PRÓLOGO

Los conceptos de compiladores en muchas ocasiones se han considerado como algo esotérico. Esto necesariamente no tiene que ser así. De hecho, todas las investigaciones que han desembocado en los conceptos que ahora conocemos son un gran arsenal que los ingenieros de sistemas deberían conocer. Existen múltiples facetas de la programación cotidiana que se pueden beneficiar de los descubrimientos presentados en los estudios acerca del tema.

El objeto de este libro es poner al alcance del lector los *elementos básicos de los compiladores*. Por esta razón, sólo se tratan aquellos temas fundamentales sobre los cuáles se ha construido paulatinamente conceptos muchos más avanzados, pero que escapan al alcance del libro. También se ha tratado de ejemplificar la puesta en práctica de cada uno de los conceptos a medida que se presentan los mismos. El lector no habrá sido exigente con el libro si no intenta trasladar los ejemplos y programas a las herramientas de programación que posee y conoce. Esto permitiría observar de manera directa la puesta en escena de los conceptos y poder cuestionarlos, ampliarlos y apropiarse de ellos. De lo contrario, el lector *sabrá más* del tema pero no lo *comprenderá más*. En muchas ocasiones la existencia de demasiados hechos representa un obstáculo para la comprensión tanto como la existencia de demasiados pocos. Por esta razón se ha tratado de crear un "mínimo suficiente" en relación con el contenido del libro y el resultado está en sus manos.

Vistazo General del Contenido del Libro

Capítulo 1. Conceptos Básicos.
Este capítulo define qué es un compilador y presenta una visión general de todas las fases que componen un compilador. Así mismo se muestra un ejemplo completo del proceso de compilación que recorre todos los temas que serán tratados en los capítulos posteriores.

Capítulo 2. Análisis Léxico.
El capítulo 2 trata de las funciones básicas que debe realizar un analizador léxico. Adicionalmente se presentan y discuten varias alternativas para implementar un analizador léxico. El capítulo concluye con una discusión acerca de la tabla de símbolos y lo que debería contener.

Capítulo 3. Análisis Sintáctico.
Este capítulo inicia con la presentación de las gramáticas libres de contexto. Durante esta explicación se menciona la construcción de los árboles de

análisis sintáctico y el papel que juegan en la construcción de un analizador sintáctico. Seguidamente se muestra como construir un analizador sintáctico descendente, que es la única estrategia de análisis que presentaremos en el libro. El capítulo continua con las tablas sintácticas, para darle luego paso al tema de cómo manejar los errores en el proceso de compilación. Se concluye el capítulo con la presentación de la traducción dirigida por sintaxis, la cual provee el soporte básico para comprender el apéndice A. Este es, con mucho, el capítulo central del libro.

Capítulo 4. Análisis Semántico.
El capítulo 4 muestra las situaciones sujetas a un análisis semántico y cómo modificando los conceptos vistos en el tema de traducción dirigida por sintaxis se puede construir un comprobador sencillo de tipos.

Capítulo 5. Generación de Código.
En este capítulo habla de algunas representaciones intermedias y presenta el código de tres direcciones. Adicionalmente se define el lenguaje intermedio que se utilizará para representar los programas fuentes y se muestra su utilización con un ejemplo completo.

Capítulo 6. Optimización y Generación de Código.
El capítulo 6 presenta dos maneras sencillas de optimizar el código: las simplificaciones algebraicas y la utilización del menor número posible de variables temporales. Finalmente el capítulo concluye con los algoritmos propuestos para llevar a cabo estas dos clases de optimizaciones.

Apéndice A. Evaluador de expresiones.
Este apéndice muestra un evaluador de expresiones implementado en pascal. El programa mostrado sirve para ilustrar la puesta en escena de prácticamente la mayoría de temas discutidos en el libro.

Apéndice B. Gramática propuesta para un compilador.
Para aquellos lectores exigentes, que deseen cumplir con lo exhortado al inicio de este prólogo, se presenta una gramática BNF para un lenguaje hipotético que podrían utilizar para construir su propio compilador, como lo plantea el mismo apéndice.

