UNIVERSIDAD NACIONAL DE CATAMARCA
FACULTAD DE TECNOLOGÍA Y
CIENCIAS APLICADAS

LICENCIATURA EN
SISTEMAS DE INFORMACIÓN

TRABAJO FINAL

Emulador de un Computador Básico para el Aprendizaje de Micro-operaciones

Autor:

Fernando Rubén Marchioli – M.U. N° 648

Directora:

Mgtr. Carola Victoria Flores

Catamarca, Marzo de 2017

AGRADECIMIENTOS

Me gustaría que estas líneas sirvieran para expresar mi más profundo y sincero agradecimiento a todas aquellas personas que con su ayuda han colaborado en la realización del presente trabajo, en especial a la Mgtr. Carola Victoria Flores, directora de esta investigación, por la orientación, el seguimiento y la supervisión continúa de la misma, pero sobre todo por la motivación y el apoyo recibido a lo largo de estos años.

Especial reconocimiento merece el interés mostrado por mi trabajo y las sugerencias recibidas de la profesora María Isabel Korzeniewski. También me gustaría agradecer la ayuda recibida del Lic. Mariano Argerich.

Un agradecimiento muy especial merece la comprensión, paciencia y el ánimo recibidos de mi familia y amigos.

A todos ellos, muchas gracias.

TABLA DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

RESUMEN

Una forma de involucrar a los estudiantes con los temas enseñados es llevarlos a la práctica y las universidades deben desarrollar vías de integración de las tecnologías de la información y la comunicación en los procesos de formación, por ello es importante contar con herramientas de software para ayudar a los alumnos a que su trabajo sea más entretenido y provechoso. En este contexto se emnarca el presente trabajo de tesis donde se aborda el tema de aprendizaje de micro-operaciones para un computador básico.

La construcción de una aplicación que emule un computador básico, implica la implementacion de una maquina virtual que puede ser usada por alumnos y profesores de las areas de conocimiento referentes a sistemas operativos y arquitectura de computadores. El Presente trabajo final provee una herramienta educativa para el aprendizaje de micro-operaciones, mediante la implementación de un emulador de computador básico.

El objetivo principal es proveer una aplicación software del tipo herramienta educativa, debido a que los alumnos de la Facultad de Tecnología y Ciencias Aplicadas no cuentan con una computadora con las caracteristicas necesarias para el aprendizaje de micro-operaciones, en la cual puedan ejecutar y depurar sus programas pudiendo ver los resultados de la ejecución de una instrucción, es decir los ciclos de ejecución, estado de registros, memoria y banderas del procesador.

Para llevar adelante el desarrollo del software se utilizó los principios de la Ingeniería de software y la metodología estructurada.

INTRODUCCIÓN

Hoy por hoy la educación ocupa un renglón prioritario en el desarrollo de los pueblos donde se liga íntimamente la evolución tecnológica, que a su vez representa un auxiliar invaluable en la acción docente durante el proceso enseñanza-aprendizaje.

En el ámbito educativo con el apoyo de la Tecnología de la Información y Comunicación (TIC) es posible que, tanto estudiantes como docentes, participen activamente en la construcción del contenido necesario para las actividades formativas, que además pueden compartir con otros miembros de la comunidad educativa. Incluso están en la capacidad de diseñar y producir sus propios contenidos apoyados en las TIC.

Todo esto y mucho más, es perfectamente posible en la actualidad, teniendo acceso a las TIC y haciendo uso efectivo de todas sus potencialidades, aplicándolas pertinentemente para el mejoramiento productivo y cualitativo de los procesos educativos.

Estas herramientas permiten a la educación enfocarse hacia el dominio del *saber hacer*. Son evidentes las potencialidades que ofrecen las actividades vinculadas con la utilización de herramientas TICs en el proceso de enseñanza-aprendizaje. El presente trabajo profundiza sobre este tema para promover el uso de herramientas educativas y todas las aplicaciones relacionadas a temas de estructura de computadores, que seguramente permitirán al docente mejorar su praxis educativa. Este trabajo propone una herramienta de enseñanza-aprendizaje de micro-operaciones, que emule el computador básico y permita el desarrollo, ejecución, y depuración de programas escritos en el lenguaje ensamblador y código máquina. La herramienta permite que los alumnos tengan la posibilidad de ejecutar sus programas, depurarlos, observar las micro-operaciones ejecutadas e inspeccionar los valores de los registros del procesador, interactuando con un computador.

El trabajo final involucra tanto aspectos teóricos como prácticos. Se fundamenta en el plan de estudios de la carrera Ingeniería en Informática.

Con esta herramienta se espera reducir los tiempos necesarios para adquirir los conocimientos por parte de los alumnos. Facilitar el proceso enseñanza-aprendizaje a través de una herramienta de software. Autocorrección, por parte del compilador e intérprete, de los errores de programación, complementando la participación del docente.

El trabajo de tesis se compone de cuatro capítulos donde se describe la labor realizada y los resultados obtenidos.

- Capítulo I: contiene los conceptos del tema tratado en la investigación y conceptos relacionados, para interpretar de manera correcta el desarrollo de la investigación y poder satisfacer los objetivos planteados.
- Capítulo II: presenta el Marco Metodológico compuesto por una breve introducción, el problema de investigación, los antecedentes, la justificación y los objetivos. Se desarrolla el diseño metodológico abordando los métodos y procedimientos que fueron utilizados para llevar a cabo el presente trabajo.

- Capítulo III: se presenta el proceso de desarrollo de software llevado a cabo, se muestra la especificación de requerimientos las funcionalidades y restricciones del sistema. Se especifican los modelos obtenidos del análisis del sistema los cuales son el modelo ambiental y el modelo de comportamiento. Los artefactos de la actividad de diseño expuestos son los diagramas de estructuras, las interfaces gráficas de la herramienta y el diseño de archivos.
- Capítulo IV: presenta las Conclusiones.
- El trabajo finaliza con la exposición de los anexos que complementan el trabajo, las referencias y la bibliografía general.

Capítulo I

Marco Teórico

1.1 INTRODUCCIÓN

Este Capítulo aborda los conceptos teóricos y presenta el marco teórico en el que se basa el trabajo, el cual surgió del análisis bibliográfico y del contexto de la investigación, ya que ningún hecho o fenómeno de la realidad puede abordarse sin una adecuada conceptualización, sino que siempre parte de algunas ideas o informaciones previas, de algunos referentes teóricos y conceptuales.

1.2 ORGANIZACIÓN Y FUNCIONAMIENTO DE UN COMPUTADOR BÁSICO

A continuación se describe el computador básico especificado por Morris Mano (1993) que se considera para realizar el emulador que se desarrolló y se describe en el trabajo.

1.2.1 Sistemas Digitales

La organización interna de un sistema digital se define por una secuencia de micro-operaciones que se realizan en los datos almacenados en sus registros. Un computador digital es un sistema digital de propósito general. Un computador digital de propósito general es capaz de ejecutar diversas micro-operaciones y, además, puede ser instruido sobre la secuencia de operaciones específicas que debe realizar.

El usuario de un sistema de estos puede controlar el proceso por medio de un programa, esto es, un conjunto de instrucciones que especifican las operaciones, operandos y la secuencia mediante la cual tiene que ocurrir el procesamiento. La tarea de procesamiento de datos puede alterarse simplemente especificando un nuevo programa con instrucciones diferentes o especificando las mismas instrucciones con datos diferentes.

Una instrucción de un computador es un código binario que especifica una secuencia de micro-operaciones para el computador. Los códigos de instrucción junto con los datos se almacenan en la memoria. El control lee cada instrucción de la memoria y la coloca en un registro de control. El control entonces interpreta el código binario de la instrucción y procede a ejecutar la instrucción emitiendo una secuencia de funciones de control. Cada computador de propósito general tiene su repertorio propio único de instrucciones. La habilidad para almacenar y ejecutar instrucciones, el concepto de programa almacenado, es la propiedad más importante de un computador de propósito general.

1.2.2 Operaciones del computador

Un código de instrucción es un grupo de bits que le dice al computador que realice una operación específica. Usualmente se divide en partes, cada una tiene su interpretación particular. La parte básica de un código de instrucción es su código de operación. El código de operación de una instrucción es un grupo de bits que define operaciones tales como suma, resta, multiplicación, desplazamiento y complemento. El conjunto de operaciones formuladas por un computador depende del procesamiento que se intente llevar a cabo. El número total de operaciones así obtenido determina el conjunto de las operaciones de la máquina. El número de bits requerido por parte de una operación de instrucción es una función del número total de operaciones utilizadas. Consta de por lo menos n bits para 2^n operaciones diferentes dadas (o menores).

Una operación es parte de una instrucción almacenada en la memoria de computador. Es un código binario que le dice al computador que realice una operación específica. La unidad de control recibe la instrucción de la memoria e interpreta los bits del código de operación. Entonces emite un mensaje de las funciones de control que realiza micro-operaciones en los registros internos del computador. Para cada código de operación, el control emite una secuencia de micro-operaciones para la implementación en hardware de la operación especial. Por esta razón, un código de operación algunas veces se denomina una macro-operación debido a que especifica un conjunto de micro-operaciones.

La parte de operación de un código de instrucción especifica la operación que se debe realizar. Esta operación debe ser ejecutada en algunos datos almacenados en la memoria y/o registros del procesador. Un código de instrucción, por consiguiente, debe especificar no solamente la operación, sino también los registros y/o palabras de memoria en donde los operandos se deben encontrar, como también los registros o palabras de memoria en donde el resultado sea almacenado. Las palabras de memoria pueden especificarse en los códigos de instrucción por su dirección. Los registros de procesamiento pueden especificarse asignando a la instrucción otro código binario de k bits que especifica uno de 2^k registros. Hay muchas variantes para arreglar el código binario de instrucción y cada computador tiene su propio formato particular de código de instrucción.

1.2.3 Organización del computador

La manera más simple de organizar un computador es tener un registro procesador y un formato de código de instrucción con dos partes. La primera especifica la operación que se debe realizar y la segunda especifica una dirección. La dirección le dice al control en dónde encontrar un operando en la memoria. Este operando se lee de la memoria y utiliza los datos en que debe operar junto con los datos almacenados en el registro procesador.

La Figura 1-1 muestra este tipo de organización. Las instrucciones son almacenadas en una sección de la memoria y los datos en otra. Para una unidad de memoria con 4096 palabras necesitamos 12 bits para especificar una dirección puesto que $2^{12} = 4096$. Si almacenamos cada código de instrucción en una palabra de memoria de 16 bits, tenemos disponibles cuatro bits para especificar una de las 16 operaciones posibles y 12 bits para especificar la dirección de un operando. El control lee una instrucción de 16 bits de una porción del programa de memoria. Utiliza la parte de dirección de 12 bits de la instrucción para leer un operando de la porción de datos de la memoria. Entonces él ejecuta la operación por medio de micro-operaciones entre el operando y el registro procesador. Los computadores que tienen un solo registro procesador usualmente le asignan el nombre de acumulador y lo rotulan como AC.

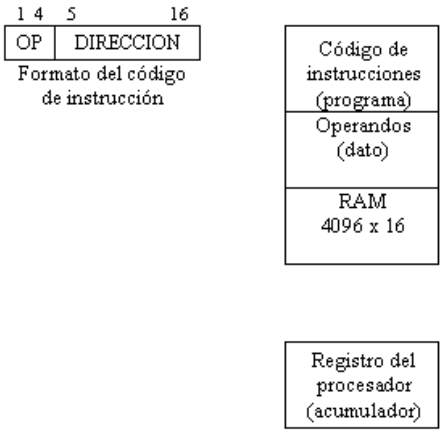

Figura 1-1 Organización del programa almacenado.

Si una operación en un código de instrucción no necesita un operando de la memoria, el resto de los bits en la instrucción pueden utilizarse para otros fines. Por ejemplo, operaciones tales como aclarar AC, complementar AC, e incrementar AC operan en los datos almacenados en el registro AC. Ellas no necesitan un operando de la memoria. Para este tipo de operaciones, la segunda parte del código de instrucción bits 5 a 16 no se necesitan para especificar una dirección de memoria y pueden utilizarse para especificar otras operaciones para el computador.

Algunas veces es conveniente utilizar los bits de dirección de un código de instrucción no como una dirección sino como un operando actual. Cuando la segunda parte de un código de instrucción especifica un operando, la instrucción se dice que tiene un operando inmediato. Cuando la segunda parte especifica la dirección de un operando, la instrucción se dice que tiene una dirección directa. Esto contrasta con una tercera posibilidad denominada dirección indirecta, en donde los bits en la segunda parte de la instrucción designan una dirección de la palabra de memoria en la que se encuentra la dirección del operando. Es costumbre utilizar un bit en el código de instrucción para distinguir entre una dirección directa y una indirecta.

Como ilustración de este concepto, considere el formato del código de instrucción que se muestra en la Figura 1-2(a). Consta de un código de operación de tres bits designado por OP, una dirección de seis bits designado por AD, y un bit de modo de dirección indirecta designado por I. El bit de modo es 0 para una dirección directa y 1 para una dirección indirecta. Una instrucción de dirección directa se muestra en la Figura 1-2(b). Es colocada en la dirección 2 en la memoria. El bit I es 0, de tal manera que la instrucción se reconoce (por el control) como una instrucción de dirección directa. Puesto que la parte de la dirección AD es igual al equivalente binario de 9 (001001), el control encuentra el operando en la memoria en la dirección 9. La instrucción está en la dirección 2 mostrada en la Figura 1-2(c) y tiene un bit de modo I = 1. Por consiguiente, es reconocida como una instrucción de dirección indirecta. La parte de la dirección es el equivalente binario de 9. El control va a la dirección 9 para encontrar la dirección del operando. Esta dirección está en la porción de la dirección de la palabra y es designada por M(AD). Puesto que M(AD) contiene 14 (el binario 001110), el control encuentra el operando en la memoria en la dirección 14. La instrucción

de dirección indirecta necesita dos referencias en memoria para fechar un operando. La primera referencia se necesita para leer la dirección del operando; la segunda para el operando mismo.

