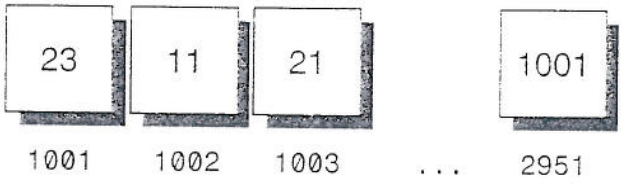
que sigue teniendo una longitud de 2, aunque es mucho más cara que las anteriores en tiempo de ejecución, porque se requiere efectuar dos accesos adicionales a la memoria: ir a la primera dirección especificada y extraer de allí la segunda dirección, que finalmente llevará al valor requerido por la operación. La cantidad de micropasos necesarios para completar la instrucción aumentó considerablemente, pero se obtiene la capacidad de "seguirle la pista" a direcciones variables mediante un indicador que no cambia de posición, como se verá más adelante.

De esta forma, el CARGA indirecto

```
CARGA-ind <direcc>
```

tiene longitud 2, y se le asignará el código 22.

Por ejemplo, supóngase que la memoria tiene los siguientes contenidos en las celdas indicadas:



Entonces la instrucción

```
CARGA-ind 2951
```

irá a la celda 2951, que es interpretada como la dirección1, y extraerá de allí su contenido: la dirección2 en donde se encuentra el valor a ser cargado en el acumulador. En el ejemplo, la dirección 2951 lleva a la 1001, en donde está el valor a ser cargado en el acumulador: 23.

La potencia de esto se vuelve evidente si se incrementa en uno el valor de la dirección2 contenida en la celda 2951, porque entonces una nueva ejecución de la misma instrucción de carga indirecta llevará ahora al valor almacenado en la siguiente celda (1002), y así sucesivamente.

Por último, el **direccionamiento indizado** toma la dirección que está a la derecha de la instrucción y la suma con el contenido de un registro especial de la UCP llamado registro índice. Al igual que la modalidad anterior, esto permite la variación de direcciones para realizar un recorrido sobre los elementos de un conjunto de datos:

Op ((direcc + Registro índice) valor)

Existen muchos otros modos de direccionamiento diferentes, aunque la máquina que "construimos" en el capítulo 2 sólo emplea los tres primeros. Para más referencias, consúltese por ejemplo [MACA95].

5.2 Primer nivel de lenguajes: Ensamblador

Acabamos de plantear la idea de usar el diccionario de equivalencias para comunicarnos con la computadora, y esto nos obliga a resolver el problema de traducir el programa fuente (escrito con los nombres simbólicos —**mnemónicos**— allí definidos) al lenguaje de máquina (el único que la UCP admite).

Se podría pensar en contratar a un traductor (que llamaremos T_1) para que lea cada programa fuente y lo traduzca a lenguaje de máquina. Con un traductor así ya no sería necesario trabajar en lenguaje de máquina, pues programaríamos en un lenguaje de más alto nivel. Para que esto siempre funcionara habría que encontrar la forma de integrar el traductor a la máquina misma y hacer que la computadora traduzca por sí sola los programas fuente a programas objeto.

¿Cómo se logra este paso fundamental?

Si el traductor se convierte en un programa y se deja residente en la memoria de la computadora, el proceso de comunicación con ella tendría dos pasos: primero, convertir el programa fuente a programa objeto y, segundo, llevar a la memoria y ejecutar ese programa objeto, que ya quedó traducido a lenguaje de máquina.

Ahora la comunicación con la computadora requiere dos pasos

El traductor ensamblador

Analizaremos los pasos necesarios para construir un traductor de esta clase. Se propone una manera sencilla de atacar problemas complejos, consistente en describir en español, a grandes rasgos, una solución general. Esa solución será un primer acercamiento al problema, y seguramente harán falta varios acercamientos progresivos para entender y resolver una situación compleja, aunque por lo menos ya se ha descrito una forma general de lograrlo. Más aún, esto constituye una metodología de diseño que debe explorarse en cursos posteriores. Ahora, volvamos a nuestro asunto.

Se desea hacer un programa T_1 que reciba como entrada un programa fuente escrito en mnemónicos (a partir del diccionario de equivalencias entre los códigos de instrucciones y sus nombres mnemónicos), y produzca como salida el mismo programa, convertido ya a lenguaje de máquina listo para ser ejecutado.

Diseño inicial de un primer traductor

Un primer acercamiento podría ser el siguiente:

```
! Programa "T1", primera versión.
! Traductor de lenguaje de mnemónicos a lenguaje de máquina.
! (El símbolo "!" se usa para escribir comentarios.)
Para cada renglón del programa fuente se ejecuta lo siguiente:
   Buscar la palabra mnemónica en el diccionario de equivalencias.
   Si está, entonces traducirla a lenguaje de máquina
   (es decir, reemplazarla por la columna de la derecha de esa entrada en el
   diccionario).
   En caso contrario mandar un mensaje de error que diga, por ejemplo,
   "mnemónico desconocido".
Fin del programa T1.
```

En este nivel de detalle, el programa T_1 funciona. Pruebe el lector aplicarlo al ejemplo de la página 80 y verá cómo pasa del programa fuente

```
CARGA   20
SUMA    21
GUARDA  22
ALTO
```

al programa objeto equivalente

```
20 20 30 21 02 22 70
```

suponiendo, claro, que el traductor T_1 tiene acceso al diccionario de equivalencias.

Si por error el segundo renglón dijera, por ejemplo,

```
SOMA 21
```

entonces el traductor se quejaría, indicando que el mnemónico "SOMA" no está definido en el diccionario, y por ende es intraducible y constituye un error en el programa fuente.

El lenguaje ensamblador

Siguiendo por este camino, aparece la posibilidad de no tener ya que preocuparse por escoger celdas particulares de memoria, sino dejar esta responsabilidad al propio traductor. Esto es, eximir al programador de la tarea de escoger celdas de memoria y asignársela al traductor T_1. Es en este momento cuando debe introducirse el concepto de **variable**. Una variable será un nombre simbólico asociado con una celda cualquiera de la memoria de la computadora: sólo que el traductor hará esta asociación de manera automática.

Así, en lugar de decidir si se cargará el acumulador con la casilla 21 (o, si fuera el caso, con la 2951 o la 19689, etc.), se escribirá

```
CARGA ALFA
```

donde ALFA es el nombre simbólico de una celda de memoria (y ya no importa cuál será ésta). Mientras el traductor T_1 reconozca que ALFA corresponde a una celda en particular (escogida por el traductor mismo) no habrá problemas. Nosotros, como programadores, diremos ALFA y el traductor le dirá a la computadora 21 (o, si fuera el caso, 2951 o 19689, etcétera).

Independencia de las direcciones de la memoria: el concepto de variable simbólica

Éste es un paso de fundamental importancia; lograrlo implica estar, efectivamente, "por encima" de la memoria, al no tener ya que preocuparnos por direcciones absolutas. Es decir, el programador trabajará en un ambiente simbólico, no absoluto. ¿Cuáles son las ventajas de "flotar" por encima de la memoria†? En primer lugar, comenzar el largo camino de liberarse de la computadora, y dirigirse a ella en un lenguaje más simbólico que antes. Además, los programas son mucho más flexibles pues, por ejemplo, si la celda 21 está ocupada por código de otro usuario, el traductor T_1 podrá asignar ALFA a cualquier otra celda libre. Todo esto permite la creación de programas más legibles para un ser humano y, por tanto, más útiles.

El programa, entonces, dirá:

```
CARGA   ALFA
SUMA    BETA
GUARDA  GAMA
ALTO
```

que ya es algo más parecido a lo que realmente se desea hacer:

```
GAMA = ALFA + BETA
```

en donde, por ejemplo, ALFA vale 5 y BETA vale 7.

(†) Es obvio que los programadores de sistemas siempre tienen presente este verso del poeta español Antonio Machado (1875-1939):

[...]
yo amo los mundos sutiles,
ingrávidos y gentiles
como pompas de jabón.