1 CONCEPTOS BASICOS

1.1. Qué es un Compilador ?

El mundo de la computación ha mejorado notablemente con la llegada de los lenguajes de alto nivel. Estos últimos permiten que los problemas se resuelvan más fácilmente con motivo de que la solución se puede plantear en términos más cercanos al problema mismo. No obstante, una vez alcanzada dicha solución en términos del lenguaje de alto nivel esta se debe trasladar a lenguaje de máquina, que es el lenguaje que puede finalmente ejecutar el computador.

El proceso de traducir el programa escrito en lenguaje de alto nivel a un formato ejecutable por el computador se conoce como *compilación*. Por lo tanto, diremos que un *compilador* es un programa que lee un programa escrito en un lenguaje y lo traduce a un programa equivalente en otro lenguaje.

Nótese que el destino final de la traducción no necesariamente tiene que ser lenguaje máquina, aunque es lo más corriente. Es también normal utilizar compiladores que generan otro tipo de salida, para ser usada con fines diferentes a la ejecución en lenguaje máquina. Por ejemplo, un *intérprete* puede procesar un programa fuente escrito en un lenguaje de alto nivel, de la misma manera que lo hace un compilador. Para lograr este fin el compilador produce una versión interna especial del programa fuente, en lugar de traducirla a lenguaje máquina. Una vez generada dicha versión especial interna del programa fuente, el intérprete ejecuta las operaciones especificadas por el programa. Un intérprete normalmente es un conjunto de subrutinas cuya ejecución está a cargo de la forma especial interna del programa.

1.2. Fases de un Compilador

Por razones de diseño, la construcción de un compilador se divide en varios pasos o fases. Cada fase resulta más simple y, por lo tanto, más fácil de comprender, escribir y probar. El proceso de compilación, en términos generales, se divide en dos procesos genéricos:

- *Análisis* **:** El cual hace referencia al hecho de que para analizar la correctitud de un programa fuente, escrito por un programador, inicialmente se debe dividir en los diferentes componentes que lo conforman y, a partir de allí, verificar que el uso del lenguaje cumpla con las especificaciones establecidas para el compilador.

- *Síntesis* **:** Una vez que se ha ejecutado el análisis de manera exitosa se procede a agrupar los componentes, que conforman el programa fuente, para construir frases con sentido con el fin de generar una salida. Ya hemos discutido que la salida puede ser en lenguaje máquina o algún otro lenguaje destino que se considere conveniente.

El proceso de *análisis* comprende varias fases que son:

- *Análisis léxico:* Esta fase es la encargada del trabajo de lectura y exploración misma del programa fuente. Por exploración queremos decir que el analizador léxico lee el programa fuente, y lo fracciona en componentes que tienen sentido para el lenguaje de programación que se esté considerando.

- *Análisis sintáctico:* A esta fase le corresponde evaluar que el programa fuente escrito realmente cumpla con las especificaciones del lenguaje definido para el compilador. Para ello normalmente el programa fuente debe reflejar una estructura especial. La estructura que debe cumplir un programa correctamente escrito se expresa con reglas, que pueden ser recursivas o no, las cuales se denominan con el nombre de *gramática*. Es importante anotar que esta es una de las fases más importante de la compilación.

- *Análisis Semántico:* Esta fase se dedica a determinar si todos los componentes del programa están siendo usados de manera válida, para el contexto en el cual aparecen. Es decir, se debe analizar los componentes colindantes a cada componente siendo analizado, antes de determinar que las operaciones ejecutadas sobre el mismo estén dentro de las operaciones permitidas, por el lenguaje, para dicho tipo de situaciones.

Una vez que el programa fuente ha sido analizado completamente se puede tener la certeza que está correctamente escrito. Sólo queda faltando generar algún tipo de salida para que el ciclo de compilación quede completo. Las fases restantes hacen una *síntesis* del programa fuente para generar la salida. Estas fases son:

- *Generación de Código Intermedio:* La mayoría de los compiladores modernos intentan optimizar, hasta donde sea posible, el código que generan. Para lograr este objetivo los compiladores analizan internamente el programa fuente, y tratan de generar secuencias de instrucciones equivalentes a las del programa fuente, pero que se ejecuten más rápidamente. O reemplazan instrucciones para hacer un uso

más eficiente de la memoria. Esta tarea se vería facilitada enormemente si el programa fuente estuviera escrito de cierta forma en especial. Como en la mayoría de los casos esto no es así, el objetivo de la fase de generación de código intermedio es generar una representación intermedia del programa fuente, para que sea usada posteriormente por el optimizador de código.