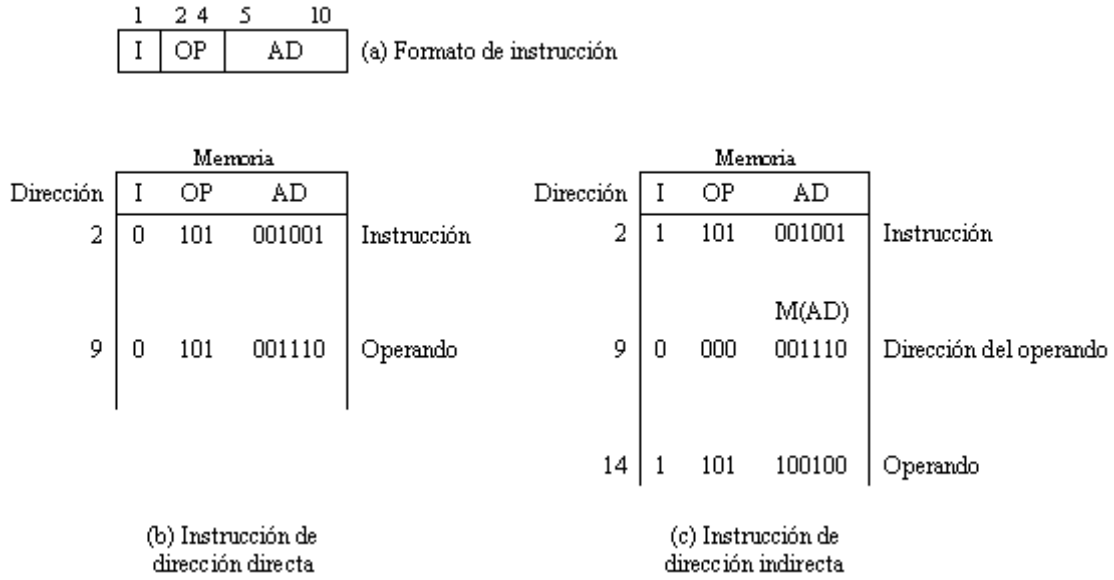

Figura 1-2 Demostración de las instrucciones de dirección directa e indirecta.

1.2.4 Configuración de registros

El control lee una instrucción de una dirección específica en la memoria y la ejecuta. Continúa entonces leyendo la siguiente instrucción en secuencia y la ejecuta, y así sucesivamente. Este tipo de secuenciación de instrucción necesita un contador para calcular la dirección de la instrucción siguiente después de que se completa la ejecución de la instrucción corriente. Además, las palabras de la memoria no pueden comunicarse con registros procesadores directamente sin ir a través de un registro separador y de dirección. Es también necesario proporcionar un registro en la unidad de control para almacenar los códigos de operación después de que ellos han sido leídos de la memoria. Este requisito hace que la configuración de registro sea como la mostrada en la Figura 1-3. Esta configuración de registro será utilizada para describir la organización interna de un computador digital básico.

La unidad de memoria tiene una capacidad de 4096 palabras y cada palabra contiene 16 bits. Doce bits de una palabra de instrucción se necesitan para especificar la dirección de un operando. Esto deja cuatro bits para la parte de operación de la instrucción. Sin embargo, solamente tres bits se utilizan para especificar un código de operación. El cuarto bit se utiliza para especificar un modo de dirección directo o indirecto. El registro separador de memoria (MBR) consta de 16 bits, lo mismo que el registro AC (acumulador). El flip-flop E es una extensión de AC. El es utilizado durante las operaciones de desplazamiento, y recibe el acarreo final durante la suma, y de otra manera es un flip-flop que se utiliza para simplificar las capacidades de procesamiento de datos del computador. El registro I tiene una sola

celda para almacenar el bit de modo y el registro de operación (OPR) almacena el código de operación de tres bits leídos en la memoria.

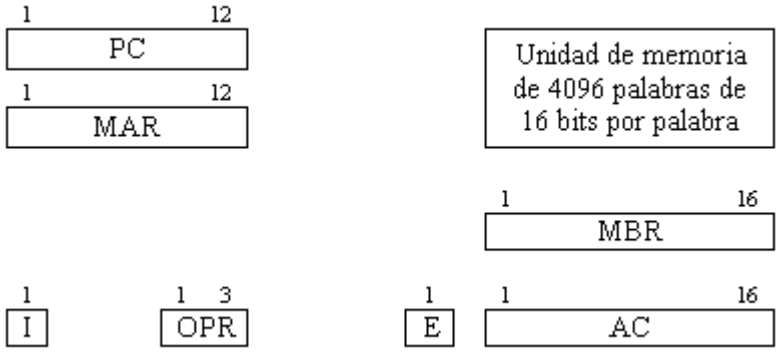

Figura 1-3 Registros básicos del computador.

El registro de dirección de memoria MAR tiene 12 bits puesto que esta es la longitud de la dirección de memoria. El contador de programa (PC) también tiene 12 bits y retiene la dirección de la instrucción siguiente que debe leerse de la memoria después de que la instrucción corriente se está ejecutando. Este registro va a través de una secuencia de cuenta y hace que el computador lea las instrucciones secuencialmente y que han sido almacenadas previamente en la memoria. Las palabras de instrucción son leídas y ejecutadas en secuencia a no ser que se encuentre una instrucción de ramificación. Una instrucción de ramificación tiene una parte de operación que exige una transferencia a una instrucción no consecutiva en la memoria. La parte de dirección de una instrucción de ramificación es transferida a PC para que sea la dirección de la instrucción siguiente. Para leer una instrucción, el contador de PC es transferido a MAR, y se inicia un ciclo de lectura de memoria, y PC se incrementa a uno. Esto coloca el código de instrucción en MBR y prepara PC para la dirección de la instrucción siguiente. El código de operación es transferido a OPR, el bit de modo a I y la parte de la dirección en MAR. Una operación de lectura de memoria coloca el operando (si I = 0) en MBR. El AC y el MBR son utilizados como registros fuente para las micro-operaciones especificadas por el código de operación. El resultado de la operación se almacena en AC.

Sin embargo, una instrucción puede tener un bit Indirecto I igual a 1, o puede que no requiera una operando de la memoria, o puede ser una instrucción de ramificación. En cada uno de estos casos, el control debe emitir un conjunto diferente de funciones de control para ejecutar tipos diferentes de transferencia de registros.

1.2.5 Formatos de Instrucción

El computador básico tiene tres formatos de códigos de instrucción diferente, como se muestra en la Figura 1-4. La parte de operación de la instrucción contiene tres bits; el significado de los trece bits restantes depende del código de operación encontrado. Una instrucción de referencia de memoria utiliza los 12 bits últimos para especificar una dirección y el primer bit para especificar el modo I. Una instrucción de referencia de registro especifica

una operación o una prueba de registro AC o E. Un operando de la memoria no se necesita; por consiguiente, los últimos 12 bits son utilizados para especificar la operación o probar lo ejecutado. Una instrucción de referencia de registro se reconoce por el código de operación 111 con un 0 en el primer bit de la instrucción. Similarmente, una instrucción entrada-salida, no necesita una referencia de la memoria y se reconoce por el código de operación 111 con un 1 en el primer bit de la instrucción. Los 12 bits restantes se utilizan para especificar el tipo de operación entrada-salida o la prueba realizada. Note que el primer bit del código de instrucción no se utiliza como un bit de modo cuando los últimos 12 bits no son utilizados para designar una dirección.

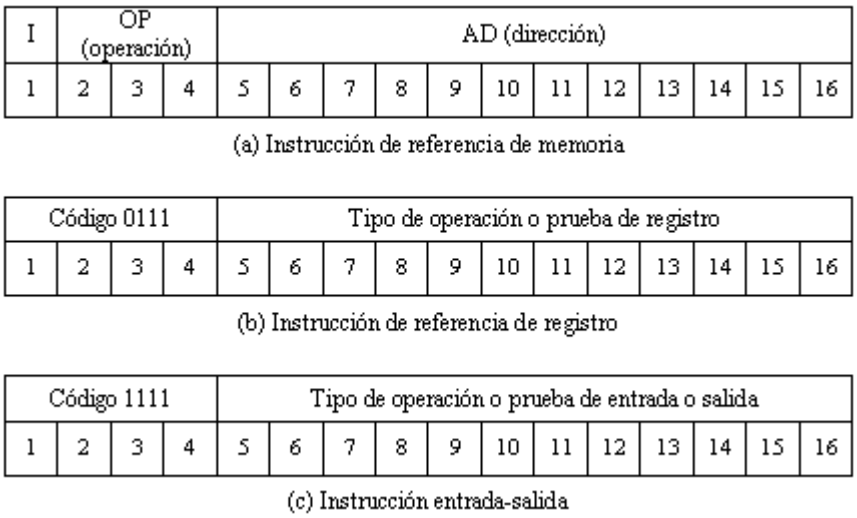

Figura 1-4 Formato de instrucción para el computador básico.

Solamente tres bits de la instrucción son utilizados para código de la operación. Parece que el computador estuviera restringido a un máximo de ocho operaciones distintas. Sin embargo, puesto que las instrucciones de referencia de registro y entrada-salida utilizan los 12 bits restantes como parte del código de operación, el número total de instrucciones puede exceder ocho. En realidad, el número total de instrucciones elegidas para el computador básico es igual a 25.

Las instrucciones para el computador se enumeran en la Tabla 1-1. El símbolo designa una palabra de tres letras y representa una abreviatura que se intenta para los programadores y usuarios. El código hexadecimal es igual al número hexadecimal equivalente del código binario utilizado para la instrucción. Utilizando el equivalente hexadecimal reducimos los 16 bits de un código de instrucción a cuatro dígitos con cada dígito hexadecimal siendo el equivalente a cuatro bits. Una instrucción de referencia de memoria tiene una parte de dirección de 12 bits. La parte de dirección se denota por el símbolo AD y debe ser especificada por tres dígitos hexadecimales. El primer bit de la instrucción se designa con el símbolo I. Cuando I = 0, AD es la dirección del operando. En este caso, los primeros cuatro bits de una instrucción tienen una designación hexadecimal de 0 a 6 puesto que el primer bit es 0. Cuando I = 1, AD es una dirección en donde se encuentra la dirección del operando en

la memoria. El dígito hexadecimal equivalente de los primeros cuatro bits de la instrucción está en el rango de 8 a E puesto que el primer bit es 1.

Las instrucciones de referencia de registro utilizan 16 bits para especificar una operación. Los primeros cuatro bits son siempre 0111, que es equivalente al hexadecimal 7. Los otros tres dígitos hexadecimales dan el binario equivalente a los 12 bits restantes. Las instrucciones de entrada-salida también utilizan 16 bits para especificar una operación. Los primeros cuatro bits son siempre 1111 que es el equivalente al hexadecimal F. Las tres X que siguen a F para una instrucción entrada-salida son dígitos que distinguen entre instrucciones diferentes I/O. Estos dígitos se especifican posteriormente en la Tabla1-5.

Símbolo	Código hexadecimal			Descripción
	I = 0	I = 1	Dirección	
AND	0	8	AD	AND la palabra de memoria a AC
ADD	1	9	AD	Suma de la palabra de memoria a AC
LDA	2	A	AD	Carga AC a partir de la memoria
STA	3	B	AD	Almacena AC en la memoria
BUN	4	C	AD	Ramifica incondicionalmente
BSA	5	D	AD	Ramifica y mantiene la dirección de retorno
ISZ	6	E	AD	Incrementa y salta si es cero
CLA	7800			Aclara AC
CLE	7400			Aclara E
CMA	7200			Complementa AC
CME	7100			Complementa E
CIR	7080			Circula a la derecha E y AC
CIL	7040			Circula a la izquierda E y AC
INC	7020			Incrementa AC
SPA	7010			Salta si AC es positivo
SNA	7008			Salta si AC es negativo
SZA	7004			Salta si AC es cero
SZE	7002			Salta si E es cero
HLT	7001			Para el computador
I/O	FXXX			Instrucciones Entrada-salida

Tabla 1-1 Instrucciones del computador.

1.2.6 Unidad de Control

El computador digital opera en pasos discretos. Las micro-operaciones son realizadas durante cada paso. Las instrucciones son leídas de la memoria y ejecutadas en los registros por una secuencia de micro-operaciones. Una vez que se activa un interruptor de arrancar, la secuencia del computador sigue un patrón básico. Una instrucción cuya dirección está en el registro PC es leída de la memoria en el MBR. Su parte de operación es transferida a OPR y el bit de modo en el registro I. La parte de operación es decodificada en la unidad de control. Si es del tipo de referencia de memoria aquella necesita un operando de la memoria, y el control verifica el bit en I. Si I = 0, la memoria se accesa de nuevo para leer los bits del operando. Si I = 1, la memoria se accesa para leer la dirección del operando y de nuevo para leer el operando. Así una palabra leída de la memoria en el MBR puede ser una instrucción, un operando, o una dirección de un operando. Cuando una instrucción se lee de la memoria, el computador se dice que está en el ciclo de instrucción fetch. Cuando la palabra leída de la memoria es la dirección de un operando el computador está en un ciclo

indirecto. Cuando la palabra leída de la memoria es un operando, el computador está en un ciclo de ejecución de datos. Es la función del control mantener la pista de los diversos ciclos.