La tabla de símbolos

Veamos qué cambios habrá que hacer a T_1 para que por sí solo pueda escoger y asignar celdas de memoria a las variables simbólicas. En primer término, deberá guardar en una tabla las direcciones absolutas que asignó a las variables simbólicas, para poderlas luego reconocer cuando sean requeridas. Esta tabla, de uso interno del traductor, se conoce como **tabla de símbolos**.

Por ejemplo, si se desea elaborar un programa para calcular C = A + B y luego D = E − C, se escribiría algo como:

```
CARGA    A
SUMA     B
GUARDA   C
CARGA    E
RESTA    C
GUARDA   D
ALTO
```

Además, el traductor debe reconocer las direcciones de todas las variables simbólicas ya que, por ejemplo, C es requerida dos veces a lo largo del programa.

Si se supone que T_1 decidió asignar celdas de memoria a partir de la dirección 70 (aunque pudo haber usado cualquier otro número), la tabla de símbolos para este programa será:

Variable simbólica	Dirección asignada
A	70
B	71
C	72
E	73
D	74

Con lo que la traducción realizada debe ser:

	Programa fuente		Programa objeto obtenido
1	CARGA	A	20 70
2	SUMA	B	30 71
3	GUARDA	C	02 72
4	CARGA	E	20 73
5	RESTA	C	33 72
6	GUARDA	D	02 74
7	ALTO		70

Obsérvese que, como la variable C aparece dos veces en el programa fuente, su dirección absoluta correspondiente (72) también aparece dos veces. Sin embargo, no hay que confundir las dos apariciones del número 70. La primera vez que aparece es porque representa la dirección que el traductor asignó a la variable A, mientras que la segunda representa la codificación (de acuerdo con nuestro diccionario de equivalencias) de la instrucción ALTO.

Cuando el traductor T_1 está leyendo el renglón fuente número 3 se encuentra por primera vez con la variable C, por lo que la introduce en la tabla de símbolos y la reemplaza en el código objeto por la dirección absoluta 72 recién asignada. Cuando la encuentra de nuevo, en el renglón 5, no la introduce en la tabla, pues ya está allí; ahora simplemente la reemplaza en el código objeto por su dirección, 72.

Falta por resolver algunos pequeños problemas técnicos, porque por lo pronto con ligereza hemos supuesto que el traductor puede por sí mismo darse cuenta de la existencia de las variables empleadas por el programador, pero en ningún momento le fueron definidas. Tampoco se dijo nunca a partir de cuál dirección se desea comenzar a asignar celdas de memoria; es decir, no se ha sustentado aún la intención de comenzar a "virtualizar" nuestra forma de comunicación con la computadora†.

Comunicación con el traductor: las pseudoinstrucciones

Para subsanar la primera omisión, inicialmente el programador debe avisar al traductor T_1 cuáles variables decidió emplear, aunque —como se dijo— sí le delegará la responsabilidad de asignarles los números de celda específicos. Para ello es necesario inventar el concepto de **pseudoinstrucción**: una instrucción que el programador escribe para el traductor, no para la máquina.

Para definir una variable se propone la pseudoinstrucción `DATO`, a la que debe preceder el nombre simbólico que el programador escogió para ella. Además, para propósitos de generalidad, conviene indicar la cantidad de celdas de memoria que ocupará (normalmente será una, pero más adelante se verá un caso con más).

Por ejemplo, la pseudoinstrucción

```
ALFA: DATO 1
```

especifica que la variable `ALFA` ocupa una celda de memoria, pero se sigue suponiendo que el traductor conoce la dirección a partir de la cual debe asignar variables, lo cual tampoco es cierto.

Para cubrir la segunda omisión se propone la pseudoinstrucción `ORIGEN`, seguida de la dirección física inicial que el programador desea que el traductor emplee.

Con todo esto en mente, el programa anterior debe entonces escribirse así:

```
        ORIGEN      70
A:      DATO        1
B:      DATO        1
C:      DATO        1
D:      DATO        1
E:      DATO        1
        PROGRAMA
        CARGA       A
        SUMA        B
        GUARDA      C
        CARGA       E
        RESTA       C
        GUARDA      D
        ALTO
        FIN
```

para que el código objeto generado por el traductor (ahora sí, en forma verdaderamente automática y sin suposiciones ligeras) sea

```
70: 00 00 00 00 00
75: 20 70
```

(†) Recuérdese que ésta es la función última del software de base: virtualizar el modo de comunicación entre la computadora y nosotros, para acercarnos y tratar de aligerar la convivencia. Pero eso no es fácil, aunque intentarlo bien vale la pena.

```
77: 30 71
79: 02 72
81: 20 73
83: 33 72
85: 02 74
87: 70
```

tal como se había dicho. Para propósitos visuales se incluyen las direcciones, aunque debe quedar claro que éstas no forman parte del código del programa objeto. Las primeras cinco celdas (a partir de la dirección 70) contendrán datos aún no especificados, y se presuponen valores iniciales de cero. La primera instrucción ejecutable está en la dirección 75, que aparece subrayada para mayor claridad.

Obsérvese que se emplearon dos nuevas pseudoinstrucciones, `PROGRAMA` y `FIN`, simplemente para indicar dónde comienzan y terminan las verdaderas instrucciones del programa fuente.

Manejo adicional de direcciones simbólicas: etiquetas

Para continuar con el tema del lenguaje simbólico, debemos ocuparnos del manejo de las **etiquetas**: referencias simbólicas a direcciones del programa, empleadas para dirigir los saltos condicionales y modificar el flujo de ejecución dependiendo de los valores que tomen ciertas variables. Como se vio en el capítulo 2, la máquina puede comparar dos números y determinar cuál es mayor; luego, con esta información resulta posible decidir cuál parte del programa debe ejecutarse a continuación, "brincándose" partes del código.

En los programas escritos en lenguaje de máquina, las direcciones a las que se puede saltar deben ser especificadas en forma absoluta, es decir, mediante un número que indica la dirección física de la celda deseada. Por el contrario, en los programas escritos en el nuevo lenguaje simbólico que estamos desarrollando, esas direcciones pueden especificarse empleando etiquetas que apuntan a direcciones físicas, pero que en sí mismas son nombres simbólicos y no números absolutos.

Un primer programa escrito en ensamblador

Por ejemplo, cuando se toma el programa del capítulo 2 para encontrar el máximo de tres números leídos (pág. 91) y se expresa ahora con pseudoinstrucciones, instrucciones mnemónicas, variables y etiquetas, se tendrá algo como:

```
! Búsqueda del máximo
        ORIGEN   3000
ALFA:   DATO     1
BETA:   DATO     1
GAMA:   DATO     1
MAX:    DATO     1
        PROGRAMA
        IN               ! Leer el primer número
        GUARDA   ALFA    ! Almacenarlo
        IN               ! Leer el segundo
        GUARDA   BETA    ! Almacenarlo
        IN               ! Leer el tercero
        GUARDA   GAMA    ! Almacenarlo
        CARGA    ALFA    ! Suponer que el primero
        GUARDA   MAX     ! es el máximo
        COMP     BETA    ! Comparar 1o. vs. 2o.
        BR>      ETIQ1
        CARGA    BETA    ! Reemplazar cuando el 2o. fue mayor
        GUARDA   MAX
```

(5) Si luego de la etiqueta aparece una instrucción
mnemónica, añadir su longitud a la dirección actual.
Si existe una pseudoinstrucción ORIGEN seguida de un número o un nombre simbólico, considerar ese número como la dirección actual e inicial, o bien buscar el nombre en la tabla de símbolos y considerar su valor como la dirección inicial.
Si existe la pseudoinstrucción FIN, termina el primer paso
!–Segundo paso–
Regresar al primer renglón después de la pseudoinstrucción PROGRAMA.
Para cada renglón del programa fuente ejecutar lo siguiente:
Ignorar la posible etiqueta que contenga.
Buscar la instrucción mnemónica en el diccionario de equivalencias.
Si está, escribir su código en el programa objeto;
en otro caso, marcar error: mnemónico desconocido.
Si a continuación el renglón contiene una variable simbólica, buscar su dirección en la tabla de símbolos.
Si está, escribir esa dirección en el programa objeto;
en otro caso, marcar error: variable (o etiqueta) desconocida.
Si existe la pseudoinstrucción FIN, termina el proceso.
Fin del programa T_1.