- *Optimización del código*: El objetivo de esta fase es tratar de mejorar el código fuente escrito para que sea más rápido de ejecutar, o use de manera eficiente los recursos de la máquina. Este proceso se apoya en el código intermedio que fue generado en la fase de generación del mismo. Un código intermedio bien diseñado permite manipular la mayoría de las instrucciones del programa fuente de una manera eficiente, para lograr los objetivos de esta fase. Lo anterior no quiere decir que exista la necesidad exclusiva de tener un código intermedio antes de optimizar un programa escrito, pero con frecuencia es mucho más complicado optimizar el código basándose en el programa fuente tal como fue escrito por el programador.

- *Generación de código*: El proceso de generación código es el que construye la salida, es decir, genera el código de máquina que corresponde al programa fuente. Como se hizo la anotación más arriba, cuando hablamos de los intérpretes, existen múltiples opciones para la salida ya que esta depende del objetivo del compilador, y no necesariamente siempre tiene que ser el código de máquina.

Obsérvese que existen muchas ventajas de ejecutar el proceso de construcción de un compilador en fases. Una de las más atractivas es que si se deseara hacer que el compilador generara lenguaje de máquina para diferentes arquitecturas de procesadores, básicamente toda la estructura del compilador quedaría intacta, y sólo existiría la necesidad de reescribir la fase de generación de código. En el caso de los intérpretes esta necesidad desaparece ya que nunca hace falta generar lenguaje máquina. La transportabilidad del compilador en este caso está supeditada a que el lenguaje, en el cuál fue escrito el compilador, esté disponible para la arquitectura de procesadores a los cuales se desea trasladar el mismo.

Para la ejecución de las diferentes fases que hemos mencionado en ocasiones es necesario recorrer el programa fuente en una o varias ocasiones. Un compilador de *una sola pasada* es aquel que sólo necesita recorrer el programa fuente en una sola oportunidad para realizar la compilación. No obstante, cuando se desea generar código optimizado por razones obvias se necesita *más de una pasada* para lograr este fin. Lo frecuente es encontrar en el mercado compiladores de dos o más pasadas destinados a ser usados en

ambientes de programación profesionales. Un compilador que no produzca código optimizado casi siempre es de una sola pasada, como por ejemplo Pascal.

Se hace necesario, para una adecuada distribución del trabajo que debe realizar el compilador, que existan algunas actividades de soporte para los procesos de análisis y síntesis. Estas actividades de soporte son:

- *Administración de la tabla de símbolos*: Una tabla de símbolos es una estructura de datos, v.gr.: una lista doblemente enlazada o una tabla en memoria, que se utiliza para almacenar la información concerniente a las variables definidas en el programa fuente. El objetivo de la tabla de símbolos es facilitar la consulta de la información referente a las variables, sin necesidad de hacer retrocesos en la lectura del programa fuente. Por tal razón, existen actividades propias de inserción y búsqueda en la tabla de símbolos que ameritan que estén localizadas en su propia área del compilador.

- *Detección e información de errores:* En cada fase de la compilación se pueden detectar errores. Uno de los objetivos básicos de la compilación es tratar de detectar el mayor número posible de errores, antes de que se detenga la compilación. Para estos efectos se debe informar del error y luego tratar de manipularlo de alguna forma, para que el compilador pueda continuar con el proceso de compilación. Todos conocemos la poca utilidad de los compiladores que detienen el proceso de compilación ante la presencia del primer error encontrado en el programa fuente.

De una manera gráfica se pueden observar las fases del compilador en la figura 1-1.