La unidad de control utiliza dos flip-flops para distinguir entre los tres ciclos. Estos flip-flops se denotan por las letras F y R. Un decodificador 2 por 4 asociado con estos flip-flops provee cuatro salidas, tres de las cuales pueden utilizarse para diferenciar entre los ciclos arriba mencionados. El computador tiene un cuarto ciclo de interrupción que se presentará posteriormente. La Tabla1-2 enumera los valores binarios de F y R, y la variable de decodificación c_i que es igual a 1 para cada uno de los cuatro ciclos.

Flip-flops		Salida del decodificador	Ciclo del computador
F	R		
0	0	c_0	Ciclo fetch (lectura de instrucción)
0	1	c_1	Ciclo indirecto (lectura de la dirección del operando)
1	0	c_2	Ciclo de ejecución (lectura del operando)
1	1	c_3	Ciclo de interrupción (ver Capítulo 5)

Tabla 1-2 Control del ciclo del computador.

El diagrama de bloques para la unidad de control del computador básico es el de la Figura 1-5.

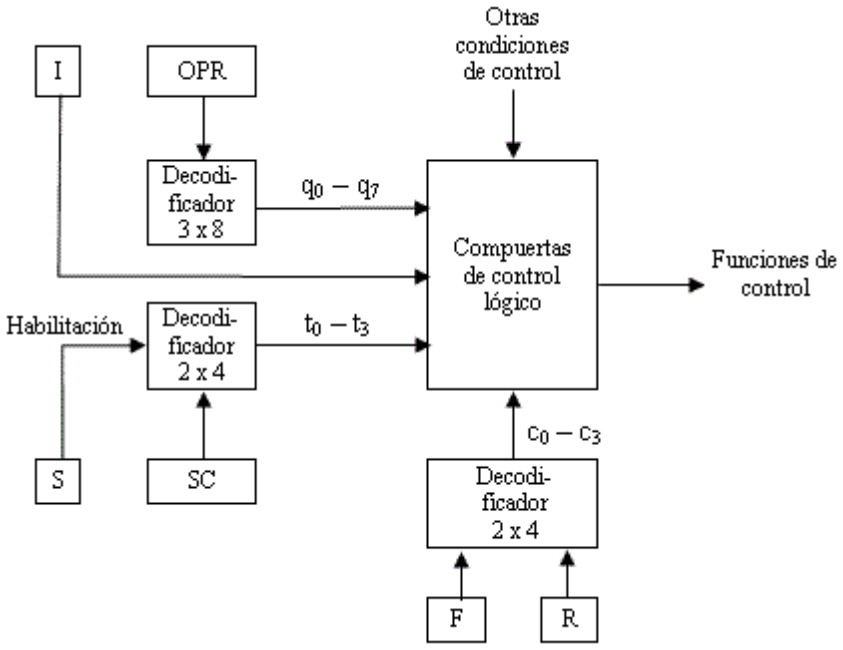

Figura 1-5 Diagrama de bloques de la unidad de control.

La secuencia de tiempo en el computador es generada por un contador de secuencia de 2 bits (SC) y por un decodificador de 2 por 4. Las señales de tiempo que salen del

decodificador se designan por t_0, t_1, t_2, y t_3. Suponemos que el ciclo de memoria es más corto que el intervalo de tiempo entre los pulsos de reloj. De acuerdo a esta suposición, un ciclo de lectura o escritura iniciado por el flanco descendente de una variable de tiempo será completado en el momento en que llegue el siguiente pulso de reloj.

La parte de código de operación de la instrucción en OPR es decodificada en ocho salidas q_0 a q_7, el número subíndice es igual al equivalente binario del código de operación. Los flip-flops de control de ciclo F y R son decodificados en las cuatro salidas c_0 a c_3 como se especifica en la Tabla1-2. Las compuertas lógicas de control generan las diversas funciones de control para las micro-operaciones en el computador. Cada función de control incluye con ella una variable de tiempo t_i y una designación de ciclo c_i. Durante el ciclo de ejecución, la función de control incluye también una variable q_i. La variable I y otras condiciones de control también se necesitan para la generación de funciones de control.

El diagrama de bloques del control muestra un flip-flop S de arranque-parada conectado a la entrada habilitadora del decodificador de tiempo. Todas las micro-operaciones están condicionadas a las señales de tiempo. Las señales de tiempo son generadas únicamente cuando S = 1. Cuando S = 0, ninguna de las variables t_0 a t_3 son iguales a 1 y por consiguiente la secuencia del control para y el computador se detiene. El flip-flop S puede estar en set o ser aclarado por medio de interruptores que están en la consola del computador, o aclarados por la instrucción HLT (parada). Es necesario asegurar que cuando S esté es su condición de set por el interruptor de arranque, la señal t_0 es la primera que ocurre; y cuando S es aclarado por el interruptor de parada la instrucción corriente es completada antes de que el computador se detenga.

Estamos ahora listos para especificar en una notación simbólica la secuencia de funciones de control y de micro-operaciones para el computador. Cada enunciado simbólico consta de una función de control seguida por dos puntos, seguidos por una o más micro-operaciones.

1.2.6.1 El Ciclo Fetch

Una instrucción es leída de la memoria durante el ciclo de instrucción fetch. Las relaciones de transferencia de registro que especifican este proceso son:

c_0t_0: MAR<-- PC Transferencia de la dirección de la instrucción

c_0t_1: MBR <-- M, PC <-- PC + 1 Lea la instrucción e incremente PC

c_0t_2: OPR <--MBR(OP), I <-- MBR(I) Transfiera el código OP y el bit de modo

$q'_7Ic_0t_3$: R <-- 1 Vaya al ciclo indirecto

$(q_7 + I')c_0t_3$: F <-- 1 Vaya al ciclo de ejecución

El ciclo fetch se lo conoce por la variable c_0. Las cuatro señales de tiempo que ocurren durante este ciclo inician la secuencia de micro-operaciones para el ciclo fetch. Las direcciones, que están en PC, son transferidas a MAR. La memoria lee la instrucción y la coloca en MBR. Al mismo tiempo, el contador del programa se incrementa en uno para prepararlo para la dirección de la instrucción siguiente. La parte de operación y el bit de

modo son transferidos de MBR a OPR e I, respectivamente. Note que la parte de dirección de la instrucción permanece en MBR.

En el instante t_3 el control toma una decisión referente al cuál debe ser el ciclo siguiente del computador. Si el código de operación contiene 111, la instrucción es o una referencia de registro o una instrucción entrada-salida. Por consiguiente, si $q_7 = 1$, el flip-flop F se lleva a set y el control va al ciclo de ejecución. Si $q_7 = 0$, la instrucción es una instrucción de referencia de memoria. Ahora, si I = 0, esto significa que la instrucción es una instrucción directa de tal manera que el control va al ciclo de ejecución colocando F en 1. Si I = 1, es una instrucción indirecta. El control va al ciclo indirecto colocando R en 1. Note que la condición para ir al ciclo de ejecución es el complemento de la condición para ir al ciclo indirecto, esto es, $(q'_7 I)' = q_7 + I'$.

Recuerde que F y R son 0 durante el ciclo fetch. Colocando R en 1 se obtiene FR = 01. Colocando F en 1 se obtiene FR = 10. La variable de tiempo que se vuelve 1 después de t_3 es t_0. Cuando el siguiente ciclo t_0 se vuelve 1 el computador puede estar en el ciclo de ejecución o en el ciclo indirecto.

1.2.6.2 El Ciclo Indirecto

El ciclo indirecto se reconoce por la variable c_1. Durante este ciclo, el control lee la palabra de la memoria en donde se encuentra la dirección del operando. Las micro-operaciones de transferencia de registros para el ciclo indirecto son:

c_1t_0: MAR<-- MBR(AD) Transferir la parte de dirección de la instrucción

c_1t_1: MBR <-- M Leer la dirección del operando

c_1t_2: Nada

c_1t_3: F <-- 1, R <-- 0 Vaya al ciclo de ejecución

La parte de dirección de la instrucción está en MBR (5-16). Estos 12 bits son simbolizados por MBR(AD). Son transferidos a MAR y se inicia un ciclo de memoria. Los bits 15-16 de la palabra justamente leída de la memoria contienen la dirección del operando. Ahora que la dirección del operando está en MBR(AD), el control va al ciclo de ejecución haciendo el set de F y aclarando R. Note que nada se hace durante el tiempo t_2. El cambio de F y de R se debe evitar durante este tiempo porque la variable de tiempo t_3 que sigue a t_2 encontrará el computador en un ciclo diferente. Los cambios de un ciclo a otro deben hacerse en el instante t_3 de tal manera que el ciclo siguiente pueda comenzar con la variable t_0.

1.2.6.3 Diagrama de Flujo del Control

El control alcanza el ciclo de ejecución por dos caminos diferentes, después del ciclo de fetch o después del ciclo indirecto. El diagrama de flujo de la Figura 1-6 ilustra las diferentes rutas disponibles en la unidad de control. El, resume la discusión hasta este punto e indica las rutas que las tareas de control toman durante el ciclo de ejecución. Un diagrama de flujo es un diagrama de bloques conectados por líneas directas. Las líneas directas entre los

bloques designan las rutas que deben tomar de un paso al otro. Los dos tipos principales de bloques son (1) bloques de función que muestran las operaciones que deben realizarse (en cajas rectangulares), y (2) bloques de decisión que tienen dos o más rutas alternas que dependen del estatus de la condición indicada dentro de una caja de forma de diamante.

Como se muestra en el diagrama de flujo, el interruptor de arranque aclara los flip-flops de control de ciclo. Esto coloca el computador en el ciclo de fetch. Una instrucción es leída de la memoria y su operando y bits de modo colocados en los registros de control. Si $q_7 = 1$, o si $q_7 = 0$ e I = 0, el control va al ciclo de ejecución colocando F en 1. Si $q_7 = 0$ e I = 1; va al ciclo indirecto colocando R en 1. Durante el ciclo indirecto, el control lee la dirección del operando y se mueve al ciclo de ejecución.

En el comienzo del ciclo de ejecución, el código de operación de la instrucción está en OPR, el primer bit de la instrucción está en I, y MBR(5-16) retiene el resto de la instrucción. Si $q_7 = 0$, MBR(5-16) contiene la dirección efectiva. Esta es la dirección actual del operando y puede haber venido de la parte de dirección de la instrucción (cuando I = 0) o del ciclo indirecto (cuando I = 1). En cualquier caso, esta parte de la dirección en MBR se designa por MBR(AD). El control lee el operando encontrado en la dirección efectiva y ejecuta la instrucción de referencia de memoria. Si $q_7 = 1$, los bits en MBR(5-16) son parte del código de operación. El control verifica el bit en I para determinar si la instrucción es una referencia de registro o del tipo entrada-salida. El entonces verifica los bits de MBR(5-16) para decidir cuál instrucción específica se debe ejecutar. El flip-flop F de control de ciclo es aclarado después de la ejecución de la instrucción. Esto introduce un retorno al ciclo de fetch para comenzar de nuevo a leer y ejecutar la siguiente instrucción.

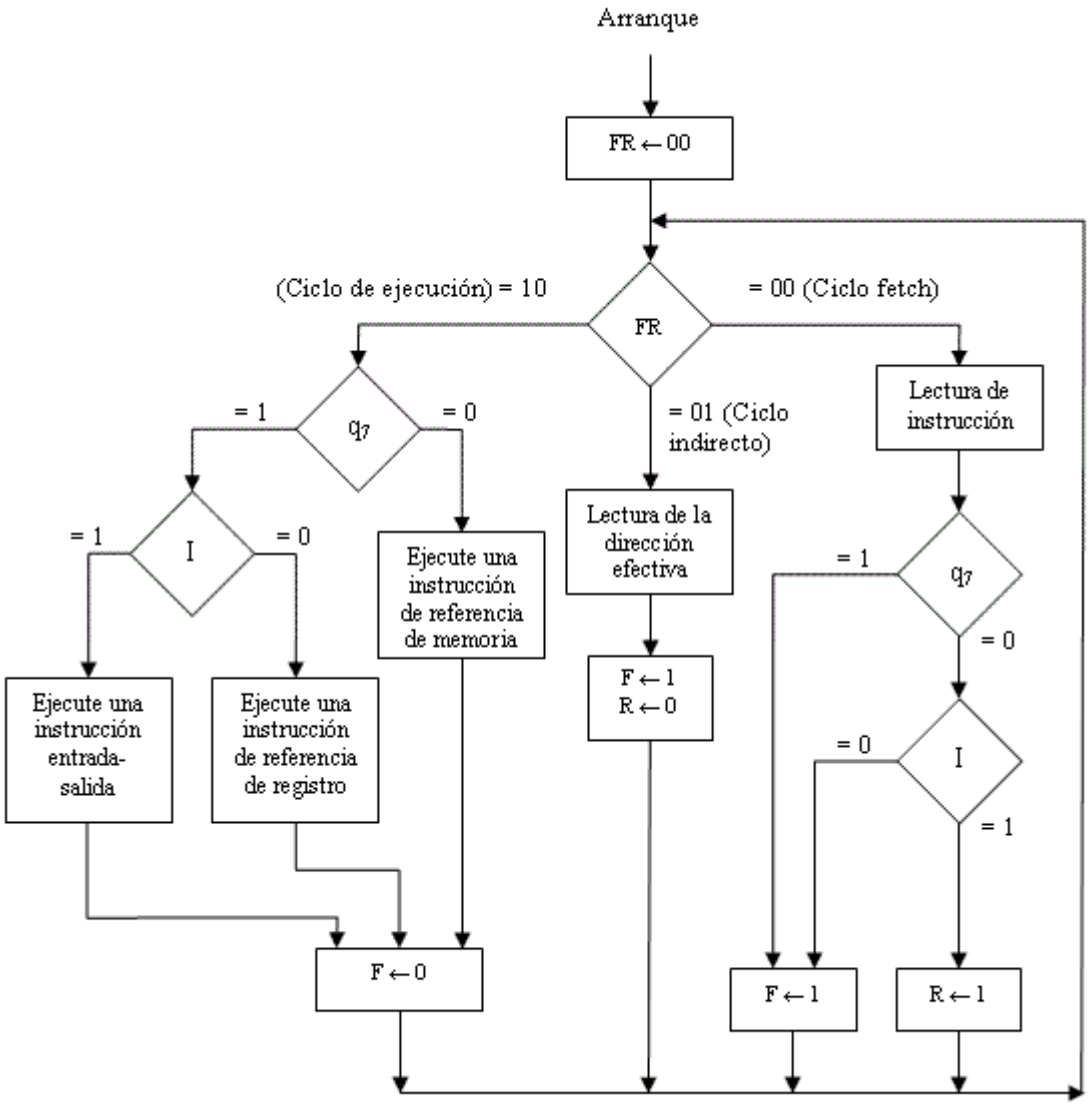

Figura 1-6 Diagrama de flujo para el control del ciclo del computador.