Como aún nos faltan técnicas de estructuración de programas debemos conformarnos con esta rebuscada receta, aunque en realidad no es tan complicada.

Con este elaborado diseño terminan nuestros esfuerzos iniciales por elevar el nivel de la comunicación con la computadora. Ojalá para el lector sea evidente que es mucho más conveniente escribir programas en ensamblador que en lenguaje de máquina, aunque también lo será el hecho de que las cosas parecen comenzar a complicarse un tanto. Como mínima compensación, puede consultar las referencias [BECL88], [DONJ72], [MACA95] y [ULLJ76] para información adicional sobre el diseño de los traductores ensambladores.

5.3 MACROPROCESADORES

En la ya imparable carrera por reducir la distancia que nos separa de la computadora (que además es lo que le da sentido a los estudios sobre computación), se podría también pensar en dar al ensamblador la capacidad de repetir, por medio de una orden, grupos complejos de instrucciones que deban aparecer en múltiples ocasiones. Esto es, compactar de alguna forma renglones repetitivos en uno solo que funja como su abreviatura, y pedir al traductor que lo expanda a la hora de la traducción.

Si los renglones

```
CARGA A
SUMA B
GUARDA C
```

aparecen con frecuencia en un programa en ensamblador, se podrían agrupar en uno solo que se llamara, por ejemplo, ADICIÓN. Cada vez que el traductor observara la abreviatura ADICIÓN como parte del programa fuente, la expandiría para producir los tres renglones anteriores.

Las macrodefiniciones ocupan un lugar central dentro de las ciencias computacionales

Este nuevo esquema recibe el nombre de **macroprocesamiento**, y es de importancia capital dentro de las ciencias de la computación porque permite —en su expresión más general— la sustitución textual de símbolos de un tipo con símbolos de otro. Esto, que parece tan simple e inocente forma parte de la idea central de las matemáticas y la computación, como se verá en la sección 6.3 .

Un **macroprocesador**[÷] trabaja con definiciones globales de renglones o símbolos, llamadas **macrodefiniciones** (o **macros**), que serán expandidas cuando se las llame para producir nuevos renglones de texto. Una **macrollamada**, por tanto, será la invocación de una macrodefinición por su nombre para lograr una sustitución textual.

Para convertir los tres renglones anteriores en una macrodefinición deberán encerrarse entre las palabras especiales `MACRO` y `FIN_MACRO`, con lo cual se constituyen en su "cuerpo". Además, la macro debe tener un nombre, y esto se logra mediante una etiqueta, a la manera usual. Así, una macrodefinición consta de los siguientes elementos:

```
<nombre>: MACRO
          renglón 1 del cuerpo
                .
                .
                .
          renglón n del cuerpo
         FIN_MACRO
```

En el caso anterior, la macrodefinición será entonces:

```
ADICIÓN: MACRO
           CARGA   A
           SUMA    B
           GUARDA  C
         FIN_MACRO
```

Donde se requiera llamarla sólo habrá que escribir su nombre (posiblemente precedido de un símbolo especial de identificación) para que entonces, cuando el macroprocesador lo detecte, produzca los tres renglones especificados como cuerpo de la macro.

Escogemos el símbolo % como indicador de una macrollamada. Entonces, cuando dentro del programa fuente el macroprocesador encuentre el renglón

```
%ADICIÓN
```

realizará la macroexpansión para obtener en su lugar los renglones

```
CARGA A
SUMA B
GUARDA C
```

que constituyen el cuerpo de la macro `ADICIÓN`, y quedan formando parte de un programa intermedio (porque no es el programa fuente original, pero tampoco es un programa objeto; simplemente es un programa que ya no tiene macros).

(÷) El término hace referencia a un programa (o conjunto de programas), y no debe confundirse con un dispositivo físico como el microprocesador.

Es decir, el macroprocesador recibe un programa fuente con macrodefiniciones, macrollamadas y texto en general, y produce como resultado un programa intermedio que contiene las macroexpansiones y (ese mismo) texto en general.

Diseño del macroprocesador

Otro traductor

Visto así, se está hablando de otro traductor, al que sólo le interesan las macrodefiniciones y las macrollamadas: cuando detecta una macrollamada dentro del programa fuente, la expande y la reemplaza por su cuerpo dentro del programa intermedio; pero si le llega un texto que no sea ni definición ni llamada de macro, despreocupadamente lo copia sin alteración al programa intermedio.

Por supuesto que para efectuar una macroexpansión (es decir, la expansión de una macro), el traductor debió haber previamente almacenado las macrodefiniciones en algún lugar interno, que llamaremos **tabla de macros**.

Con estas ideas ya podemos proponer la primera versión del macroprocesador, como sigue:

```
! Programa macroprocesador, primera versión
Para cada renglón del programa fuente ejecutar lo siguiente:
   Si contiene la palabra MACRO, buscar su nombre en la tabla de macros,
   esperando que no esté;
      si esa etiqueta ya está en la tabla, marcar error: nombre de macro repetido;
      en otro caso, darla de alta en la tabla de macros y comenzar a copiar allí los
      renglones que constituyen su cuerpo, hasta llegar al que contiene la palabra
      FIN_MACRO.
   Si el renglón contiene una macrollamada, buscar ese nombre en la tabla de
   macros, esperando que sí esté; si no se encuentra, marcar error: referencia a
   una macro desconocida;
      en otro caso, realizar la macroexpansión, copiando al programa intermedio
      los renglones almacenados en la tabla como cuerpo de esa macrodefinición.
   Si el renglón contiene texto que ni es macrodefinición ni macrollamada,
   copiarlo al programa intermedio.
Fin del programa macroprocesador.
```

A modo de ejemplo se aplicará el procedimiento anterior a un texto fuente genérico, en el cual simplemente se numeraron los renglones para poderlos distinguir en forma individual tanto en el programa fuente como en el programa intermedio resultante.

Si este fuera el "programa" fuente:

```
          ----------1
          ----------2
          ----------3
          ----------4
ALFA:  MACRO
          ----------A1
          ----------A2
          ----------A3
          ----------A4
          FIN_MACRO
          ----------5
          ----------6
```

aunque en realidad la comunicación con la computadora sigue siendo bastante limitada aún. El resto del capítulo se dedicará a explorar formas de seguirla ampliando.

Para mayor información sobre las macroexpresiones se pueden consultar las referencias [KERB76] y [BECL88]. Todos los ensambladores comerciales son en realidad macroensambladores, y en sus manuales se pueden encontrar los inacabables detalles necesarios para emplearlos.

Después de todo esto, disponemos ya de un primer lenguaje simbólico (macroensamblador) para establecer comunicación con la computadora. Sabemos también que la máquina sólo "entiende" lenguaje de máquina, y por ello resulta imprescindible la existencia de un traductor, diseñado a lo largo de las páginas anteriores. Pero aún falta lograr que la propia computadora efectúe la traducción, porque por lo pronto la hemos realizado nosotros, siguiendo la receta del traductor T_1.

Para que la computadora realice la traducción será necesario algo que parece raro: traducir el traductor†, y por ello se procederá a averiguar con mayor precisión los requerimientos para que la computadora pueda traducir los programas fuente a programas objeto por sí sola.

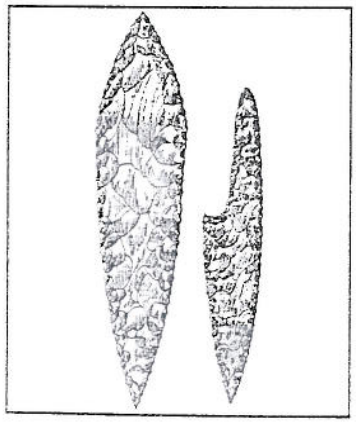
5-4 CARGADORES

Para que la computadora pueda hacer la traducción de los programas escritos en lenguaje (macro)ensamblador hacia lenguaje de máquina, es necesario que el programa traductor T_1 resida, ya traducido, en la memoria††.

Supóngase que en lenguaje (macro)ensamblador se escribió un programa, P_1, para jugar ajedrez†††. Se estudiará ahora la serie completa de pasos a realizar para lograr que la computadora traduzca y ejecute P_1. (Se iniciará desde que la máquina está apagada y se analiza lo que debe hacerse para llegar al final.)