1. 3 Un Ejemplo Completo del Proceso de Compilación

Para ejemplificar de manera completa el proceso de compilación considérese el problema de encontrar los términos de la secuencia de fibbonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21... y determinar el primer término que excede a 10 en dicha secuencia.

Primero crearemos un diagrama de flujo de datos que presenta la solución al problema planteado (ver figura 1-2).

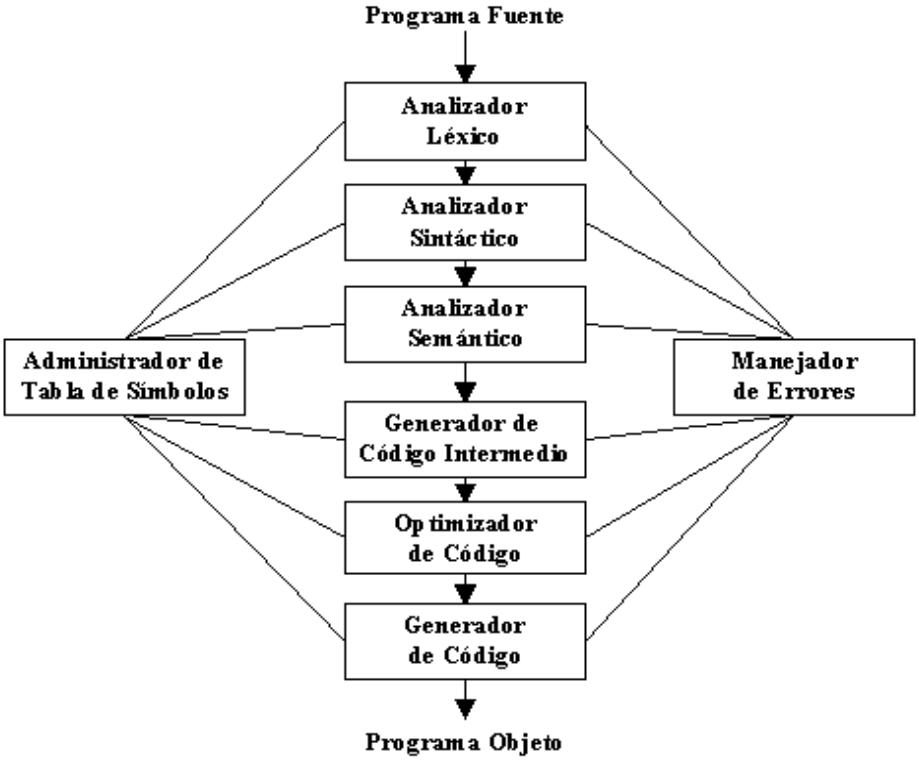


Figura 1-1. *Fases de un compilador*

Una vez hallada la solución se puede trasladar la solución a un lenguaje de programación, de los tantos disponibles en el mercado. Para simplificar la exposición supondremos que eso ya ha sido hecho, ya que no es relevante para lo que sigue a continuación. Con el fin de ilustrar las fases de compilación diremos que se ha creado un archivo de texto, que contiene la solución de programación en el lenguaje elegido que se desea compilar, entonces el proceso de compilación seguirá la secuencia general que se describe a continuación.

El analizador léxico tomará el archivo texto (programa fuente) y comenzará a leer cada una de las líneas hasta finalizar de explorarlo completamente. En determinado momento a la luz del ejemplo que traemos estará leyendo la siguiente línea:

Suma := ultimo + penultimo

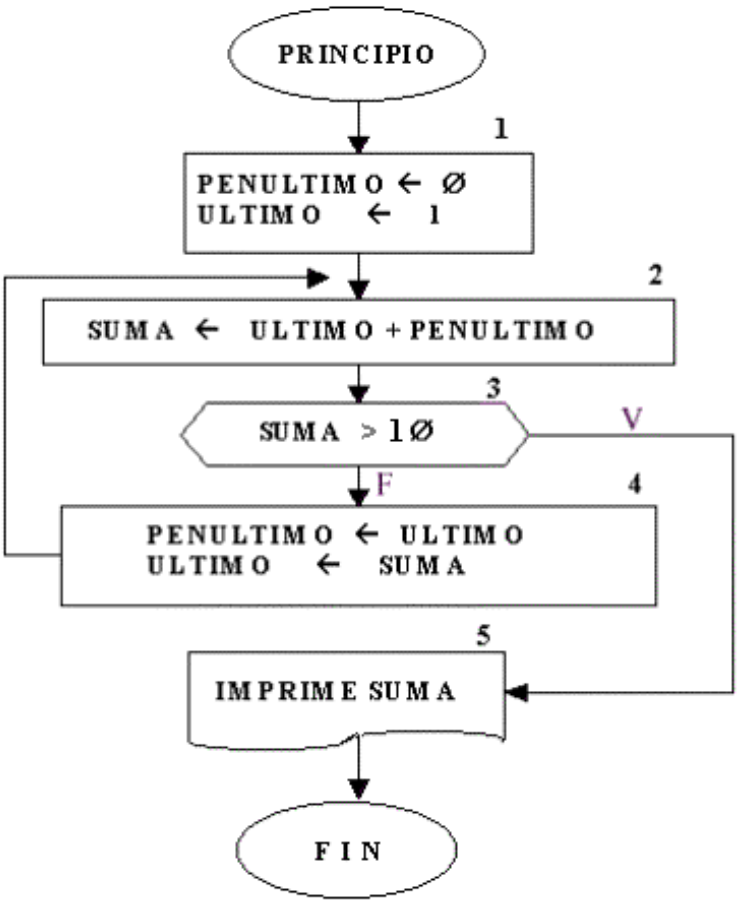


Figura 1-2. *Solución algorítmica del problema planteado de los números de fibbonacci.*

El analizador léxico leerá la línea caracter a caracter usando como referencia los espacios en blanco, que indican donde comienza y termina cada componente. En el caso que la línea sea escrita sin espacios en blanco, el analizador léxico está en capacidad de determinar que caracteres, diferentes al espacio en blanco, se comportan como separadores válidos para un lenguaje determinado.

Para el caso que nos atañe, el analizador léxico será invocado de manera sucesiva por el analizador sintáctico y, a cada llamada, en su orden, irá devolviendo lo siguiente:

* El identificador **suma**
* El símbolo de asignación **:=**
* El identificador **ultimo**
* El signo + de la operación de sumar

* El identificador **penultimo**

A medida que el analizador sintáctico llama al analizador léxico el primero va recibiendo los símbolos devueltos por este último. El analizador sintáctico intenta evaluar la estructura de la línea en cuestión para determinar si es válida desde un punto de vista gramatical, asociado al lenguaje siendo compilado. Lo anterior lo hace usando reglas (*gramática*) que permiten representar la estructura jerárquica, de la línea, a medida que se avanza en el análisis sintáctico. Normalmente esta jerarquia se representa en forma de árbol (*árbol de análisis sintáctico*) y que para el ejemplo que traemos se muestra en la figura 1-3.

Las reglas gramaticales usadas para la línea que estamos discutiendo podrían ser las siguientes:

<asignacion> → identificador := < expresion>
<expresion > → identificador
<expresion > → < expresion > + < expresion>

Por ahora bástenos con decir que cuando el analizador sintáctico llama al analizador léxico y recibe como respuesta un identificador, el primero buscará todas las reglas gramaticales que comiencen con **identificador**. Para nuestro caso la regla que cumple dicha condición es <asignacion>, respetando el orden en que aparecen las reglas. Ya que la regla <asignacion> se cumple en su parte inicial (**identificador**), el analizador sintáctico toma la decisión de avanzar en la evaluación utilizando dicha regla. Para ello vuelve a invocar el analizador léxico esperando que este le devuelva el signo de asignación **:=**. Si esto sucede, de la manera esperada, el analizador sintáctico nuevamente llama al analizador léxico y, con lo que le sea devuelto, intentará usar alguna de las reglas definidas para