1.2.7 Notación Simbólica para Instrucciones

Para especificar las micro-operaciones que se necesitan para la ejecución de cada instrucción, es necesario que la función que ellas intentan realizar sea definida en forma precisa. Mirando la Tabla1-1, en donde están enumeradas las instrucciones encontramos que algunas instrucciones tienen una descripción ambigua. Esto es debido a que la explicación de una instrucción en palabras es usualmente larga, y no hay espacio suficiente disponible en la tabla para una explicación tan larga. Mostraremos ahora que la función de las instrucciones del computador puede definirse precisamente por medio de una notación simbólica.

Símbolo	Decodificador de operación	Dirección efectiva	Palabra de memoria	Designación simbólica
AND	q_0	m	M	AC ← AC ^ M
ADD	q_1	m	M	EAC ← AC + M
LDA	q_2	m	M	AC ← M
STA	q_3	m	M	M ← AC
BUN	q_4	m	-	PC ← m
BSA	q_5	m	M	M ← PC, PC ← m + 1
ISZ	q_6	m	M	M ← M + 1, si (M + 1 = 0) entonces (PC ← PC + 1)

Tabla 1-3 Instrucciones de referencia de memoria.

La Tabla 1-3 enumera las siete instrucciones de referencia de memoria y sus funciones. La variable q_i en el decodificador de operación que pertenece a cada una de las instrucciones se incluye en la tabla. La dirección efectiva de la instrucción se designa por la letra m. Esta dirección especifica una palabra en la memoria (el operando) y esta palabra es designada por la letra M. La función simbólica de cada instrucción es una macro-operación y no una micro-operación. No es una micro-operación porque el enunciado

AC <-- AC + M

no puede ser realizado durante un pulso de reloj. Recuerde que los datos almacenados en las palabras de memoria no pueden procesarse directamente con registros externos. Los datos deben ser leídos de la memoria en los registros del procesador en donde puede ser operado su contenido binario. Para implementar la instrucción anterior necesitamos la siguiente secuencia de micro-operaciones

MAR <-- m

MBR <-- M

AC <-- AC + MBR

La dirección efectiva debe colocarse en MAR. Una operación de lectura de memoria lee el operando en MBR. Solamente después de que el operando está en MBR el control puede iniciar la micro-operación ADD entre dos registros de procesador.

Cuando se escribe un enunciado en notación simbólica, uno debe distinguir entre una macro-operación y una micro-operación. Siempre y cuando se haga esta distinción, no hay razón porque uno no pueda utilizar un lenguaje de transferencia de registro para especificar las macro-operaciones definidas por las instrucciones del computador.

Explicaremos ahora la función de cada una de las instrucciones separadamente y enumeraremos las micro-operaciones y funciones de control asociadas con cada una.

1.2.7.1 Instrucción AND

Esta es una instrucción que realiza la operación lógica de AND en pares de bits en el AC y la palabra de memoria especificada por la dirección efectiva. El resultado de la operación permanece en AC. Las micro-operaciones que ejecutan esta instrucción son;

$q_0c_2t_0$: MAR<-- MBR(AD) Transfiere la dirección efectiva

$q_0c_2t_1$: MBR <-- M Lee el operando

$q_0c_2t_2$: AC <-- AC ^ MBR AND con AC

c_2t_3: F <-- 0 Vaya al ciclo fetch

Las funciones de control para esta instrucción necesitan las variables q_0 y c_2. La primera reconoce el código de la operación AND y la segunda reconoce el ciclo de ejecución. Tres de las variables de tiempo que ocurren durante el ciclo de ejecución inician las micro-operaciones para leer el operando de la memoria y realizar la micro-operación de AND. En el instante t_3, F se aclara a 0. La señal de tiempo que ocurre después de t_3 es t_0 de tal manera que este t_0 encuentre el computador en el ciclo fetch. Note que q_0 no se utiliza con t_3. La condición para regresar al ciclo fetch es común a todas las instrucciones y pertenece al final del ciclo de ejecución no importa que la instrucción ya haya sido procesada. Como una consecuencia, el último enunciado en la secuencia anterior no se repetirá de nuevo en la lista de enunciados para las otras instrucciones.

El lector debe recordar que el contador de programa (PC) es incrementado durante el ciclo fetch y al mismo tiempo la instrucción corriente es leída de la memoria. Por consiguiente, cuando el control regresa al ciclo fetch después de ejecutar la instrucción presente, él encuentra en PC la dirección de la siguiente instrucción.

1.2.7.2 Instrucción ADD

Esta operación agrega el contenido de la palabra de memoria especificada por la dirección efectiva al valor presente en AC. La suma es transferida al AC y el final del acarreo al flip-flop E (EAC representa un registro que combina los registros E y AC). El bit de signo en la posición de más a la izquierda se trata como cualquier otro bit de acuerdo a la regla de suma en el complemento a 2 con signo. Las micro-operaciones que ejecutan esta instrucción son:

$q_1c_2t_0$: MAR<-- MBR(AD) Transfiere la dirección efectiva

$q_1c_2t_1$: MBR <-- M Lee el operando

$q_1c_2t_2$: EAC <-- AC + MBR Suma a AC y almacene el acarreo en E

1.2.7.3 Instrucción LDA

Esta instrucción transfiere la palabra de memoria (especificada por la dirección efectiva) en el AC. La palabra leída de la memoria en MBR puede transferirse al AC por la micro-operación AC <-- MBR. Un enunciado como este define una ruta directa del MBR a AC. Sin

embargo, podemos utilizar la ruta existente a través del sumador paralelo que se prevé para la implementación de la micro-operación add. Esta ruta puede utilizarse si el AC es aclarado antes de la transferencia. Utilizando la ruta del sumador, podemos especificar las micro-operaciones para la instrucción LDA como sigue:

$q_2c_2t_0$: MAR<-- MBR(AD) Transfiere la dirección efectiva

$q_2c_2t_1$: MBR <-- M, AC <-- 0 Lee el operando, aclare AC

$q_2c_2t_2$: AC <-- AC + MBR Suma a AC

El operando se lee en MBR y el AC es aclarado. La suma de MBR al contenido cero de AC resulta en la transferencia del contenido de MBR en el AC. Note que el acarreo no es transferido a E. No deseamos perturbar el valor de E en una transferencia directa.

1.2.7.4 Instrucción STA

Esta instrucción almacena el contenido presente del AC en la palabra de memoria especificada por la dirección efectiva. Las micro-operaciones que ejecutan esta instrucción son:

$q_3c_2t_0$: MAR<-- MBR(AD) Transfiere la dirección efectiva
$q_3c_2t_1$: MBR <-- AC Transfiere datos a MBR

$q_3c_2t_2$: M <-- MBR Almacena la palabra en la memoria

1.2.7.5 Instrucción BUN

Esta instrucción transfiere el programa a la instrucción especificada por la dirección efectiva. La instrucción es enumerada con las instrucciones de referencia de memoria porque ella tiene una parte de dirección. Sin embargo, no necesita una referencia a memoria para leer un operando (puede necesitar una referencia a la memoria para leer una dirección efectiva se I = 1). Recuerde que PC retiene la dirección de la instrucción para leer de la memoria en el ciclo siguiente de fetch. Normalmente, el PC es instrumentado para dar la dirección de la secuencia de instrucción siguiente. El programador tiene la prerrogativa de especificar cualquier otra instrucción fuera de la secuencia utilizando la instrucción BUN. Esta instrucción informa al control para tomar la dirección efectiva y la transfiere en PC. Durante el ciclo siguiente de fetch, esta dirección se convierte en la dirección de la instrucción que es leída de la memoria. La micro-operación que realiza esta función es:

$q_4c_2t_0$: PC<-- MBR(AD) Transfiere la dirección efectiva a PC

Las variables de tiempo t_1 y t_2 no se utilizan para esta instrucción y no iniciaran ninguna de las micro-operaciones.

Las instrucciones discutidas hasta ahora no son difíciles de comprender comparadas a la función de las dos instrucciones siguientes. Las dos instrucciones siguientes de referencia de memoria requieren de una secuencia de micro-operaciones un poco más complicada.

1.2.7.6 Instrucción BSA

Esta instrucción es útil para ramificar a una porción del programa denominada subrutina. Cuando se ejecuta, la instrucción almacena la dirección de la instrucción siguiente mantenida en PC (denominada la dirección de retorno) en la palabra especificada por la dirección efectiva m. El contenido de m + 1 es transferido en PC para servir como la dirección de la instrucción para el ciclo fetch siguiente (el comienzo de la subrutina). El retorno de la subrutina al programa en la dirección retenida se logra por medio de una instrucción indirecta BUN. Esta instrucción, cuando se coloca al final de la subrutina, hace que el programa se transfiera a la posición que él dejo cuando era ramificado a la subrutina.

Este proceso se muestra gráficamente en la Figura 1-7. La instrucción BSA realiza las macro-operaciones (ver Tabla 1-3).

M <-- PC, PC <-- m + 1

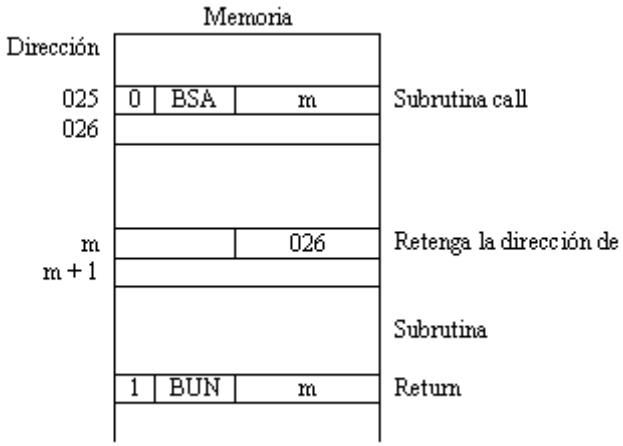

Figura 1-7 Demostración de llamada y retorno de subrutina.

Suponga que la instrucción BSA leída durante el ciclo fetch está en la dirección 25. Durante el ciclo de ejecución, PC contiene 26, puesto que fue incrementado durante el ciclo fetch. El contenido de PC es transferido a la palabra de memoria especificada por la dirección m, y m + 1 es colocado en PC. El ciclo siguiente de fetch encuentra a PC con el valor de m + 1, de tal manera que el control continúa para ejecutar el programa de subrutina. La última instrucción en la subrutina es una instrucción BUN indirecta (I=1) con una dirección m. Cuando esta instrucción es ejecutada, el control va al ciclo indirecto para leer la dirección efectiva en la localización m. El encuentra la dirección previa retenida 26, y la coloca en PC (Note que aquí la dirección efectiva es 26 y no m). El siguiente ciclo fetch encuentra PC con el valor 26 de tal manera que el control continúa ejecutando la instrucción en la dirección de regreso. La instrucción BSA realiza la función usualmente referida como llamada a subrutina. La última instrucción en la subrutina realiza la función conocida como un retorno de subrutina.

Las macro-operaciones para la instrucción BSA son implementadas en los registros del computador por una secuencia de micro-operaciones como sigue:

$q_5c_2t_0$: MAR<-- MBR(AD), Transfiere m a MAR

MBR(AD) <-- PC, Transfiere PC para ser almacenado

PC <--MBR(AD) Transfiere m a PC

$q_5c_2t_1$: M <-- MBR Almacene el contenido previo de PC

$q_5c_2t_2$: PC <-- PC + 1 Incremente m en PC

Se realizan tres micro-operaciones simultáneas durante t_0. Los contenidos de PC y MBR(AD) son intercambiados, y la dirección efectiva transferida a MAR. En este instante, m está en MAR y el contenido previo de PC en MBR. Una operación de escritura de memoria resulta en la implementación de M <-- PC. El valor previo de m, almacenado en PC, es incrementado en el instante t_2 para implementar PC <-- m + 1.

1.2.7.7 Instrucción ISZ

La instrucción de incrementar y saltar es útil para la modificación de dirección y para contar el número de veces que se ejecuta el circuito de un programa. Un número negativo previamente almacenado en la memoria en la dirección m se lee por medio de la instrucción ISZ. Este número es incrementado en 1 y almacenado de nuevo en la memoria. Si, después de que es incrementado, el número alcanza el valor 0, la siguiente instrucción se salta. Así, al final del circuito del programa uno inserta una instrucción ISZ seguida por una instrucción BUN. Si el número almacenado no alcanza cero, el programa regresa para ejecutar de nuevo el circuito. Si se alcanza el cero, la instrucción siguiente (BUN) se salta y el programa continúa para ejecutar las instrucciones fuera del circuito.

La función de la instrucción ISZ como se enuncia en la Tabla 1-3 se simboliza por los macro-enunciados.

