, la verificación de tipos, el uso de recursividad, el anidamiento en las funciones, los métodos de paso de parámetros, los métodos de direccionamiento, las estrategias para la asignación de memoria desde los entornos completamente estáticos sin memoria dinámica ni recursividad hasta los completamente dinámicos que requieren complejos algoritmos para gestionar la memoria y donde el compilador debe generar código para ello, las estructuras de datos permitidas, las formas de llamadas a las funciones y los valores de retorno. Por otro lado, la representación de los tipos, la forma en que la Tabla de Símbolos es mantenida, las reglas usadas para la verificación e inferencia de tipos dependen de las expresiones y constructores de tipos del lenguaje. De igual modo, las limitaciones de las técnicas actuales de traducción restringe en gran medida la sintaxis de las construcciones de los lenguajes de programación, de modo que esta interacción es en doble sentido.

Inteligencia Artificial e Ingeniería del Conocimiento: Si bien en el procesamiento del lenguaje natural se utilizan métodos de análisis sintáctico no-deterministas, los conceptos y técnicas relacionados con la especificación y reconocimiento léxico de un lenguaje, de su sintaxis y el análisis semántico asociado a las construcciones del lenguaje son un común denominador en el procesamiento de cualquier lenguaje.

Ingeniería del Software I: El diseño y construcción de un traductor suele ser en general un proyecto de nivel intermedio o alto en cuanto a complejidad, por tanto un buen conocimiento sobre los aspectos relacionados con el diseño lógico y físico de sistemas y su impacto en la fiabilidad, eficiencia, utilidad y mantenimiento son importantes.

Visión por Computador: Por un lado, en la descripción de los contornos se hace uso de cadenas que explotan las relaciones estructurales que existen entre los elementos de un contorno y por otro lado los conceptos y técnicas de análisis sintáctico se pueden aplicar en el problema del reconocimiento de patrones, incluso se utilizan reglas semánticas para especificar las relaciones de conectividad entre primitivas. Además, no se puede olvidar, el enfoque estructural en la clasificación de texturas, basado en la analogía entre las relaciones espaciales de las primitivas de la textura y la estructura de un lenguaje formal.