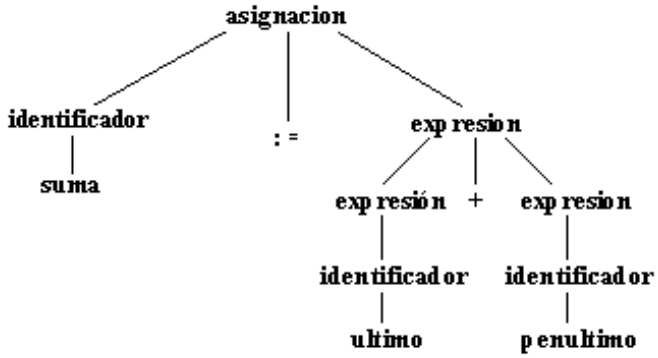


Figura 1-3. *Arbol de análisis sintáctico para la sentencia suma := ultimo + penultimo*

una <expresion> válida, ya que esta es la que continúa, en la evaluación, de la regla elegida. Si en algún momento determinado ninguna regla, o la parte de la que se está usando para la evaluación, concuerda con la entrada, entonces se habrá detectado un error sintáctico.

Ahora, es importante anotar el hecho de que, con la ayuda de las reglas gramaticales, se podría ir creando un árbol de arriba hacia abajo, y de izquierda a derecha, para graficar el recorrido hecho por el analizador sintáctico. A medida que el analizador léxico va devolviendo símbolos al analizador sintáctico, se podría ir dibujando cada nodo del árbol mostrado. Por esta razón el método se llama análisis sintáctico descendente. Este no es el único método utilizado de análisis, pero si uno de los más sencillos de implementar, y el que usaremos a lo largo de este texto.

El proceso descrito se ejecuta de manera repetida hasta evaluar completamente el programa fuente. Obviamente, para las demás líneas del programa fuente existirá las reglas gramaticales respectivas con las cuáles se determinan si las sentencias de dicho programa están correctamente escritas.

El análisis semántico se puede realizar de manera conjunta con el sintáctico, o esperar que este último termine. En este último caso se estaría haciendo una nueva pasada, o recorrido, por el programa fuente. En cualquier caso, el análisis semántico verifica que los tipos de datos involucrados en las operaciones, entre otras verificaciones, sean válidos.

Una vez ejecutadas las fases de análisis léxico, sintáctico y semántico, se puede concluir que el programa está correctamente escrito. Alcanzado este punto, la mayoría de los compiladores modernos tratan de mejorar la eficiencia, o el tamaño del código escrito. Para lograr esto se genera algún tipo de representación interna, o intermedia, que permita manipular más fácilmente el programa. Esto con el fin de determinar que sitios del programa son susceptibles de mejorar, bien sea reduciendo el número de instrucciones para llevar a cabo ciertas operaciones, o mejorando algunos flujos de control.

Finalmente, para conseguir que un algoritmo adopte una forma tal que la máquina lo pueda ejecutar, se requiere traducirlo a lenguaje de máquina que se coloca entonces en la memoria de la computadora. Estas instrucciones se colocarán en orden, en localidades con direcciones consecutivas de la memoria comenzando, por ejemplo, en la 0000. Después que la computadora ejecute una instrucción, la unidad de control obtendrá siempre la siguiente instrucción a partir de la dirección en que esté posicionado, exceptuando el caso que haya una instrucción de bifurcación, o ramificación, que indique una dirección distinta para aquella que sigue a la sentencia que este siendo ejecutada.

En el lenguaje de máquina las variables no pueden ser referidas por su nombre, sino únicamente por las direcciones de memoria asociadas con dichas variables. Supóngase que a las variables **penultimo, ultimo** y **suma** se les han dado respectivamente las localidades 0100, 0101 y 0102. Entonces, en un lenguaje simbólico, similar al lenguaje ensamblador (no usaremos lenguaje de máquina en este texto), la proposición **suma := ultimo + penultimo** tendrá la forma de la secuencia de tres instrucciones que se muestra en la figura 1-4. En donde, las letras a la izquierda de las instrucciones indican la operación que se va a ejecutar, y los números de cuatro dígitos del extremo derecho son las direcciones.