M <-- M + 1

Si (M + 1 = 0) entonces (PC <-- PC + 1)

La secuencia de micro-operaciones que implementan la función es la siguiente:

$q_6c_2t_0$: MAR<-- MBR(AD) Transfiera la dirección efectiva
$q_6c_2t_1$: MBR <-- M Lea la palabra de la memoria

$q_6c_2t_2$: MBR <-- MBR + 1 Incremente el valor

$q_6c_2t_3$: M <-- MBR, Restaure la palabra incrementada

si (MBR = 0) entonces (PC <-- PC + 1) Salte si es cero

La palabra de memoria especificada por la dirección efectiva es leída de la memoria en el instante t_1. Es incrementada en MBR en t_2 (Recuerde que no se puede realizar ningún procesamiento con las palabras de memoria; deben ser leídas en un registro procesador tal como MBR al ser incrementadas). En t_3, la palabra incrementada es restaurada a la memoria, y PC es incrementado si la palabra en MBR es cero. Incrementando PC durante el ciclo de ejecución se produce un salto de una de las instrucciones en el programa porque también es incrementado durante el ciclo fetch.

1.2.7.8 Instrucciones de referencia de registro

Las instrucciones de referencia de registro son reconocidas por el control cuando $q_7 = 0$ e I = 0. Estas instrucciones utilizan los otros 12 bits del código para especificar una de las 12 micro-operaciones diferentes. Estos 12 bits están disponibles en MBR(5-16) durante el ciclo de ejecución.

Los enunciados de las micro-operaciones para las instrucciones de referencia de registro se enumeran en la Tabla 1-4. Estas instrucciones son ejecutadas con las variables de tiempo t_3, aunque cualquier variable de tiempo podría ser utilizada (excepto para la instrucción HLT).

Símbolo	Código hexadecimal	Función de control	Micro-operación
		$r = q_7 I' c_2 t_3$ $B_i = MBR(i)$	
CLA	7800	rB_5:	$AC \leftarrow 0$
CLE	7400	rB_6:	$E \leftarrow 0$
CMA	7200	rB_7:	$AC \leftarrow \overline{AC}$
CME	7100	rB_8:	$E \leftarrow \overline{E}$
CIR	7080	rB_9:	cir EAC
CIL	7040	rB_{10}:	cil EAC
INC	7020	rB_{11}:	$EAC \leftarrow AC + 1$
SPA	7010	rB_{12}:	Si (AC(1) = 0) entonces ($PC \leftarrow PC + 1$)
SNA	7008	rB_{13}:	Si (AC(1) = 1) entonces ($PC \leftarrow PC + 1$)
SZA	7004	rB_{14}:	Si (AC = 0) entonces ($PC \leftarrow PC + 1$)
SZE	7002	rB_{15}:	Si (E = 0) entonces ($PC \leftarrow PC + 1$)
HLT	7001	rB_{16}:	$S \leftarrow 0$

Tabla 1-4 Instrucciones de referencia de registro.

Cada función de control necesita la relación Booleana $q_7 I' c_2 t_3$ que designamos por conveniencia con el símbolo r. La función de control se distingue por un bit en MBR(5-16) el cual es igual a 1. Asignando el símbolo B_i al bit i de MBR, Todas las funciones de control pueden simplificarse denotándolas por rB_i.

Por ejemplo la instrucción CLA tiene el código hexadecimal 7800 que da el equivalente binario 0111 1000 0000 0000. El primer bit es un cero y es reconocido en I´. Los siguientes tres bits constituyen el código de operación y son reconocidos en la salida del decodificador q_7. El bit 5 en MBR es 1 y es reconocido en B_5. La función de control que inicia la micro-

operación para esta instrucción es $q_7I'c_2t_3B_5 = rB_5$. Note que, puesto que las instrucciones de referencia de registro operan en un solo registro, pueden ser especificadas por enunciados de micro-operación.

Las primeras siete instrucciones de referencia de registro realizan micro-operaciones de aclaramiento, complemento, desplazamiento circular, e incremento en los registros AC y/o E. Las cuatros instrucciones siguientes realizan un salto de la instrucción siguiente en la secuencia cuando una condición enunciada se satisface. El salto de la instrucción se logra incrementando PC una vez de nuevo (además de incrementarlo dutante el ciclo fetch). Los enunciados de la condición de control deben reconocerse como parte del requisito de control. El AC es positivo cuando su bit de signo AC(1) = 0; es negativo cuando AC(1) = 1. El contenido de AC es cero (AC = 0) si todas las celdas del registro son cero. La instrucción HLT aclara el flip-flop S de arranque-parada y detiene la secuencia de tiempo.

1.2.8 Unidades de Entrada-Salida

Un computador no es de utilidad si no se puede comunicar con el ambiente externo. Las instrucciones y los datos almacenados en la memoria deben salir de algún dispositivo de entrada. Los resultados del cálculo deben transmitirse al usuario a través de algún dispositivo de salida. Los computadores comerciales incluyen muchos tipos diferentes de dispositivos de entrada y salida. Para demostrar los requisitos más básicos para la comunicación de entrada y salida, utilizaremos como ilustración una máquina de escribir de teletipos para el computador básico.

Una máquina de escribir de teletipos, también conocida por su nombre de marca Teletipo, tiene un teclado eléctrico de máquina de escribir, una impresora, una lectora de cinta de papel, y una perforadora de cinta de papel. El dispositivo de entrada consta o del teclado de la máquina de escribir o de la lectora de papel, con un interruptor manual disponible para seleccionar el que se debe seleccionar. El dispositivo de salida consta de la impresora de la máquina de escribir o de la perforadora de cinta de papel, con otro interruptor disponible para seleccionar cualquier dispositivo. La unidad tiene facilidad de producir una serie de pulsos equivalentes a un código binario del carácter cuya tecla se pulsa. Estos pulsos son transmitidos en un registro de desplazamiento y constituyen el carácter de entrada. Los pulsos en serie de un código de carácter alfanumérico son enviados a la impresora en donde son decodificados para determinar el carácter que se debe imprimir. La velocidad del teletipo es muy lenta, usualmente alrededor de diez caracteres por segundo.

Las instrucciones y datos son transferidos en el computador o en forma simbólica o a través del teclado o en forma binaria a través de la lectora de cinta de papel. Una instrucción de 16 bits o una palabra de datos binarios puede preparase en cinta de papel en dos filas de de ocho bits cada una. La primera fila perforada representa la mitad de la palabra y la segunda fila suministra la otra mitad de la palabra. Las dos partes pueden empaquetarse en una palabra de un computador de 16 bits y almacenarse en la memoria.

1.2.8.1 Registros Entrada-Salida

La máquina de escribir teletipo envía y recibe información en serie. Cada cantidad de información tiene ocho bits de un código alfanumérico. La información en serie del teclado es desplazada en un registro de entrada. La información en serie para el impresor es

almacenada en un registro de salida. Estos dos registros se comunican con la máquina de escribir teletipo en forma serie y con el AC en paralelo. La configuración del registro se muestra en la Figura 1-8.

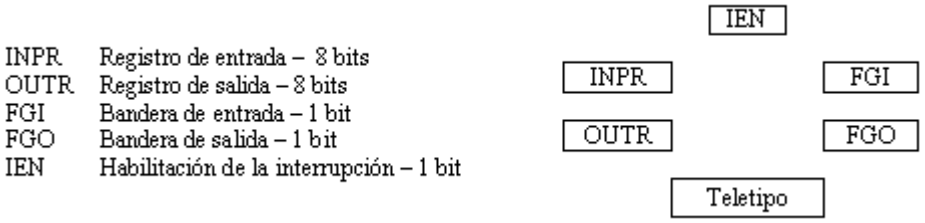

Figura 1-8 Registro de entrada, salida e interrupción.

El registro de entrada INPR consta de ocho bits y retiene una información de entrada alfanumérica. La bandera de entrada de un bit FGI es un flip-flop de control. El bit de bandera es llevado a set cuando la nueva información está disponible en el dispositivo de entrada y es aclarada cuando la información es aceptada por el computador. La bandera es necesaria para diferenciar la tasa de tiempo diferencial entre el dispositivo de entrada y el computador. El proceso de transferencia de información es como sigue: Inicialmente, la bandera de entrada FGI es aclarada. Cuando se pulsa una tecla, el código de ocho bits es desplazado en INPR y la bandera de entrada es colocada en 1. Siempre y cuando que se coloque la bandera, la información en INPR no puede cambiar golpeando cualquier tecla. El computador verifica el bit de bandera; si es 1, la información de INPR es transferida en paralelo en el AC y el FGI es aclarado. Una vez que la bandera es aclarada, la nueva información puede desplazarse a INPR golpeando otra tecla.

El registro de salida OUTR trabaja en forma similar pero la dirección del flujo de información es invertida. Inicialmente, la bandera de salida (FGO) se coloca en 1. El computador verifica el bit de bandera; si es 1, la información de AC es transferida en paralelo a OUTR y el FGO es aclarado. El dispositiva de salida acepta la información codificada, imprime el carácter correspondiente, y cuando la operación se completa, coloca FGO en 1. El computador no carga un nuevo carácter en OUTR cuando FGO es 0 debido a que esta condición indica que el dispositivo de salida está en el proceso de impresión del carácter.

El proceso de comunicación que se acaba de describir se conoce como una transferencia controlada por programa. El computador se mantiene verificando el bit de bandera y cuando lo encuentra en set, inicia una transferencia de información. La diferencia de la tasa de flujo de información entre el procesador y la unidad dispositivo entrada-salida hace ineficiente este tipo de transferencia. Para ver por qué esto es ineficiente, considere un computador que puede ir a través de los ciclos fetch y ejecución en 10 µs. El dispositivo entrada-salida puede transferir información a una tasa máxima de 10 caracteres por segundo. Esto es equivalente a un carácter por cada 100.000 µs. Dos instrucciones son ejecutadas cuando el computador verifica el bit bandera y decide no transferir la información. Esto significa, que a la máxima tasa, el computador verifica la bandera 5.000 veces entre cada transferencia. El computador está perdiendo tiempo mientras chequea la bandera en vez de hacer otra tarea de procesamiento útil.

Una alternativa, para un procedimiento controlado por programa es dejar que el dispositivo externo informe al computador cuando está listo para la transferencia. Mientras tanto el procesador puede estar ocupado con otras tareas. Este tipo de transferencia utiliza la opción de interrupción. Mientras que el computador está corriendo un programa, él no verifica las banderas. Sin embargo, cuando se coloca una bandera en el computador éste se interrumpe momentáneamente, deja de proceder con el programa actual y es informado del hecho de que se ha colocado una bandera. El computador desvía momentáneamente lo que está haciendo para encargarse de la transferencia de entrada o salida. Regresa entonces al programa corriente para continuar lo que estaba haciendo antes de la interrupción. El hardware que ejecuta este procedimiento se explica a continuación. Por ahora explicamos la función del flip-flop habilitación-interrupción que se muestra en la Figura1-8.

El flip-flop de habilitación-interrupción (IEN) puede colocarse y aclararse por dos instrucciones. Cuando es aclarado, las banderas no pueden interrumpir el computador y se desprecia. Cuando IEN se coloca en set, el computador puede interrumpirse. Este flip-flop proporciona al programador capacidad de tomar decisiones por el uso o no uso de la facilidad de interrupción. Si emite una instrucción para aclarar IEN, entonces está diciendo en efecto que no desea que su programa sea interrumpido. Si lo coloca en set, tiene disponible la facilidad de interrupción a su disposición.

1.2.8.2 Instrucciones de Entrada-Salida

Las instrucciones entrada-salida son necesarias para transferir información hacia y desde el registro AC, para verificar los bits de bandera, y para controlar el flip-flop de habilitación-interrupción. Estas instrucciones se enumeran en la Tabla 1-5.

Símbolo	Código hexadecimal	Descripción	Función
INP	F800	Entre el caracter a AC	AC(9-16) ← INPR, FGI ← 0
OUT	F400	Saque el carácter de AC	OUTR ← AC(9-16), FGO ← 0
SKI	F200	Salte en la bandera de entrada	Si (FGI = 1) entonces (PC ← PC + 1)
SKO	F100	Salte en la bandera de salida	Si (FGO = 1) entonces (PC ← PC + 1)
ION	F080	Encendido de interrupción	IEN ← 1
IOF	F040	Apagado de interrupción	IEN ← 0

Tabla 1-5 Instrucciones entrada-salida.

Ellas tienen un código de operación 111 con I = 1, que da para el primer dígito de la instrucción el dígito hexadecimal F. Los 12 bits restantes contienen un solo 1 y once ceros para cada una de las instrucciones. La instrucción INP transfiere la información de entrada a los ocho bits de más bajo orden del AC y también aclara la bandera de entrada. La instrucción OUT transfiere ocho bits a los registros de salida y aclara la bandera. Las siguientes dos instrucciones verifican el estado de las banderas y producen un salto de la instrucción siguiente si la bandera es 1. La instrucción que se salta será normalmente una instrucción de ramificación para retornar y verificar la bandera de nuevo. Esta instrucción no se salta si la bandera es 0. Si la bandera es 1, la instrucción ramificadora se salta y se

ejecuta una instrucción de entrada o de salida. Las dos últimas instrucciones colocan y aclaran el bit habilitación-interrupción, respectivamente.

Las micro-operaciones necesarias para ejecutar cada una de las instrucciones se enumeran en la Tabla 1-5 bajo la columna función. Ellas son ejecutadas por el control durante el ciclo de ejecución c_2 en el instante t_3. Las instrucciones entrada-salida se reconocen por la variable q_7I y el bit B_i en MBR que es igual a 1. Así, la instrucción INP es ejecutada con la función de control siguiente:

$q_7Ic_2t_3B_5$: AC(9-16) <-- INPR, FGI <-- 0

Las otras instrucciones tienen una función de control similar, excepto para la variable B_i la cual es diferente en cada una.