El problema de la carga

Cuando se enciende por vez primera la computadora, la memoria está completamente vacía: es decir, el procesador está detenido, esperando alguna instrucción en memoria para leer, decodificar y ejecutar. ¿Cómo se saca a la computadora de este letargo? Está claro que aún no es posible ejecutar programas, por la sencilla razón de que no existe ninguno residente en la memoria, y todo programa debe estar cargado para poderse ejecutar. Aquí encontramos el primer problema: ¿cómo se lleva un programa a la memoria? Existen dos posibles respuestas: la primera consiste en cargar manualmente celdas de memoria con los valores numéricos que representan la codificación en lenguaje de máquina de algún programa, y la segunda en ejecutar un programa que haga esto de manera automática.



(†) "Traducir el traductor" en este contexto es equivalente a "ensamblar el ensamblador", porque el traductor (macro)ensamblador estará escrito en (macro)ensamblador, pero el bosquejo inicial de su diseño está en español, en forma de las recetas ya mostradas. Como se comenzará a ver ahora, el software de base está lleno de estos aparentes trabalenguas: traducir el traductor, ensamblar el ensamblador, cargar el cargador, compilar el compilador.

(††) Pero, ¿quién tradujo el traductor? ¿Podrá ser (otro) programa traductor? Si así fuera, ¿quién tradujo a ése? Después de analizar el problema se llega a la triste conclusión de que el primer traductor tuvo que haber sido traducido a mano por el diseñador: lo escribió en español —como en las recetas aquí mostradas— y lo fue paulatinamente traduciendo a lenguaje de máquina. Es una penosa labor, pero sólo debe efectuarse una primera vez. Ya teniendo el traductor T_1 traducido a lenguaje de máquina, bastará con ejecutarlo para que de allí en adelante parezca como si la computadora "entiende" directamente lenguaje macroensamblador. Claro que no es cierto, pero el software de base así lo hace parecer: es el concepto de *herramienta* llevado a un extraordinario nivel.

(†††) ¿Qué le parece: será fácil o difícil escribir un programa así? ¿Se podrá hacer? Por ahora sólo supóngalo y no se angustie tratando de pensar en los millones de detalles requeridos.

Exploremos la primera posibilidad. Supóngase que el programa fuente P_1 tiene cien renglones de longitud de la misma forma general que el programa de la página 194. Esto significa un enorme esfuerzo, ya que habría que colocar manualmente varios miles de ceros y unos en varios cientos de celdas de la memoria. Una vez realizado este penoso paso todavía quedaría la gigantesca molestia de tener que cargar también manualmente el traductor T_1 completo.

La segunda posibilidad es mucho más prometedora, y consiste en escribir un programa para que haga estos pasos por nosotros: lo llamaremos **cargador**. Las funciones de un cargador son relativamente sencillas, y consisten en extraer información objeto de algún medio externo a la memoria (disco o cinta magnética, por ejemplo) y colocarla en celdas sucesivas de la memoria, a partir de una celda preespecificada.

Necesidad del cargador

He aquí el diseño básico de un cargador:

```
! Programa cargador, primer acercamiento.
Localizar el dispositivo de memoria secundaria con los datos a cargar.
Averiguar la dirección de memoria a partir de la cual va a quedar cargado el
programa objeto, y considerarla como la celda actual.
Para cada "renglón" del programa objeto ejecutar lo siguiente:
    Determinar cuántas celdas de memoria se requieren para almacenar esos
    datos binarios.
    Depositar los datos en celdas contiguas de la memoria, a partir de la celda
    actual, e ir incrementando las direcciones para mantenerlas actualizadas.
```

Este programa debe traducirse a lenguaje de máquina para que entonces quede en la memoria de la computadora y se pueda ejecutar. El diseño inicial está en español pero, como antes, habrá que convertirlo en un programa escrito en ensamblador y luego ensamblarlo, para tener así una herramienta muy poderosa con la cual se podrá cargar en memoria cualquier programa objeto, siempre y cuando se le indique en cuál disco o cinta está almacenado, y a partir de cuál celda de memoria se desea que lo deposite.

Esto, sin embargo, da lugar a otro problema: ¿cómo se carga el cargador?

El dilema de la carga inicial de un programa

La respuesta ya no puede ser "por medio del cargador", porque está claro que éste no puede cargarse a sí mismo; para ello debería estar residente en memoria (para que la UCP lo pudiera ejecutar), ¡pero ese es precisamente el problema que se desea resolver!

Nos encontramos ante una situación que no puede solucionarse en términos de un programa. El problema inicial —romper el círculo vicioso recién descrito— recibe el curioso nombre de ***bootstrap***, que en inglés significa algo así como "el problema de tratar de levantarse del suelo tirando de las cintas de nuestras propias botas". Éste es, evidentemente, un problema que tiene "truco" y requiere de medios externos para poderse resolver.

El truco consiste en cargar a mano el cargador para evitar los obstáculos lógicos mencionados. Sólo que esta operación se tendría que repetir cada vez que se encienda la computadora, porque cuando se retira la corriente eléctrica la memoria pierde todos sus contenidos. La solución a este nuevo problema técnico (ya no lógico) es, a grandes rasgos, la siguiente.

Se escribe otro pequeño programa (que llamaremos "minicargador"), cuya única función consiste en *cargar el cargador*. Este miniprograma no será de uso general, y solamente servirá para extraer al cargador objeto de un lugar preestablecido (de una sección fija de un disco magnético predeterminado, por ejemplo) y depositarlo en una zona también preestablecida de la memoria central. Luego de hacer esto, el minicargador cede el control al cargador. Como este programa es de uso particular y cumple una sola función muy específica, será pequeño (unos pocos centenares de renglones fuente), por lo que será factible traducirlo a mano al lenguaje de máquina.

La situación aparece ahora así: cuando se enciende la computadora, se carga manualmente el minicargador objeto y entonces se ejecuta. Éste, a su vez, inmediatamente cargará

al cargador —que debe estar permanentemente en una posición fija y predeterminada del disco rígido—, y a partir de allí se podrá seguir con el proceso.

Por supuesto que en una computadora real el minicargador no se carga manualmente, sino utilizando una memoria especial tipo ROM, que al encender la máquina descarga su contenido en la memoria central. Como se dijo, al proceso de cargar el minicargador y con ello dar "vida" a la computadora se le conoce como *bootstrap*, o también como IPL (*Initial Program Load*, carga del programa inicial). Esto sucede cada vez que se enciende la computadora y es el requisito previo para su operación[÷].

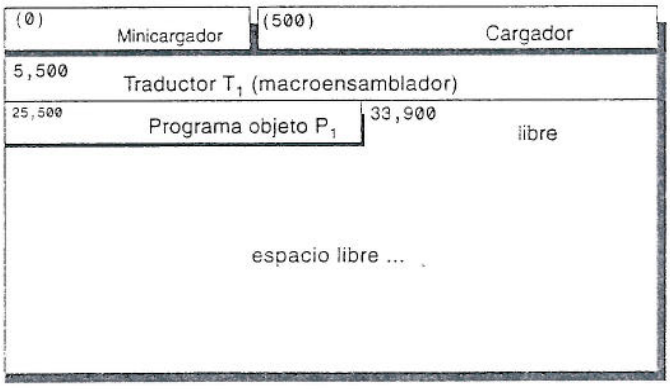
Los pasos para traducir y ejecutar el programa P_1 son, entonces:

0. Dar IPL, para cargar el minicargador (no es necesario si la computadora ya está operable).
1. Ejecutar el minicargador para que cargue al cargador, residente en una sección predeterminada del disco magnético.
2. Ejecutar el cargador para que cargue el ensamblador, residente en algún disco especificado.
3. Ejecutar el ensamblador para que lea el programa fuente (también almacenado en algún disco especificado) y lo traduzca a lenguaje de máquina. El archivo objeto resultante quedará en otra sección del disco magnético.
4. Ejecutar nuevamente el cargador, para que cargue el programa objeto recién producido por el ensamblador.
5. Ejecutar el programa objeto.