La mayoría de las operaciones que ejecuta el computador que estamos considerando las ejecuta a través de un registro especial llamado el acumulador. Las letras **LDA** denotan "Cargar al acumulador". La instrucción completa significa: "copiar el contenido de la dirección de memoria 0101 y llevarlo al acumulador". La segunda instrucción **ADD**, significa: "sumar el valor de la dirección 0100 al valor que se encuentra en el acumulador y colocar el resultado en éste". La tercera instrucción **STO**re, significa: "copiar o almacenar el número que se encuentra en el acumulador en la dirección de memoria 0102". Finalmente supongamos que se han reservado las direcciones 0103, 0104 y 0105 para las constantes 0, 1 y 10. El estado de la memoria al principio de la ejecución del programa para el algoritmo de Fibonacci, será el que muestra la figura 1-5.

Ya se han considerado las instrucciones que aparecen en las direcciones 0005 y 0006 de la memoria. Antes de estudiar las otras instrucciones, observemos las localidades 0100 a 0105 de la memoria para ver dónde se localizan las variables y las constantes.

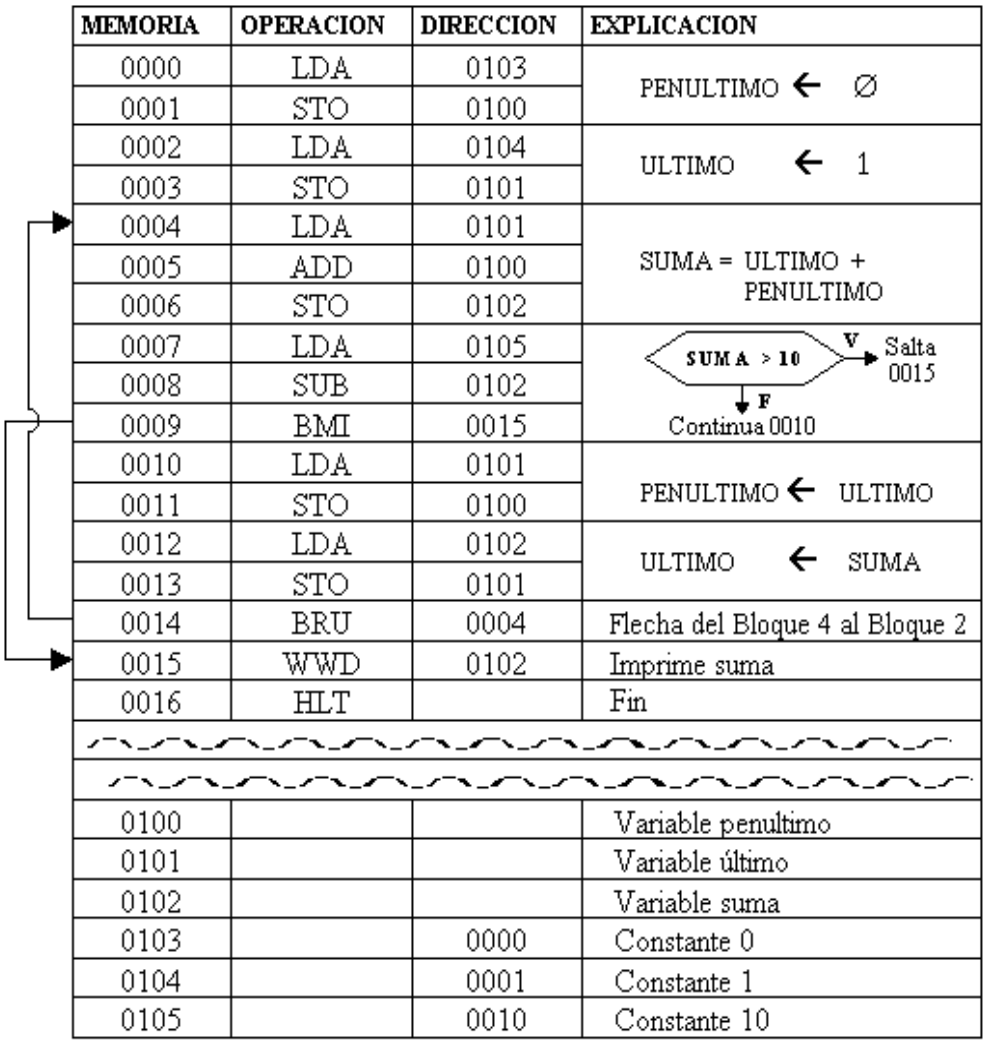
De las consideraciones previas vemos que cuando se ejecute la instrucción 0000, copiará el valor de la dirección 0103 (es decir, el número 0) y lo traerá al registro acumulador. A continuación la instrucción localizada en 0001 copiará (almacenará) el valor del acumulador en la dirección 0100. Juntos, estos pasos equivalen a asignar el valor 0 a la variable penúltimo. En forma