1.2.9 Interrupciones

La última fase del ciclo de instrucción consiste en que la Unidad de Control comprueba si hay alguna petición de interrupción pendiente sin atender. Si es así, se produce un salto a la rutina de servicio de interrupción. A continuación se describe el ciclo de interrupción y la implementación de este ciclo.

1.2.9.1 El Ciclo de Interrupción

Cuando los flip-flop de control de ciclo F y R son ambos 1, el computador se va a un ciclo de interrupción. Este ciclo se reconoce por el decodificador de ciclo de salida c_3 (ver Tabla 1-2). Es iniciado con el ciclo de ejecución después de que se completa la instrucción corriente. Previamente se supone que la última micro-operación en el ciclo de ejecución es

c_2t_3: F <-- 0

Que regresa el control al ciclo fetch. Se puede modificar esta condición como se muestra en el diagrama de flujo de la Figura 1-9. IEN se verifica al final de cada ciclo de ejecución. Si es 0, indica que el programador no desea utilizar la interrupción, de tal manera que el control va al siguiente ciclo de fetch aclarando F. Si IEN es 1, el control verifica los bits de la bandera. Si ambas banderas son 0, indican que ni los registros de entrada o de salida están listos para la transferencia de información. En este caso, el control se regresa al ciclo de fetch. Si cualquier bandera está colocada en 1, el control va al ciclo de interrupción haciendo R igual a 1(F ya está colocado en 1 durante el ciclo de ejecución).

El ciclo de interrupción es una implementación del hardware de una operación de ramificación y retención de memoria. La dirección corriente en PC es almacenada en una localización específica en donde puede encontrarse posteriormente cuando el programa regrese a la instrucción en la cual fue interrumpido. Esta localización puede ser un registro de procesador o una localización de memoria. Escogemos aquí la localización de memoria en la dirección 0 y la palabra para almacenar la dirección de retorno. El control entonces inserta la dirección 1 en PC y se mueve al siguiente ciclo fetch. También aclara IEN para

que no ocurran más interrupciones hasta la próxima requisición de interrupción de la bandera que ha sido servida.

Al comienzo del siguiente ciclo fetch, la instrucción que es leída de la memoria está en dirección 1 puesto que este es el contenido de PC. En la posición de memoria 1, el programador debe almacenar una instrucción de ramificación que envíe el computador a un programa de servicio en donde él verifica las banderas, determina cual bandera es colocada y entonces transfiere la información requerida de entrada o salida. Una vez que esto se hace, la instrucción ION es ejecutada para colocar IEN en 1 (para habilitar otras interrupciones), y se ejecuta la siguiente instrucción regresando el programa a la localización donde fue interrumpido. La instrucción que regresa el computador al programa original se muestra en la Figura 1-9.

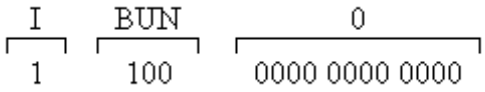

Figura 1-9 Instrucción que regresa el computador al programa original

Esta es una instrucción indirecta de ramificación con una parte de dirección igual a 0. Después de que esta instrucción es leída durante el ciclo fetch, el control va al ciclo indirecto (debido a que I = 1) para leer la dirección efectiva. Pero la localización efectiva está en la localización 0 y en la dirección de regreso que fue almacenada allí durante el ciclo interrupción previo. La ejecución de la instrucción anterior resulta en colocar en PC la dirección de retorno de localización 0.

1.2.9.2 Implementación del ciclo de Interrupción

Estamos entonces en disposición de enumerar las relaciones de transferencia de registros para el ciclo de interrupción. El ciclo de interrupción es iniciado al final del ciclo de ejecución por las siguientes micro-operaciones:

c_2t_3: Si [IEN ^ (FGI v FGO) = 1] entonces (R <-- 1)

Si [IEN ^ (FGI v FGO) = 0] entonces (F <-- 0)

Los símbolos ^ y v son para los operadores lógicos AND y OR, respectivamente. Estos enunciados de control condicional reemplazan las micro-operaciones que se supusieron previamente que aclararon F incondicionalmente. Estos enunciados están de acuerdo con los requisitos especificados en el diagrama de flujo de la Figura1-10.

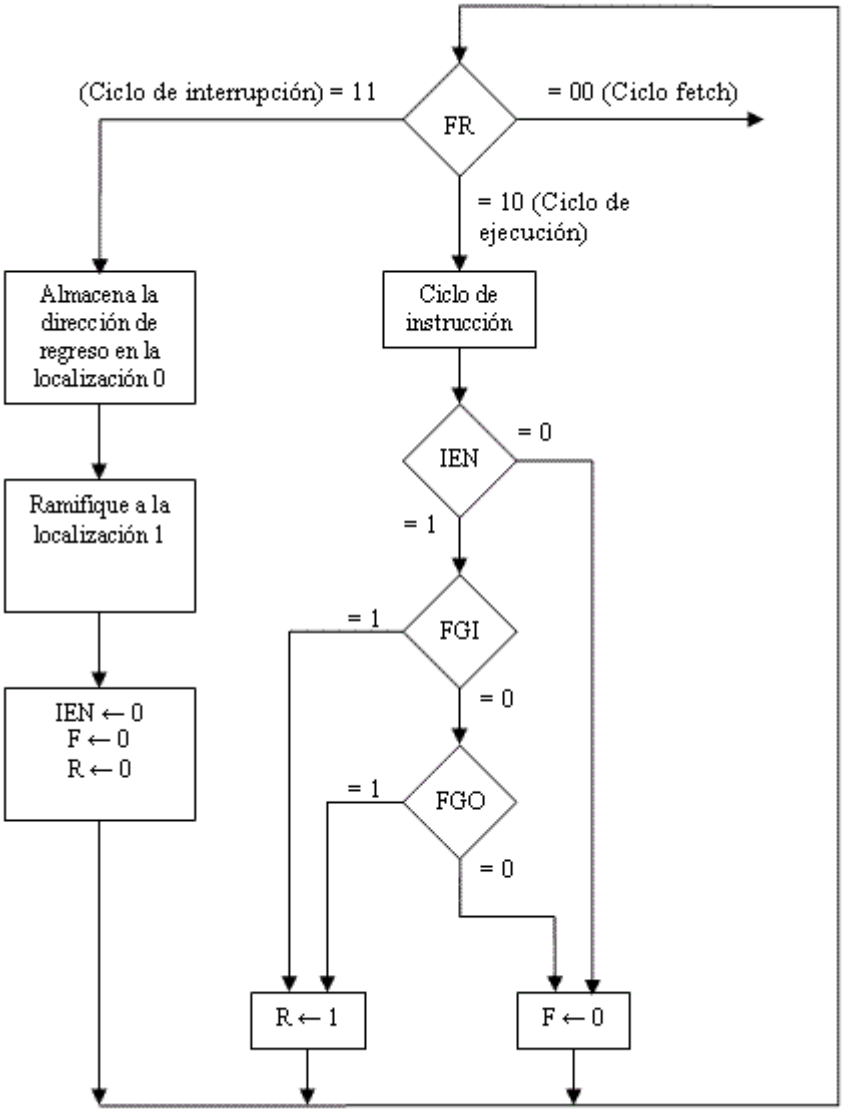

Figura 1-10 Diagrama de flujo para el ciclo de interrupción.

Note que cuando la instrucción ION (encendido de interrupción) es ejecutada por el control, es hecha durante el tiempo t_3 como sigue:

$q_7Ic_2B_9t_3$: IEN <-- 1

La micro-operación puede ser ejecutada al mismo tiempo que se ejecuta la micro-operadión condicional enumerada arriba debido a que ellas tienen variables comunes en sus funciones de control. Si esto ocurre, y IEN = 0 durante t_3, el computador va al ciclo de fetch y IEN no se vuelve 1 hasta el t_0 siguiente (en el ciclo de fetch). Así una instrucción ION no causa una interrupción (si esta colocada una bandera) hasta el final del siguiente ciclo de ejecución. Esto asegura que la dirección de retorno se coloque en PC antes de que ocurra otra interrupción.

El ciclo de interrupción ocurre cuando F = R = 1 lo cual hace la variable $c_3 = 1$. Las micro-operaciones para el ciclo de interrupción son:

c_3t_0: MBR(AD) <-- PC, PC <-- 0 Transfiere la dirección de retorno y aclara PC

c_3t_1: MAR <-- PC, PC <-- PC + 1 Transfiere 0 a MAR, coloca PC en 1

c_3t_2: M <-- MBR, IEN <-- 0 Almacena la dirección de retorno y aclara la habilitación interrupción

c_3t_3: F <-- 0, R <-- 0 Va al ciclo de fetch

La dirección de retorno está en PC y debe ser transferida a MBR para almacenarla en la memoria. El PC es aclarado y entonces se transfiere a MAR. En el instante t_2, MAR contiene 0 y MBR tiene la dirección de retorno. Una operación de escritura de memoria almacena la dirección de retorno en la localización 0. El PC se incrementa en t_1 para que contenga la dirección 1 al comienzo del siguiente ciclo fetch. IEN es aclarado y el control regresa al ciclo de fetch.

1.3 EMULADOR

En informática, un emulador es un software que permite ejecutar programas de computadora en una plataforma (arquitectura hardware o sistema operativo) diferente de aquella para la cual fueron escritos originalmente. A diferencia de un simulador, que sólo trata de reproducir el comportamiento del programa, un emulador trata de modelar de forma precisa el dispositivo que se está emulando.

La mayoría de los emuladores solo emulan una determinada arquitectura de hardware - si el sistema de explotación (o sistema operativo) también se requiere para emular cierto programa entonces ha de ser emulado también. Tanto el sistema de explotación como el programa deben ser interpretados por el emulador, como si estuviese ejecutándose en el equipo original. Aparte de la interpretación del lenguaje de la máquina emulada, es preciso emular el resto del equipo, como los dispositivos de entrada y salida, de forma virtual.

En vez de una emulación completa del equipo, una compatibilidad superficial puede ser suficiente. Esto traduce las llamadas del sistema emulado a llamadas del sistema anfitrión.

Típicamente, un emulador se divide en módulos que corresponden de forma precisa a los subsistemas del equipo emulado. Lo más común, es que un emulador este compuesto por los siguientes módulos:

- Un emulador de CPU o un simulador de UCP (ambos términos son en la mayoría de los casos intercambiables).
- Un módulo para el subsistema de memoria.
- Varios emuladores para los dispositivos de entrada y salida.

Lo más común es que los BUSes no sean emulados, por razones de simplicidad y rendimiento, y para que los periférico virtuales se comuniquen directamente con la UCP y los subsistemas de memoria (Wikipedia, 2015).

1.4 MAQUINA VIRTUAL

En informática una máquina virtual es un software que emula a una computadora y puede ejecutar programas como si fuese una computadora real. Este software en un principio fue definido como "un duplicado eficiente y aislado de una máquina física".

La acepción del término actualmente incluye a máquinas virtuales que no tienen ninguna equivalencia directa con ningún hardware real.

Una característica esencial de las máquinas virtuales es que los procesos que ejecutan están limitados por los recursos y abstracciones proporcionados por ellas.

Estos procesos no pueden escaparse de esta "computadora virtual". Uno de los usos domésticos más extendidos de las máquinas virtuales es ejecutar sistemas operativos para "probarlos" (Wikipedia, 2015).

Emulación de un sistema no nativo

Las máquinas virtuales también pueden actuar como emuladores de hardware, permitiendo que aplicaciones y sistemas operativos concebidos para otras arquitecturas de procesador se puedan ejecutar sobre un hardware que en teoría no soportan.

Algunas máquinas virtuales emulan hardware que sólo existe como una especificación. Por ejemplo:

- La máquina virtual P-Code que permitía a los programadores de Pascal crear aplicaciones que se ejecutasen sobre cualquier computadora con esta máquina virtual correctamente instalada.
- La máquina virtual de Java.
- La máquina virtual del entorno .NET.
- Open Firmware

Esta técnica permite que cualquier computadora pueda ejecutar software escrito para la máquina virtual. Sólo la máquina virtual en sí misma debe ser portada a cada una de las plataformas de hardware (Wikipedia, 2015).

1.5 COMPILADORES E INTÉRPRETES

Se llama programa a una secuencia de instrucciones que describe cómo ejecutar cierta tarea. Los circuitos electrónicos de cada computadora pueden reconocer y ejecutar directamente un conjunto limitado de instrucciones simples. Todos los programas que se desee ejecutar en una computadora deben convertirse previamente en una secuencia de estas instrucciones simples.

El conjunto de instrucciones primitivas de una computadora forma un lenguaje con el cual podemos comunicarnos con ella. Dicho lenguaje se llama lenguaje de máquina.

Los diseñadores de una nueva computadora deben decidir qué instrucciones incluir en su lenguaje de máquina. Normalmente intentan hacer las instrucciones primitivas lo más simples posibles, a fin de reducir la complejidad y el costo de la electrónica que se necesite.

Debido a que la mayoría de los lenguajes de máquina son demasiados elementales, es difícil y tedioso utilizarlos.

Hay dos formas principales de atacar este problema; ambas incluyen el diseño de un nuevo conjunto de instrucciones, más convenientes para las personas que el ya incorporado en la máquina. Estas instrucciones, en conjunto forman un nuevo lenguaje que llamaremos L2, de modo semejante al que está incorporado a la máquina lo llamaman L1. Las dos aproximaciones que mencionamos antes difieren en el modo en que los programas escritos en L2 son ejecutados por la computadora que, después de todo, sólo pueden ejecutar programas escritos en su lenguaje de máquina, L1.