El paso 4 puede eliminarse si en el 3 se pide al ensamblador dejar el resultado de la traducción directamente en la memoria, aunque entonces no quedaría registro permanente del programa objeto porque se perderá al apagar la computadora.

Para configurar un ejemplo más completo supondremos los siguientes datos: el minicargador (originalmente contenido en ROM) residirá en memoria a partir de la celda cero, y mide 500 bytes. El cargador, ya traducido, mide 5,000 bytes y quedará colocado a continuación. El ensamblador objeto T_1 mide 20,000 bytes, y el programa objeto P_1 medirá, una vez ensamblado, 8,400 bytes.

Si se efectúan todas las operaciones de carga secuencialmente y utilizando celdas contiguas de memoria, ésta se verá así justo antes de ejecutar el paso 5 anterior:



(÷) El minicargador forma parte del BIOS ya mencionado en el capítulo 3. Además, siendo éste un pequeño programa de propósito especial, carece de "inteligencia" suficiente como para algo más que simplemente quejarse cuando por error el usuario enciende la computadora habiendo dejado un diskette dentro al apagarla, con lo que hace creer al minicargador que allí reside el cargador (y no en la pista 0 del disco rígido). Es el típico mensaje `Non-system disk or disk error`.

En este diagrama del llamado **espacio de direcciones** (que no está a escala) se indica cuáles programas objeto residen en la memoria, y a partir de cuál celda.

Veamos el proceso con más detalle. Como se dijo, el cargador debe estar en una sección fija y preestablecida del disco magnético (que llamaremos "D_0"), pues allí irá ciegamente a buscarlo el minicargador; supondremos que el ensamblador T_1 está en el disco "D-T_1" y que el programa fuente F_1 quedó en el disco "D-F_1"†. Además, cuando el traductor termina de ensamblar el programa fuente, deja el resultado objeto en el disco "D-O_1".

Pasos para la ejecución de un programa en una computadora

Asignaremos al contador de programa (CP) las direcciones ya descritas, mediante la notación

$$CP \leftarrow x \{F: Y, D: Z\}$$

que se lee "ejecutar el programa que inicia en la celda x de la memoria, que lee sus datos a partir de la Fuente Y, y los escribe en el Destino Z (la fuente y el destino pueden ser posiciones en la memoria o el disco, según sea el caso)".

Aunque esta notación pudiera parecer complicada, nos deberá servir de consuelo pensar que si no la empleáramos entonces los detalles sobre las localizaciones de los archivos en disco y las posiciones finales en la memoria deberían ser generadas a partir de la nada ...y eso sí sería complicado.

Con todo esto, los pasos anteriores serán entonces:

0. Dar *boot* (si es necesario). Con ello se lleva el minicargador del ROM a la memoria, a partir de la celda 0. Obsérvese que no se ha ejecutado nada todavía porque éste es un paso efectuado por medios eléctricos externos.

1. $CP \leftarrow 0 \{F: D_0, D: 500\}$
   (Ejecutar el minicargador para que vaya al disco D_0, extraiga de allí el cargador objeto y lo deposite en la memoria, a partir de la celda 500)

2. $CP \leftarrow 500 \{F: D\text{-}T_1, D: 5{,}500\}$
   (Ejecutar el cargador, para que lea lo que contiene el disco D-T_1 (el traductor) y lo deposite en la memoria a partir de la celda 5,500)

3. $CP \leftarrow 5{,}500 \{F: D\text{-}P_1, D: D\text{-}O_1\}$
   (Ejecutar el ensamblador para que lea el programa fuente del disco D-P_1, lo traduzca, y deje el programa objeto resultante en el disco D-O_1)

4. $CP \leftarrow 500 \{F: D\text{-}O_1, D: 25{,}500\}$
   (Ejecutar el cargador nuevamente, para que tome el contenido del disco D-O_1 y lo deposite en la memoria, a partir de la celda 25,500)

5. $CP \leftarrow 25{,}500 \{F: ?, D: ?\}$
   (Finalmente, ejecutar el programa objeto, que inicia en la celda 25,500. Como no se dijo nada muy específico sobre el programa P_1 (ahora ya O_1), no sabemos —ni nos preocupa en este ejemplo— de dónde va a tomar sus datos y cómo y hacia dónde va a enviar sus resultados.

Será responsabilidad del programa P_1/O_1 ejecutar la instrucción `ALTO` como último paso; de no hacerlo así, la computadora tratará luego de ejecutar lo que contenga la celda 33,900 (¿por qué?), con resultados impredecibles.

(†) Las notaciones "D-T_1" y similares se refieren a algunas secciones del disco o discos, para no preocuparnos ahora por las particularidades específicas y necesariamente precisas sobre cómo y dónde están almacenados esos archivos.

Se trata de un archivo magnético dividido en dos partes lógicas[*]: un encabezado y el programa objeto propiamente. El encabezado (*header*, en inglés) contiene —en ese preciso orden— la dirección de carga, la dirección inicial de ejecución, y un número de datos (bytes para nosotros) determinado por la diferencia entre las dos cifras anteriores.

Si se recuerda, la dirección de carga estuvo especificada en el programa fuente por la pseudoinstrucción `ORIGEN` y, en la misma forma, la dirección inicial de ejecución está marcada por la pseudoinstrucción `PROGRAMA`. Con ello, el ensamblador dispone de todos los elementos para preparar el módulo absoluto de carga y dejarlo grabado en el disco magnético antes de terminar su función de traducción y despedirse.

De esta forma, cuando el cargador toma el módulo absoluto del disco especificado (aún sigue siendo necesario, por supuesto, indicarle cuál es) puede por sí solo cargar todos sus contenidos, con excepción de los dos primeros bytes que le sirven de guía, porque el primero es la dirección de carga y el segundo es la dirección de la primera instrucción ejecutable, después de los posibles datos.

Por supuesto que los módulos absolutos de carga empleados en las computadoras reales tienen un formato más elaborado que éste, pero —como siempre— nos interesan más los porqués que los detalles. Por otro lado, confiamos en que el lector esté convencido de que todo esto *debe* necesariamente funcionar así (o en una forma similar), porque siempre nos guía el principio ya enunciado de *necesidad lógica*: las cosas son como son por alguna razón, que debe entonces servirnos de guía para averiguar cómo funcionan o, mejor aún, para re-inventarlas.

Con todo esto, el escenario queda listo para dar otro paso hacia arriba en la calidad de la comunicación con la computadora.

5-5 COMPILADORES

Los apartados anteriores han dado la posibilidad de escribir programas fuente para aplicaciones particulares en lenguaje (macro)ensamblador, pues se dispone de un traductor T_1 que los convertirá al lenguaje de máquina. Se puede también dejar estos programas residentes en el disco y llevarlos a la memoria por medio del cargador, también ya diseñado. Estamos, pues, en una situación bastante buena, con un cierto nivel de comunicación con la computadora. No obstante, deseamos aún más capacidad y una mayor flexibilidad para transmitir los requerimientos a la máquina; es decir, establecer comunicación con la computadora en un lenguaje más parecido al nuestro (y, por tanto, menos parecido al lenguaje de unos y ceros).

¿Qué se puede hacer?

Exploremos las posibilidades de intentar comunicarnos con la computadora usando un lenguaje de más alto nivel expresivo. Cuando se dice "lenguaje de alto nivel" se piensa en uno que permita, con una sola orden, expresar cosas complejas: esto es, un lenguaje dotado de una estructura capaz de sostener esa posibilidad.

Necesariamente debemos analizar entonces el problema de la traducción de lenguajes de alto nivel expresivo, porque intentamos establecer comunicación con la máquina en un lenguaje de este tipo y esperar que reciba nuestros mensajes mediante un traductor (T_2) que los convierta a su lenguaje de máquina.

(*) Es decir, la división no es física —es un solo archivo— sino que sigue dos criterios diferentes. Recuérdese cómo en este contexto la palabra "lógico" se refiere a algo simbólico o que no está en forma física.

El **compilador** (que es el nombre de este nuevo traductor T_2) trabajará sobre las cadenas de entrada (esto es, sobre los renglones que componen el programa fuente), para traducirlos al lenguaje ensamblador. Luego de esto, el traductor ensamblador (T_1) obtendrá finalmente el mismo programa en lenguaje objeto.