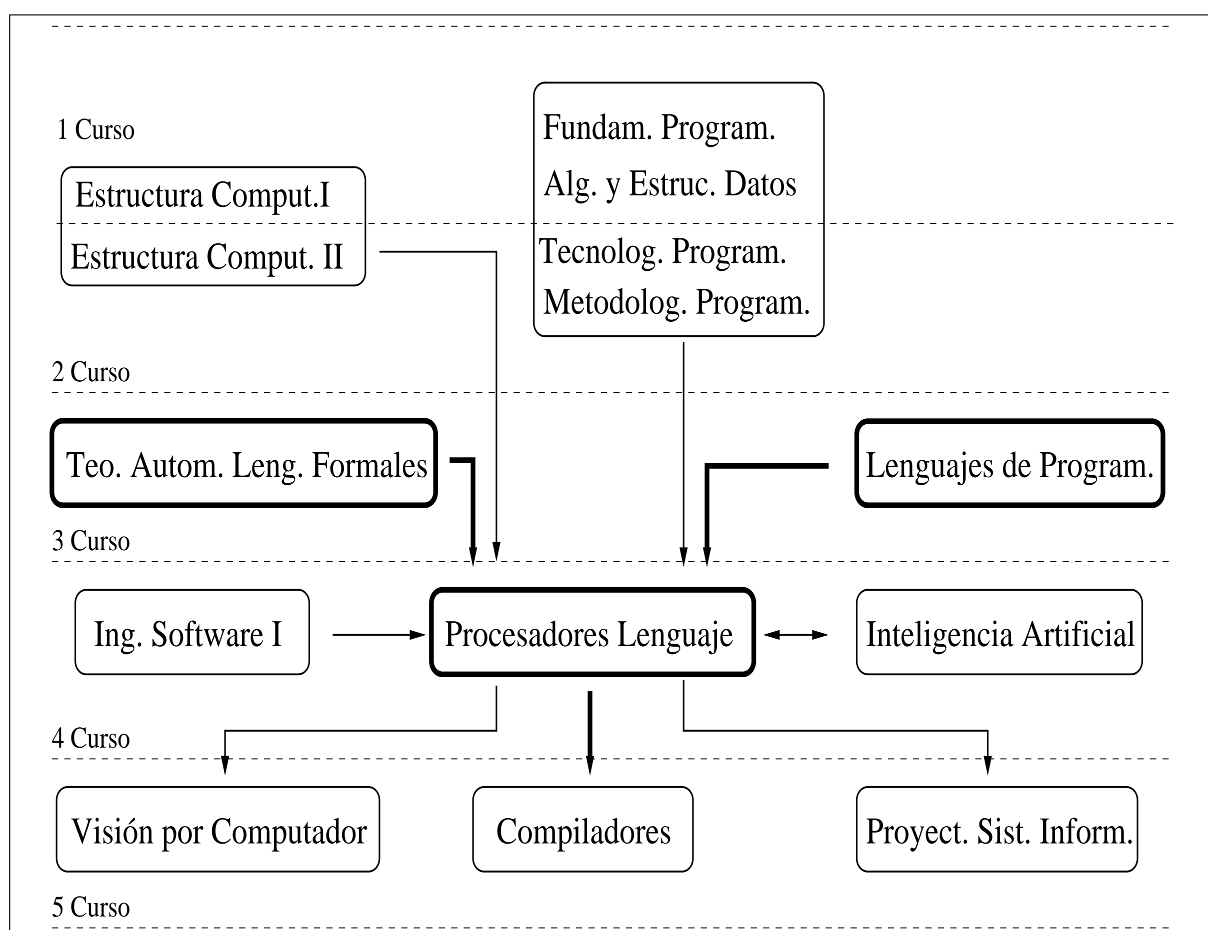


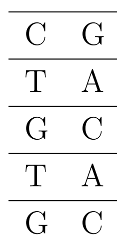
Figura 1.1: Relaciones de la asignatura de Procesadores de Lenguaje

1.11. Ejercicios:

1. (0.2 ptos) Discutir cómo el proceso de análisis que realizan pequeños sistemas se parece al proceso de traducción. Por ejemplo:
   - una pequeña calculadora que opera con enteros y reales.
   - un *script* que busca los ficheros con una determinada extensión.
   - un servidor de correo electrónico que analiza los campos de la cabecera del correo.

2. (0.2 ptos) Considerese la siguiente definición de un artículo científico: un artículo se compone de una cabecera, un texto y una lista no vacía de referencias bibliográficas. La cabecera se compone de un título, nombre de uno o más autores y un resumen opcional. Un texto se compone de párrafos. Usando la notación **t:** título, **a:** autor, **r**: resumen, **p:** párrafo, **b:** referencia bibliográfica, obtener una gramática que reconozca una colección de artículos.

3. (0.2 ptos) Las cadenas de ADN de los cromosomas están formadas por cuatro tipos de bases: A *adenina*, C *citosina*, G *guanina*, y T *tinina*. En todas las especies se cumple que la cantidad de A es igual a la cantidad de T y que la cantidad de G es siempre igual a la cantidad de C. El ADN es una larga doble hilera de bases, de manera que, entre una hilera y otra las bases se aparean siempre siguiendo las reglas: A siempre con T, G siempre con C. En el caso que nos piden, además, se debe cumplir que las parejas de bases están alternadas, es decir, las parejas AT y TA sólo pueden ir entre las parejas del tipo CG y GC y viceversa. Por ejemplo:

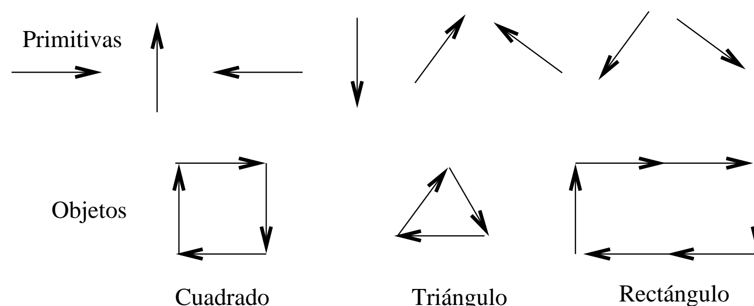


C	G
T	A
G	C
T	A
G	C

4. (0.2 ptos) Suponer que se tiene un traductor de Pascal a C escrito en C y un compilador de C sobre una determinada máquina. Se pide usar los diagramas en T para describir los pasos necesarios para crear un compilador de Pascal que funcione sobre dicha máquina.

5. (0.2 ptos) Suponer que se tiene un compilador de C sobre una determinada máquina. ¿Qué podríamos hacer para tener un compilador de C++ sobre esa máquina?

6. (0.2 ptos) Supongamos que de una imagen, una vez que ha sido procesada, extraemos del contorno de cada objeto una serie de segmentos de igual longitud orientados en determinadas direcciones (las primitivas). Diseñar una gramática que permita reconocer si el objeto se trata de un cuadrado, de un triángulo o de un rectángulo. ¿ Qué acciones semánticas se pueden añadir para obtener el perímetro del objeto?



7. (0.3 ptos) Supongamos la siguiente gramática para generar un pequeño programa con declaraciones e instrucciones :

Programa →	DeclVar ListaInstr
DeclVar →	**var id :** Tipo **;** DeclVar | ϵ
Tipo →	**integer** | **real**
ListaInstr →	**id** = Expresion **;** ListaInstr | ϵ
Expresion →	**id** | **num** | Expresion + Expresion | Expresion * Expresion

```
var tiempo : integer;
var desplazamiento : real;
var velocidad : real;
tiempo=10;
desplazamiento=20*2+5.5;
velocidad = tiempo * desplazamiento;
```

- Determinar los componentes léxicos.
- Obtener el árbol sintáctico para ese programa.
- Comentar las acciones semánticas necesarias para verificar que el código es correcto: comprobaciones de tipos, inferencia de tipos en expresiones, declaración antes de uso, correcto uso de operadores,... Indicar qué información se podría guardar en cada nodo del árbol.
- Comentar el contenido de la tabla de símbolos: identificadores y sus atributos (nombre, tipo, ámbito de referencia).
- Generar el código intermedio de 3-direcciones y las posibles optimizaciones que se podrían hacer: cálculo previo de constantes, propagación de copias,...