Técnica de compilación: un método para ejecutar un programa escrito en L2, consiste en sustituir primero cada instrucción por una secuencia equivalente de instrucciones en L1. El resultado es un nuevo programa escrito totalmente con instrucciones en L1. La computadora ejecutará entonces el nuevo programa en L1 y no el anterior en L2.

Técnica de interpretación: se escribir un programa en L1 que tome los programas en L2 como datos de entrada y los lleve a cabo, examinando cada vez, cada instrucción y ejecutando directamente la secuencia equivalente de instrucciones en L1. Esta técnica no requiere la generación previa de un nuevo programa en L1.

La interpretación y la compilación son bastantes similares. En ambos métodos, las instrucciones en L2 se llevan a cabo al ejecutar secuencias equivalentes de instrucciones en L1. La diferencia radica en que, en la traducción, todo programa en L2 se convierte en otro programa en L1. En la interpretación se ejecuta cada instrucción en L2 inmediatamente después de examinarla y decodificarla. No genera ningún programa traducido (Fennema, 1993).

1.5.1 Máquina por niveles

En vez de pensar en términos de compilador o interprete, a menudo conviene imaginar la existencia de una computadora hipotética o máquina virtual o máquina extendida cuyo lenguaje de maquina sea L2. Si la fabricación de tal máquina fuese suficientemente barata, no habría necesidad de tener L1 ni una maquina que ejecutara programas en L1. La gente podría escribir simplemente sus programas en L2 y hacer que la computadora los ejecutara directamente. A pesar de que una máquina virtual cuyo lenguaje fuese L2 sería demasiado cara de construir con circuitos electrónicos, se pueden escribir programas para ella y hacer que los compile o interprete un programa escrito en L1, y ejecutarlos en la computadora. En otras palabras se pueden escribir programas para las máquinas virtuales como si realmente existieran. Los lenguajes L1 y L2 no deben ser “demasiados diferentes” para que la traducción sea práctica y rápidamente realizable. Esta restricción significa a menudo que L2, aunque es mejor que L1, está aún lejos de ser ideal para la mayoría de las personas. Al estar cerca del lenguaje de la máquina, aún está muy distante del lenguaje de las personas. Sin embargo, esta situación está lejos de ser desesperante.

La solución más obvia consiste en inventar otro conjunto de instrucciones que esté más orientado a las personas y menos orientado a la maquina que L2. Este tercer conjunto también forma un lenguaje que llamaremos L3. La gente puede escribir programas en L3 como si existiera una máquina virtual cuyo lenguaje de máquina fuese L3. Tales programas pueden ser compilados a L2 o ejecutados por un interprete escrito en L2.

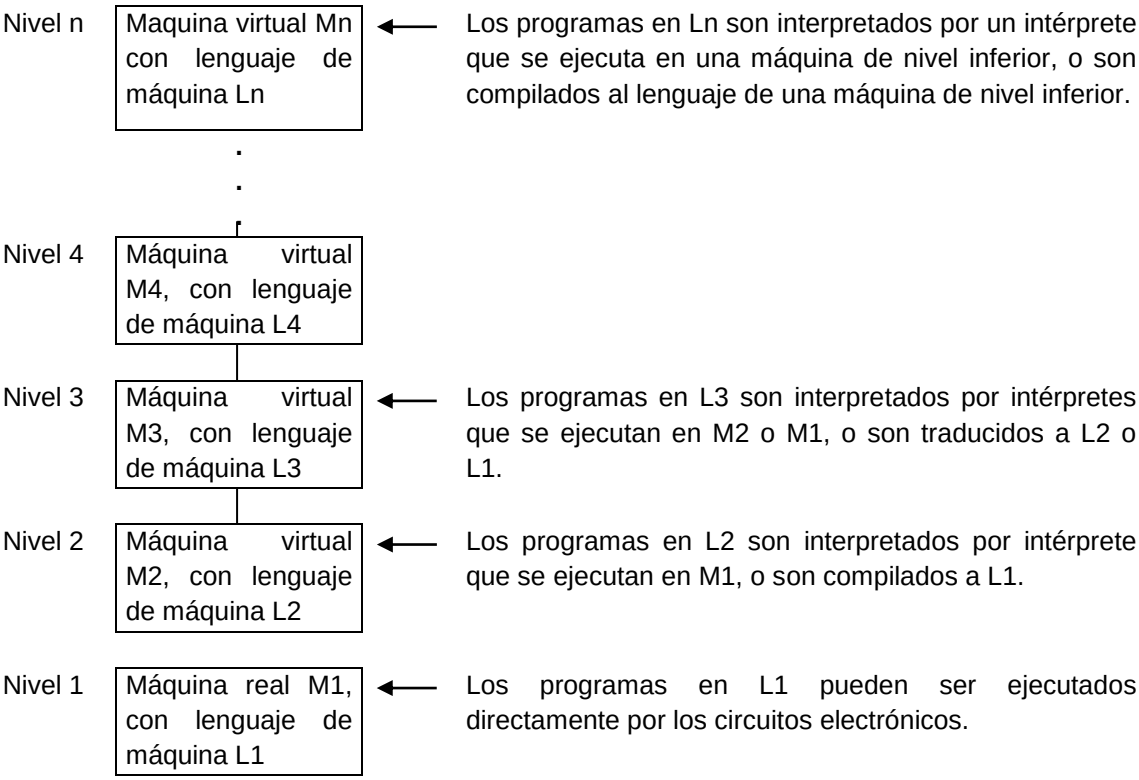

Figura 1-11 Máquina multinivel (Fennema, 1993).

La invención de una serie completa de lenguajes, cada uno más conveniente que sus predecesores, puede continuar indefinidamente hasta que se consiga uno adecuado. Cada lenguaje usa su predecesor como base, de manera que una computadora que use esta técnica puede considerarse como una serie de capas o niveles, uno encima del otro, como se muestra en la Figura 1-11. El lenguaje o nivel más bajo es el más simple y el de más arriba es el más complejo (Fennema, 1993).

1.5.2 Lenguajes, niveles y maquinas virtuales

Existe una importante relación entre un lenguaje y una máquina virtual. Cada máquina tiene algún lenguaje de máquina, que consta de todas las instrucciones que puede ejecutar. De hecho, una máquina define un lenguaje. En forma similar, un lenguaje define una máquina: la que puede ejecutar todos los programas escritos en ese lenguaje. Desde luego, la máquina definida por cierto lenguaje puede ser enormemente complicada y cara para construirla directamente con circuitos electrónicos, pero podemos imaginarla.

Una computadora con n niveles puede verse como n máquinas virtuales diferentes, cada una de las cuales tiene un lenguaje de máquina especial. Utilizaremos el término “nivel” y “máquina virtual” indistintamente. Los programas escritos en lenguaje L1 sólo pueden llevarlos a cabo directamente los circuitos electrónicos, sin que tenga que intervenir la compilación o interpretación. Los programas escritos en L2, L3, ..., Ln deben ser interpretados por un intérprete en un nivel inferior o compilados a otro lenguaje correspondiente a un nivel inferior (Fennema, 1993).

1.6 ANÁLISIS LÉXICO

Este punto trata sobre las técnicas para especificar e implantar analizadores léxicos. Una forma sencilla de crear un analizador léxico consiste en la construcción de un diagrama que ilustre la estructura de los componentes léxicos del lenguaje fuente, y después hacer la traducción "a mano" del diagrama a un programa para encontrar los componentes léxicos. De esta forma, se pueden producir analizadores léxicos eficientes.

Como la programación dirigida por patrones es de mucha utilidad, se introduce un lenguaje de patrón-acción, llamado LEX, para especificar los analizadores léxicos. En este lenguaje, los patrones se especifican por medio de expresiones regulares, y un compilador de LEX puede generar un reconocedor de las expresiones regulares mediante un autómata finito eficiente (Aho et al, 1990).

1.6.1 Función del analizador léxico

El analizador léxico es la primera fase de un compilador. Su principal función consiste en leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer el análisis. Esta interacción, esquematizada en la Figura 1-12, suele aplicarse convirtiendo al analizador léxico en una subrutina o corrutina del analizador sintáctico. Recibida la orden "obtén el siguiente componente léxico" del analizador sintáctico, el analizador léxico lee los caracteres de entrada hasta que pueda identificar el siguiente componente léxico.

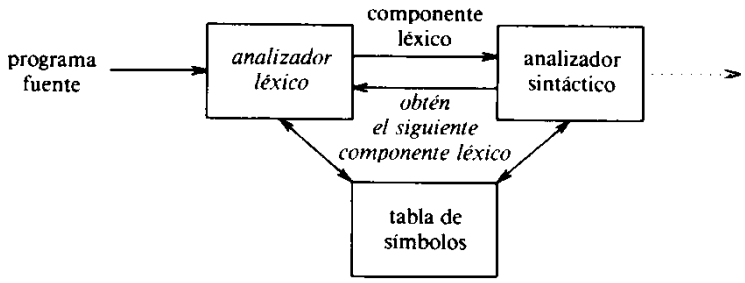

Figura 1-12 Interacción de un analizador léxico con el analizador sintáctico

Como el analizador léxico es la parte del compilador que lee el texto fuente, también puede realizar ciertas funciones secundarias en la interfaz del usuario, como eliminar del programa fuente comentarios y espacios en blanco en forma de caracteres de espacio en blanco, caracteres TAB y de línea nueva. Otra función es relacionar los mensajes de error del compilador con el programa fuente. Por ejemplo, el analizador léxico puede tener localizado el número de caracteres de nueva línea detectados, de modo que se pueda asociar un número de línea con un mensaje de error.

Aspectos del análisis léxico

Hay varias razones para dividir la fase de análisis de la compilación en análisis léxico y análisis sintáctico.

1. Un diseño sencillo es quizá la consideración más importante. Separar el análisis léxico del análisis sintáctico a menudo permite simplificar una u otra de dichas fases. Si se

está diseñando un lenguaje nuevo, la separación de las convenciones léxicas de las sintácticas puede dar origen a un diseño del lenguaje más claro.

2. Se mejora la eficiencia del compilador. Un analizador léxico independiente permite construir un procesador especializado y potencialmente más eficiente para esta función. Con técnicas especializadas de manejo de *buffers* para la lectura de caracteres de entrada y procesamiento de componentes léxicos se puede mejorar significativamente el rendimiento de un compilador.
3. Se mejora la transportabilidad del compilador. Las peculiaridades del alfabeto de entrada y otras anomalías propias de los dispositivos pueden limitarse al analizador léxico.

   Se han diseñado herramientas especializadas que ayudan a automatizar la construcción de analizadores léxicos y analizadores sintácticos cuando están separados.

Componentes léxicos, patrones y lexemas

Cuando se menciona el análisis sintáctico, los términos "componente léxico" *(token),* "patrón" y "lexema" se emplean con significados específicos. En la Tabla 1-6 aparecen ejemplos de dichos usos. En general, hay un conjunto de cadenas en la entrada para el cual se produce como salida el mismo componente léxico. Este conjunto de cadenas se describe mediante una regla llamada *patrón* asociado al componente léxico. Se dice que el patrón *concuerda* con cada cadena del conjunto. Un lexema es una secuencia de caracteres en el programa fuente con la que concuerda el patrón para un componente léxico. Por ejemplo, en la proposición de Pascal

const pi = 3.1416;

La subcadena pi es un lexema para el componente léxico "identificador".

COMPONENTE LÉXICO	LEXEMAS DE EJEMPLO	DESCRIPCIÓN INFORMAL DEL PATRÓN
const	const	const
if	if	if
relación	<, <=, =, <>, >, >=	< o <= o = o <> o >= o >
id	pi, cuenta, D2	letra seguida de letras y dígitos
núm	3.1416, 0, 6.02E23	cualquier constante numérica
literal	"vaciado de memoria"	cualquier carácter entre " y ", excepto "

Tabla 1-6 Ejemplos de componentes léxicos

Los componentes léxicos se tratan como símbolos terminales de la gramática del lenguaje fuente, con nombres en **negritas** para representarlos.

En la mayoría de los lenguajes de programación, se consideran componentes léxicos las siguientes construcciones: palabras clave, operadores, identificadores, constantes, cadenas literales y signos de puntuación, como paréntesis, coma y punto y coma. En el ejemplo

anterior, cuando la secuencia de caracteres pi aparece en el programa fuente, se devuelve al analizador sintáctico un componente léxico que representa un identificador. La devolución de un componente léxico a menudo se realiza mediante el paso de un número entero correspondiente al componente léxico.

Un patrón es una regla que describe el conjunto de lexemas que pueden representar a un determinado componente léxico en los programas fuente. El patrón para el componente léxico **const** de la Tabla 1-6 es simplemente la cadena sencilla const que deletrea la palabra clave. El patrón para el componente léxico **relación** es el conjunto de los seis operadores relacionales de Pascal. Para describir con precisión los patrones para componentes léxicos más complejos, como **id** (para identificador) y **núm** (para número), se utilizará la notación de expresiones regulares.

Errores léxicos

Son pocos los errores que se pueden detectar simplemente en el nivel léxico porque un analizador léxico tiene una visión muy restringida de un programa fuente.

Las posibles acciones de recuperación de errores son:

1. borrar un carácter extraño
2. insertar un carácter que falta
3. reemplazar un carácter incorrecto por otro correcto
4. intercambiar dos caracteres adyacentes.

Se puede probar este tipo de transformaciones de error para intentar reparar la entrada. La más sencilla de tales estrategias consiste en observar si un prefijo de la entrada restante se puede transformar en un lexema válido mediante una sola transformación de error. Esta estrategia da por supuesto que la mayoría de los errores léxicos se deben a una sola transformación de error, suposición que normalmente, pero no siempre, se cumple en la práctica.