Como la teoría matemática requerida para el diseño de un compilador rebasa los límites de este curso introductorio (pues se trata de la subárea 2.4 de los Modelos Curriculares), aquí sólo se estudiará el proceso de compilación desde un punto de vista muy general.

Para comenzar a entender el nivel de complejidad del problema analizaremos el caso de traducir frases del español al inglés; por ejemplo: "la casa es azul".

El problema de la traducción

Si usamos un diccionario, encontraremos que "la" se traduce por *the*, "casa" por *house*, "es" por *is* y "azul" por *blue*, por lo que la frase ya traducida será *the house is blue*.

Ojalá esté claro, sin embargo, que esto no fue sino una afortunada casualidad. Inténtese un ejemplo más complejo y de inmediato se encontrarán las dificultades inherentes a todo proceso de traducción de lenguajes de alto nivel expresivo; es decir, un simple diccionario no basta para lograr una buena traducción y, a veces, ni siquiera para lograr algo que medianamente se asemeje a la frase original. Esto se debe, por supuesto, a la **estructura del lenguaje**, definida por su gramática, y a un mundo de significados que no es reducible a un diccionario.

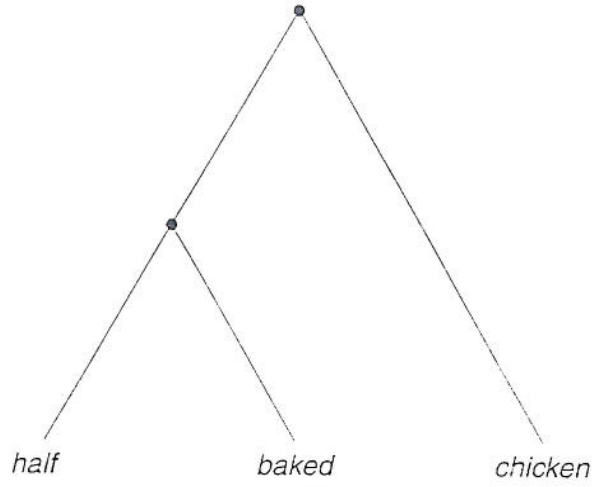
Noam Chomsky

Así, aunque se haya traducido la frase "sufragio efectivo, no reelección" por medio de un diccionario, nos veremos en dificultades casi insalvables cuando se intente traducir la frase "sufragio efectivo no, reelección"; aunque tiene exactamente los mismos componentes (las mismas cuatro palabras y una coma) significa nada menos que lo contrario que la frase original. Ningún diccionario será suficiente para dar cuenta de la diferencia, ya que éstos trabajan únicamente con palabras aisladas, sin tomar en cuenta la estructura gramatical.

Noam Chomsky, lingüista del Instituto Tecnológico de Massachusetts (MIT)†, que en 1956 publicó un estudio ya clásico sobre gramáticas formales (esto es, estudiadas desde un punto de vista matemático), proponía otro ejemplo, (en inglés), para poner en evidencia que en toda frase de un lenguaje existe como respaldo una estructura que le da forma y sentido. ¿Qué significa la frase *Half baked chicken*? Puede significar tanto "medio pollo cocido" como "pollo medio cocido", que de ninguna manera dice lo mismo. Esta es una clásica frase ambigua. La ambigüedad está determinada por la estructura que respalda la frase, como a continuación se explica.

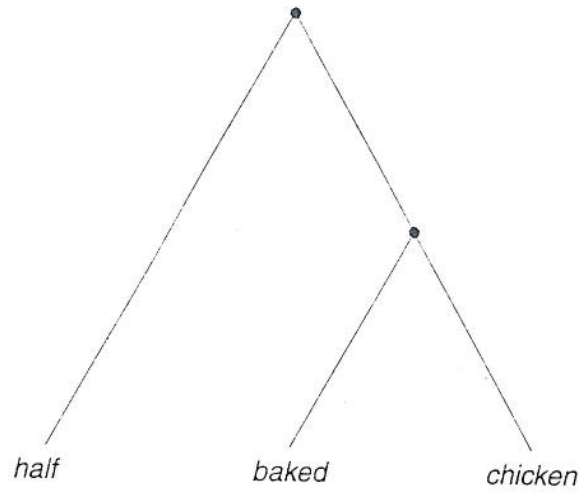
Diagramas de estructura gramatical

Para el caso de "pollo medio cocido", la estructura de la frase es como sigue:



(†) Quien además es uno de los más honestos intelectuales estadounidenses, y mordaz crítico de la política exterior de su país desde una perspectiva más que inteligente y liberal.

Mientras que para "medio pollo cocido" es:



El proceso de la traducción

Sin preocuparnos todavía por el significado de los diagramas, sí diremos que existen varias maneras de agrupar las palabras (por medio de lo que en lingüística se conoce como la "estructura profunda"), y que estas formas le confieren significados distintos a la misma frase.

Como estamos ante un problema no trivial, a continuación se comenzará a explorar lo que se requiere para poder traducir frases dotadas de estructura interna.

Análisis lexicográfico

El paso inicial consiste en **reconocer** todos y cada uno de los **símbolos** aislados constituyentes de la frase; lo cual, a su vez, implica reconocer las letras (y signos de puntuación) y reconocer las palabras. Obsérvese que reconocer no necesariamente significa entender; para reconocer un símbolo sólo se requiere buscarlo (y encontrarlo) en un diccionario previamente especificado.

Esta primera etapa se conoce como **análisis lexicográfico**. Su tarea central consiste en separar los **componentes léxicos** (o *tokens*) de entre el conjunto de símbolos fuente.

Esto es, en un renglón normal coexisten símbolos de diversas clases (letras, dígitos, símbolos de puntuación, blancos y caracteres especiales) aunque sean invisibles, y es necesario aislar los componentes sintácticos de este conglomerado de caracteres. Para nosotros es obvio que la frase "uno, dos, tres" consta de tres palabras, pero en realidad contiene catorce símbolos diferentes que es necesario agrupar de alguna manera. Todos aprendimos a hacer análisis léxicos de manera intuitiva durante el largo proceso de enseñarnos a leer, pero dentro de un compilador se requiere definir esta tarea con toda precisión en forma de un programa ejecutable, para lo cual se emplea un modelo matemático ya mencionado en el capítulo 1, el autómata finito. Un autómata así es una función matemática con capacidad de reconocer los grupos de caracteres que constituyen un componente léxico, o bien de marcar un error si no están bien construidos. Más adelante en el texto hay otras referencias a este tipo de construcciones formales.

Análisis sintáctico

Una vez concluido exitosamente ese análisis se llega a una parte muy interesante, llamada **análisis sintáctico**, cuya finalidad es *encontrar la estructura gramatical de la frase* formada por los elementos aislados ya reconocidos.

No obstante que los humanos podemos realizar este tipo de análisis igualmente en forma intuitiva e inmediata (aunque no sepamos gramática), dentro de un compilador se

requieren métodos matemáticos para especificar cómo lograrlo. La idea general consiste en diseñar un programa que trate de acomodar estructuras gramaticales predefinidas para las frases objeto del análisis, guiándose por medio de las palabras que la componen.

El análisis sintáctico sólo fue entendido formalmente hasta hace algunos años. Los analizadores sintácticos del tipo de los empleados por un compilador (llamados *parsers* en inglés) se dividen en dos grandes familias: los que funcionan en forma "ascendente" y sus contrarios, en forma "descendente". Para poder discutir estos puntos, aunque sea mínimamente, será necesario mencionar antes algunos elementos sobre la teoría de las gramáticas y los lenguajes formales†. Como se dijo, esta teoría nace en la década de 1950 y trata sobre las propiedades de ciertas construcciones formales llamadas **gramáticas**, que no son sino especificaciones matemáticas de la estructura de los lenguajes formales, similares a la gramática que se supone todos aprendimos en la escuela elemental para describir la estructura del lenguaje ordinario empleado para comunicarnos.