LDA		0101
ADD		0100
STO		0102

Figura 1-4. *Lenguaje simbólico para la sentencia suma := ultimo + penultimo*

similar las instrucciones de las direcciones 0002 y 0003 equivalen a asignar el valor 1 a la variable ultimo.

Recuérdese que la unidad de control ejecuta las instrucciones en orden, hasta que llega a una instrucción de ramificación. La primera de estas se localiza en la dirección 0009 y se lee **BMI** 0015. El código **BMI** significa "ramificar en menos". La instrucción completa significa: "si el número que se encuentra en el acumulador es negativo, proceder a la dirección 0015 para la siguiente instrucción; en caso contrario, proceder como de costumbre a la siguiente dirección numerada 0010". Al observar la instrucción de la dirección 0007, vemos que esta hace que traigamos al acumulador el contenido de 0105, es decir que coloquemos en el acumulador el número 10.



MEMORIA	OPERACION	DIRECCION	EXPLICACION
0000	LDA	0103	PENULTIMO ← Ø
0001	STO	0100	
0002	LDA	0104	ULTIMO ← 1
0003	STO	0101	
0004	LDA	0101	SUMA = ULTIMO + PENULTIMO
0005	ADD	0100	
0006	STO	0102	
0007	LDA	0105	SUMA > 10 V Salta 0015
0008	SUB	0102	F Continua 0010
0009	BMI	0015	
0010	LDA	0101	PENULTIMO ← ULTIMO
0011	STO	0100	
0012	LDA	0102	ULTIMO ← SUMA
0013	STO	0101	
0014	BRU	0004	Flecha del Bloque 4 al Bloque 2
0015	WWD	0102	Imprime suma
0016	HLT		Fin
0100			Variable penultimo
0101			Variable último
0102			Variable suma
0103		0000	Constante 0
0104		0001	Constante 1
0105		0010	Constante 10

Figura 1-5. *Lenguaje simbólico para el algoritmo de la figura 1-2.*

La siguiente instrucción que aparece en la dirección 0008, nos dice: “restar (**SUB**) el contenido de 0102 del acumulador y dejar el resultado en éste”. Como el contenido de 0102 es precisamente el valor de suma, esto equivale a colocar 10 – SUMA en el acumulador. De manera que este número será negativo sólo en el caso en el que sea SUMA > 10. En este caso la instrucción de ramificación en la dirección 0009 nos envía a la dirección 0015, en donde encontramos la instrucción **WWD** 0102 que significa: “escribir el contenido de la dirección 0102”. Esto equivale a imprimir el valor de SUMA.

Puede comprobar que las instrucciones de las direcciones 0010 a 0013 logran la asignaciones indicadas. Sólo hace falta explicar la instrucción **BRU** 0004 que aparece en la dirección de memoria 0014. **BRU** significa: “transferir incondicionalmente”. El significado completo de la instrucción es: “volver a la dirección 0004 de la memoria para siguiente instrucción y continuar en orden desde ahí”. Puede observarse que esto corresponde a la flecha que va del bloque 4 del diagrama de flujo hacia el bloque 2.

Por supuesto, la instrucción en la dirección 0016 representa un alto (**HLT**) y equivale a detener el proceso de computación. Todo lo anterior se comprenderá más claramente si sigue a mano el programa y mantiene un registro de los contenidos de las direcciones de memoria que corresponden a las variables utilizadas por el programa.

1. 4 Temas Relacionados

Secciones A, B, C y D del apéndice A.