El problema de la comunicación

Para caracterizar el problema de la comunicación se puede pensar en una gramática como un generador de palabras (o frases), que luego llegarán a un reconocedor que se encargará de decidir si una frase es "hija legítima" de cierta gramática o no; esto es, el reconocedor deberá efectuar análisis sobre las frases recibidas para proceder luego a interpretarlas.

A continuación hay un ejemplo muy elemental, sobre un subconjunto de la gramática del español.

Todo hablante de nuestro idioma reconocerá que la frase "la casa es azul" es correcta desde cualquier punto de vista. Para analizar el porqué de ello se partirá del conocimiento de que una oración (o frase) está compuesta de palabras (que a su vez constan de letras). Las palabras, sin embargo, se clasifican en ciertos tipos gramaticales agrupadas luego en construcciones más complejas. En suma:

—*la* es un ARTÍCULO.
—*casa* es un SUSTANTIVO.
—*es* es un VERBO.
—*azul* es un ADJETIVO.

y además:

—Un ARTÍCULO seguido de un SUSTANTIVO es una FRASE NOMINAL.
—Un VERBO seguido de un ADJETIVO es una FRASE VERBAL.
—Una FRASE NOMINAL seguida de una FRASE VERBAL es una ORACIÓN.

O más escuetamente,

1) <ORACIÓN>	→	<FRASE NOMINAL> <FRASE VERBAL>
2) <FRASE NOMINAL>	→	<ARTÍCULO> <SUSTANTIVO>
3) <FRASE VERBAL>	→	<VERBO> <ADJETIVO>
4) <ARTÍCULO>	→	"la"
5) <SUSTANTIVO>	→	"casa"
6) <VERBO>	→	"es"
7) <ADJETIVO>	→	"azul"

Estas siete reglas configuran una primera gramática formal, con la que trabajaremos un poco. Obsérvese el uso de algunos símbolos nuevos: con las llaves triangulares se

(†) En el capítulo 6 se dedica una sección a este importante tema, y se proponen referencias bibliográficas sobre la teoría matemática requerida por los compiladores.

encierran palabras de la gramática que se llamarán **no terminales**, mientras que con las comillas se distinguen las palabras **terminales**. Estas últimas son las que forman las frases terminales, o sea, los elementos finales de una construcción gramatical, que además son los únicos que se muestran al mundo exterior. Una frase u oración tiene una estructura interna, y los elementos empleados para definir esta estructura profunda de la frase son precisamente los no terminales.

El otro elemento nuevo es la flecha, que liga miembros izquierdos (no terminales), con miembros derechos (que pueden ser terminales o no).

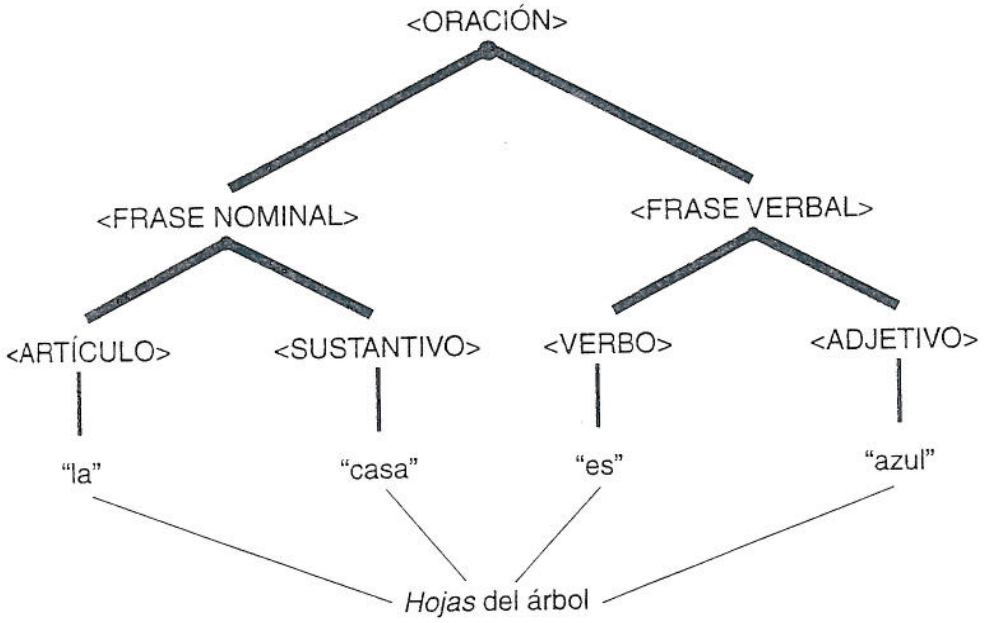
Por ejemplo, la regla 6)

<VERBO> → "es"

se lee: "el no terminal VERBO produce el terminal "es".

El lector podrá comprobar que la frase "la casa es azul" tiene la siguiente estructura:

Un primer análisis sintáctico



Ahora se pueden entender un poco mejor los conceptos del análisis sintáctico. Si iniciamos en la parte superior de este diagrama (conocido como **árbol sintáctico**) y se intenta aplicar una a una las siete reglas de producción se obtendrá lo siguiente:

<ORACIÓN> ⇒ <FRASE NOMINAL> <FRASE VERBAL>
⇒ <ARTÍCULO> <SUSTANTIVO> <FRASE VERBAL>
⇒ "la" <SUSTANTIVO> <FRASE VERBAL>
⇒ "la" "casa" <FRASE VERBAL>
⇒ "la" "casa" <VERBO> <ADJETIVO>
⇒ "la" "casa" "es" <ADJETIVO>
⇒ "la" "casa" "es" "azul"

(La doble flecha se lee "genera mediante la aplicación de una regla de producción", o simplemente "genera".)

Obsérvese que se partió del tope (o raíz) del árbol y se llegó a las hojas terminales. Éste fue el primer **análisis sintáctico descendente**; aunque trivial, es representativo.

¿Cómo será el análisis sintáctico ascendente? Hagamos lo siguiente:

"la" ⇐ <ARTÍCULO>
"casa" ⇐ <SUSTANTIVO>
<ARTÍCULO> <SUSTANTIVO> ⇐ <FRASE NOMINAL>
"es" ⇐ <VERBO>
"azul" ⇐ <ADJETIVO>
<VERBO> <ADJETIVO> ⇐ <FRASE VERBAL>
<FRASE NOMINAL> <FRASE VERBAL> ⇐ <ORACIÓN>

Ahora se procedió exactamente a la inversa: partiendo de los elementos terminales se encontró un camino hasta la raíz del árbol. Aunque en apariencia el análisis ascendente sólo es el inverso del descendente, en realidad resulta mucho más complejo. La razón de lo anterior reside, intuitivamente, en que el problema central del análisis descendente consiste en escoger alguna regla y aplicarla, partiendo de su miembro izquierdo (que consta de un solo elemento no terminal), mientras que en el caso contrario se debe escoger alguna regla y "desaplicarla"; pero ahora ya no existe un solo elemento del lado derecho, sino varios y, por tanto, aumenta la gama de combinaciones posibles.

De hecho, hace apenas pocos años que se encontraron métodos eficientes para realizar análisis sintácticos ascendentes, mientras que los descendentes fueron inventados hace más de cuarenta†.

Luego de esta somera revisión del análisis sintáctico podemos pasar a la siguiente etapa en el proceso de la traducción.

Análisis semántico

Una vez terminada exitosamente la fase sintáctica o gramatical se está ya en posición de entender el significado de la frase, por medio del **análisis semántico**.

Disquisiciones sobre el lenguaje

Aquí lo importante es determinar la coherencia entre lo dicho por medio del lenguaje, y los elementos del mundo a los que se está haciendo referencia. Nosotros efectuamos el análisis semántico también en forma intuitiva e inmediata, lo cual nos permite darle sentido a las frases y manejar ese maravilloso vehículo llamado lenguaje.

Las funciones del lenguaje son múltiples, y cubren desde la elemental necesidad de describir el mundo para propósitos de supervivencia cotidiana hasta los juegos de palabras y la significación poética.

En todos esos casos se sigue manteniendo, aunque con diversos matices, la razón semántica: dotar a las palabras con la cualidad de mundo; dar contenido real a los continentes huecos representados por los meros símbolos lingüísticos.

Por ejemplo, el sentido de la frase "la casa es azul" es directo, porque en el mundo sí existen casas con el atributo denotado por esas palabras; sin embargo la frase "la casa es triste" maneja contenidos que ya no tienen un referente tan inmediato, porque las casas que la experiencia nos muestra en el mundo son azules, rojas, grandes o pequeñas, pero no son directamente tristes. Sin embargo, exhibiríamos una sensibilidad muy pobre si nos negamos a entender que una casa pueda efectivamente (aunque no en forma directa) ser triste.

(†) Las ideas originales sobre análisis sintáctico se deben a Donald Knuth, en el artículo "On the Translation of Languages from Left to Right", aparecido en la revista *Information and Control*, octubre, 1965; pero no fue sino hasta la publicación del artículo "An Efficient Context-Free Parsing Algorithm", que es un resumen de la tesis doctoral de Jay Earley, publicado en *Communications of the Association for Computing Machinery* de febrero de 1970, cuando se contó con un método práctico, que luego se refinó más.

Pero hay más, mucho más. El poema "Piedra negra sobre una piedra blanca" del enorme escritor peruano César Vallejo (1892-1938) inicia con la extraordinaria frase:

> *Me moriré en París con aguacero,*
> *un día del cual tengo ya el recuerdo.*†

O bien, ¿qué efecto nos causan estos elaborados pasajes, tomados casi al azar del libro *El ser y la nada*, obra filosófica cumbre del escritor y literato Jean Paul Sartre (1905-1980)?††:

> *La conciencia es un ser para el cual en su ser es cuestión de su ser en tanto que este ser implica un ser otro que él mismo.* (pág. 31)
>
> *Así encontramos nuevamente, en otro plano, una necesidad ontológica que habíamos señalado con motivo de la existencia de mi cuerpo para mí: la contingencia del para-sí, decíamos, es la recuperación perpetuamente trascendida y perpetuamente renaciente del para-sí por el en-sí sobre fondo de nihilización primera.* (pág. 431)

Por otro lado, del poeta Raúl Bañuelos (Editorial Universidad de Guadalajara, 1989) leemos:

> *La lluvia ama del río lo que tiene de agua*
> *y por eso llueve.*

Pero basta. Para el caso de un compilador, claro, "el mundo" es la computadora, sus registros, sus celdas de memoria, etc. El análisis semántico efectuado por un compilador averigua, por ejemplo, si una expresión dentro de un programa significa algo válido, y no pide hacer una operación aritmética sobre una letra, cosa que no tendría sentido.

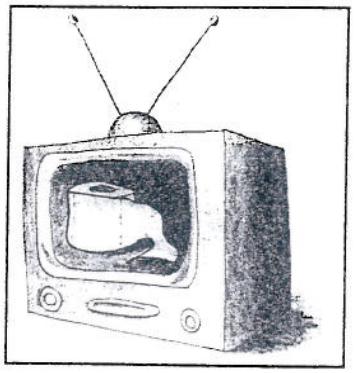
Una vez analizada a fondo una frase, y cuando ya se ha determinado su validez lexicográfica, sintáctica y semántica, llega por fin el momento de traducirla. Es importante observar que la traducción es lógicamente posterior a las operaciones de análisis. En algunas referencias sobre compiladores se llama fase semántica a la parte encargada de lo recién descrito, pero además suelen incluirse también allí las funciones de generación de código que ahora se explican.

Generación y optimización de código

La traducción —o **generación de código**— busca representar la frase fuente original en términos de elementos de un lenguaje mucho más sencillo, que ya no está dotado de estructura. O sea, precisamente, llegar a traducir la frase fuente al lenguaje de máquina (o por lo menos al lenguaje ensamblador).

En este nuevo ejemplo se muestra el concepto. Supóngase que se desea efectuar la operación `C = A + B` donde las letras representan variables de tipo numérico. Un renglón del programa fuente será, entonces,

```
C = A + B
```



(†) Si el lector no encuentra sentido en esta frase, respetuosamente le sugerimos alejarse algunos años de la televisión e Internet y sus tristes influencias.

(††) Este complejo libro de 1943, maravillosamente denso y extenso (casi 800 páginas), subtitulado "Ensayo de ontología fenomenológica", Editorial Losada, Buenos Aires, 1972, contiene la fundamentación teórica de la filosofía del existencialismo. Sartre ganó, y rechazó, el Premio Nobel de Literatura de 1964.

Resumen

La inteligencia artificial tiene muchas definiciones. La mayor parte de la investigación en IA se centra en conseguir que las computadoras hagan aquellas cosas que generalmente las personas hacen mejor. Algunos investigadores de IA intentan simular el comportamiento inteligente del ser humano, pero la mayoría intenta diseñar máquinas inteligentes independientemente de cómo piensan las personas. Generalmente, la investigación en IA implica trabajar en problemas con dominios limitados, en lugar de intentar abordar problemas más grandes e indefinidos. Los programas de IA emplean varias técnicas, como la búsqueda, la heurística, el reconocimiento de patrones y el aprendizaje de la máquina, para lograr sus objetivos.

Desde un punto de vista práctico, la comunicación en lenguaje natural es una de las áreas más importantes del estudio de la IA. Los programas de lenguaje natural que tratan con un subconjunto del lenguaje se utilizan en aplicaciones que van desde los programas de traducción hasta las interfaces en lenguaje natural. Pero no hay programas capaces de manipular el tipo de texto de lenguaje natural sin restricciones que las personas utilizan a diario. Los programas en lenguaje natural se ven confundidos por el extenso vocabulario, la sintaxis farragosa y la semántica (los significados de las palabras) ambigua del inglés.

Los investigadores en IA han desarrollado varios esquemas para representar el conocimiento en las computadoras. Una base de conocimiento contiene hechos y un sistema para determinar y cambiar la relación entre esos hechos. Las bases de conocimiento actuales sólo resultan prácticas para representar dominios estrechos de conocimiento, como el conocimiento de un experto en una tema en particular. Los sistemas expertos son programas diseñados para reproducir el proceso de toma de decisiones de los expertos humanos. Un sistema experto incluye una base de conocimiento, un motor de deducción para aplicar las reglas lógicas a los hechos de una base de conocimiento, y una interfaz humana para interactuar con los usuarios. Una vez construida la base de conocimiento (normalmente en base a las entrevistas y observaciones de los expertos humanos), un sistema experto puede proporcionar un asesoramiento que rivaliza con las sugerencias humanas en muchas situaciones. Las personas utilizan satisfactoriamente los sistemas expertos en variedad de aplicaciones científicas, empresariales y de otra índole.

El reconocimiento de patrones es un área importante de la investigación en IA que implica la identificación de patrones repetitivos en los datos de entrada. La tecnología del reconocimiento de patrones se encuentra en el corazón de la visión por computadora, la comunicación por voz y otras importantes aplicaciones de la IA. Todas estas variadas aplicaciones utilizan técnicas parecidas para aislar y reconocer patrones. Las personas son mejores que las computadoras en el reconocimiento de patrones, en particular porque el cerebro humano puede procesar masas de datos en paralelo. Las computadoras de las modernas redes neuronales se diseñan para procesar datos de la misma forma que lo hace el cerebro humano. Muchos investigadores creen que las redes neuronales, a medida que crezcan en tamaño y sofisticación, ayudarán a que mejore el rendimiento de las computadoras en muchas tareas complejas.

Un robot es una máquina controlada por una computadora diseñada para llevar a cabo tareas manuales específicas. Los robots tienen periféricos de salida para manipular sus entornos y sensores de entrada que les permiten acciones autocorrectivas en base a la información del exterior. Los robots ejecutan muchas tareas peligrosas y tediosas, en muchos casos superando a los trabajadores humanos. A medida que avance la tecnología robótica, los trabajadores artificiales se encargarán de muchos de los trabajos humanos tradicionales.

A pesar de las numerosas dificultades a las que se enfrentan los investigadores en IA al intentar producir máquinas realmente inteligentes, muchos expertos creen que las personas acabarán creando seres artificiales más inteligentes que sus creadores (una perspectiva con implicaciones asombrosas).