1.6.2 Especificación de los componentes léxicos

Las expresiones regulares son una notación importante para especificar patrones. Cada patrón concuerda con una serie de cadenas, de modo que las expresiones regulares servirán como nombres para conjuntos de cadenas.

Cadenas y lenguajes

El término *alfabeto* o *clase de carácter* denota cualquier conjunto finito de símbolos. Ejemplos típicos de símbolos son las letras y los caracteres. El conjunto {0, 1} es el *alfabeto binario.* Los códigos ASCII y EBCDIC son dos ejemplos de alfabetos de computador.

Una *cadena* sobre algún alfabeto es una secuencia finita de símbolos tomados de ese alfabeto. En teoría del lenguaje, los términos *frase* y *palabra a* menudo se utilizan como sinónimos del término "cadena". La longitud de una cadena *s,* que suele escribirse |*s*|, es el número de apariciones de símbolos en *s.* Por ejemplo, camino es una cadena de longitud

seis. La cadena *vacía,* representada por ε, es una cadena especial de longitud cero. En la Tabla 1-7 se recogen algunos términos comunes asociados con las partes de una cadena.

El término *lenguaje* se refiere a cualquier conjunto de cadenas de un alfabeto fijo. Esta definición es muy amplia, y abarca lenguajes abstractos como *Ø,* el conjunto *vacío,* o {ε} y el conjunto que sólo contiene la cadena vacía, así como al conjunto de todos los programas de Pascal sintácticamente bien formados.

Si *x* e *y* son cadenas, entonces la *concatenación* de *x* e *y,* que se escribe *xy,* es la cadena que resulta de agregar *y* a *x.* Por ejemplo, si *x* = caza e *y* = fortunas, entonces *xy* = cazafortunas. La cadena vacía es el elemento identidad que se concatena. Es decir, *s*ε = ε*s*= *s.*

Cuando se considera la concatenación como un "producto", cabe definir la "exponenciación" de cadenas de la siguiente manera. Se define s^0 como ε, y para *i* >0 se define s^icomo $s^{i-1}s$. Dado que ε*s* es *s,* $s^1 = s$. Entonces, $s^2 = ss$, $s^3 = sss$, etcétera.

TÉRMINO	DEFINICIÓN
prefijo de *s*	Una cadena que se obtiene eliminando cero o más símbolosdesde la derecha de la cadena *s;* por ejemplo, ban es un prefijode bandera.
sufijo de *s*	Una cadena que se forma suprimiendo cero o más símbolosdesde la izquierda de una cadena *s;* por ejemplo, era es un sufijo de bandera.
subcadena de *s*	Una cadena que se obtiene suprimiendo un prefijo y un sufijode s; por ejemplo, ande es una subcadena de bandera. Todoprefijo y sufijo de *s* es una cadena de *s,* pero no toda subcadena de *s* es un prefijo o un sufijo de *s.* Para toda cadena *s,*tanto *s* como ε son prefijos, sufijos y subcadenas de *s.*
prefijo, sufijo o subcadena *propios* de *s*	Cualquier cadena no vacía *x* que sea, respectivamente, un prefijo, sufijo o subcadena de *s* tal que *s* es distinto de *x.*
subsecuencia de *s*	Cualquier cadena formada mediante la eliminación de cero o más símbolos no necesariamente contiguos a s; por ejemplo, bada es una subsecuencia de bandera.

Tabla 1-7 Términos de partes de una cadena (Aho et al, 1990).

Operaciones aplicadas a lenguajes

Hay varias operaciones importantes que se pueden aplicar a los lenguajes. Para el análisis léxico, interesan principalmente la unión, la concatenación y la cerradura, definidas en la Figura 1-15. También se puede extender el operador de "exponenciación" a los lenguajes

definiendo L^0 como {ε}, y L^icomo $L^{i-1}L$. Por tanto, L^ies L concatenado consigo mismo i - 1 veces.

Ejemplo Sea *L* el conjunto {A, B, . . . , Z, a, b , . . ., z} y *D* el conjunto {0, 1, . . . , 9}. Se pueden considerar *L* y *D* de dos maneras: *L,* como el alfabeto que contiene el conjunto de letras mayúsculas y minúsculas, y *D,* como el alfabeto que contiene el conjunto de los diez dígitos decimales. Se puede dar el caso, puesto que un símbolo puede ser considerado como una cadena de longitud uno, de que *L* y *D* sean los dos lenguajes finitos. Los siguientes son algunos ejemplos de nuevos lenguajes creados a partir de *L* y *D* mediante la aplicación de los operadores definidos en la Tabla 1-8.

1. *L* U *D* es el conjunto de letras y dígitos.
2. *LD* es el conjunto de cadenas que consta de una letra seguida de un dígito.
3. L^4es el conjunto de todas las cadenas de cuatro letras.
4. *L** es el conjunto de todas las cadenas de letras, incluyendo ε, la cadena vacía.

OPERACIÓN	DEFINICIÓN
unión de *L* y *M,* que se escribe *L* U *M*	*L* U *M* = { *s* | *s* está en L o *s* está en M }
concatenación de *L* y *M,* que se escribe *LM*	*LM* = { *st* **|** *s* está en *L* y *t* está en *M* }
cerradura de Kleene de *L,* que se escribe *L**	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^*denota "cero o más concatenaciones de" *L.*
cerradura positiva de *L,* que se escribe L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+denota "una o más concatenaciones de" *L.*

Tabla 1-8 Definiciones de operaciones sobre lenguajes (Aho et al, 1990)

1. *L*(*L* U *D*)* es el conjunto de todas las cadenas de letras y dígitos que comienzan con una letra.
2. *D*+ es el conjunto de todas las cadenas de uno o más dígitos.

Expresiones regulares

En Pascal, un identificador es una letra seguida de cero o más letras o dígitos. En esta sección, se presenta una notación, llamada expresiones regulares. Con esta notación, se pueden definir los identificadores de Pascal como

letra (letra | dígito) *

La barra vertical aquí significa "o", los paréntesis se usan para agrupar subexpresiones, el asterisco significa "cero o más casos de" la expresión entre paréntesis, y la yuxtaposición de letra con el resto de la expresión significa concatenación.

Una expresión regular se construye a partir de expresiones regulares más simples utilizando un conjunto de reglas definitorias. Cada expresión regular *r* representa un lenguaje *L(r)*. Las reglas de definición especifican cómo se forma *L(r)* combinando de varias maneras los lenguajes representados por las subexpresiones de *r.*

Las siguientes son las reglas que definen las *expresiones regulares del alfabeto* Σ. Asociada a cada regla hay una especificación del lenguaje representado por la expresión regular que se está definiendo.

1. ε es una expresión regular designada por {ε}; es decir, el conjunto que contiene la cadena vacía.
2. Si *a* es un símbolo de Σ, entonces *a* es una expresión regular designada por {*a*}; por ejemplo, el conjunto que contiene la cadena *a.* Aunque se usa la misma notación para las tres, técnicamente, la expresión regular *a* es distinta de la cadena*a* o del símbolo *a.* El contexto aclarará si se habla de *a* como expresión regular, cadena o símbolo.
3. Suponiendo que *r* y *s* sean expresiones regulares representadas por los lenguajes *L*(*r*)y*L*(*s*), entonces,

a) (*r*) | (*s*) es una expresión regular representada por *L*(*r*)U*L*(*s*).
b) (*r*)(*s*) es una expresión regular representada por *L*(*r*)*L*(*s*).
c) (*r*)* es una expresión regular representada por (*L*(*r*))*.
d) (*r*)es una expresión regular representada por *L*(*r*).

Se dice que un lenguaje designado por una expresión regular es un *conjunto regular.*

Se pueden evitar los paréntesis innecesarios en las expresiones regulares si se adoptan las convenciones:

1. el operador unario * tiene la mayor precedencia y es asociativo por la izquierda.
2. la concatenación tiene la segunda mayor precedencia y es asociativa por la izquierda.
3. | tiene la menor precedencia y es asociativo por la izquierda.

Según estas convenciones, *(a)* | *((b)*(c))* es equivalente a *a|b*c.* Estas dos expresiones designan el conjunto de cadenas que tienen una sola *a,* o cero o más *b* seguidas de una *c.*

Ejemplo Sea Σ = {*a, b*}.

1. La expresión regular *a* | *b* designa el conjunto {*a, b*}.
2. La expresión regular *(a* | *b)(a* | *b)* se indica con {*aa, ab, ba, bb*}, el conjunto de todas las cadenas de *a* y *b* de longitud dos. Otra expresión regular para este mismo conjunto es *aa* | *ab* | *ba* | *bb.*
3. La expresión regular *a** designa el conjunto de todas las cadenas de cero o más *a,* por

ejemplo, {ε, *a, aa, aaa, . . .* }.

4. La expresión regular (*a* |*b*)* designa el conjunto de todas las cadenas que contienen cero o más casos de una *a* o *b,* es decir, el conjunto de todas las cadenas de *a* y *b.* Otra expresión regular para este conjunto es *(a* b*)*.*
5. La expresión regular *a* | *a** *b* designa el conjunto que contiene la cadena *a* y todas las que se componen de cero o más *a* seguidas de una *b.*

Si dos expresiones regulares *r* y *s* representan al mismo lenguaje, se dice que *r* y *s* son *equivalentes* y se escribe *r = s.* Por ejemplo, (*a* |*b*) = (*b* |*a*).

AXIOMA	DESCRIPCIÓN
r | s=s | r	| es conmutativo
r|(s|t) = (r|s)|t	| es asociativo
(rs)t = r(st)	la concatenación es asociativa
r(s|t) = rs|rt *(s|t)r = sr|tr*	la concatenación distribuye sobre |
εr = r ***rε = r***	ε es el elemento identidad para la concatenación
r = (r |**ε)***	la relación entre * y ε
*r** = *r**	* es idempotente

Tabla 1-9 Propiedades algebraicas de las expresiones regulares (Aho et al, 1990).

Son varias las leyes algebraicas que obedecen las expresiones regulares y pueden ser utilizadas para transformar las expresiones regulares a formas equivalentes. En la Tabla 1-9 se muestran algunas leyes algebraicas que se cumplen para las expresiones regulares r, s y t.

Definiciones regulares

Por conveniencia de notación, puede ser deseable dar nombres a las expresiones regulares y definir expresiones regulares utilizando dichos nombres como si fueran símbolos. Si es un alfabeto de símbolos básicos, entonces una *definición regular* es una secuencia de definiciones de la forma

$d_1 \longrightarrow r_1$
$d_2 \longrightarrow r_2$
. . .
$d_n \longrightarrow r_n$

Donde cada $d_i$es un nombre distinto, y cada $r_i$es una expresión regular sobre los símbolos de Σ U{$d_1, d_2, \ldots, d_{i-1}$}, por ejemplo, los símbolos básicos y los nombres previamente definidos.

Para distinguir los nombres de los símbolos, se imprimen en **negritas** los nombres de las definiciones regulares.

Ejemplo: como ya se estableció antes, el conjunto de identificadores de Pascal es el conjunto de cadenas de letras y dígitos que empiezan con una letra. A continuación se da una definición regular para este conjunto.

letra —> A | B | . . . | Z | a | b | . . . | z
dígito —> 0 | 1 | ... | 9
id —> letra (letra | dígito)*

Abreviaturas en la notación

Ciertas construcciones aparecen con tanta frecuencia en una expresión regular, que es conveniente introducir algunas abreviaturas.

1. *Uno o más casos.* El operador unitario postfijo $^+$ significa "uno o más casos de". Si *r* es una expresión regular que designa al lenguaje *L*(*r*), entonces $(r)^+$ es una expresión regular que designa al lenguaje $(L(r))^+$.Así, la expresión regular a^+representa al conjunto de todas las cadenas de una o más *a.* El operador $^+$ tiene la misma precedencia y asociatividad que el operador *. Las dos identidades algebraicas $r^* = r^+ | \varepsilon$ y $r^+ = rr^*$ relacionan los operadores de la cerradura de Kleene y los de la cerradura positiva.
2. *Cero o un caso.* El operador unitario postfijo ? significa "cero o un caso de". La notación *r*?es una abreviatura de *r* | ε. Si *r* es una expresión regular, entonces (*r*)? es una expresión regular que designa el lenguaje *L(r)* U {ε}. Por ejemplo, usando los operadores $^+$ y ?, se puede escribir la definición regular para **núm** en la forma

dígito	—>	0 | 1 | ... | 9
dígitos	—>	**dígito**$^+$
fracción_optativa	—>	(.**dígitos)?**
exponente_optativo	—>	(E (+ | -)? **dígitos)?**
Núm	—>	**dígitos fracción_optativa exponente_optativo**

3. *Clases de caracteres.* La notación [abc], donde a, b y c son símbolos del alfabeto, designa la expresión regular a | b | c. Una clase abreviada de carácter como [a-z] designa la expresión regular a | b | . . . | z. Utilizando clases de caracteres, se puede definir los identificadores como cadenas generadas por la expresión regular

   [A-Za-z] [A-Za-z0-9]*

Conjuntos no regulares

Algunos lenguajes no se pueden describir con ninguna expresión regular. Para ilustrar los límites del poder descriptivo de las expresiones regulares, se dan a continuación ejemplos de construcciones de lenguajes de programación que no se pueden describir con expresiones regulares.

No se pueden utilizar las expresiones regulares para describir construcciones equilibradas o anidadas. Por ejemplo, el conjunto de todas las cadenas de paréntesis equilibrados no se puede describir con una expresión regular. Por otra parte, este conjunto se puede especificar mediante una gramática independiente del contexto.

Las cadenas de repetición no se pueden describir con expresiones regulares. El conjunto

{ *wcw* | *w* es una cadena de símbolos *a* y *b* }

No se puede representar con ninguna expresión regular, ni se puede describir con una gramática independiente del contexto.

Las expresiones regulares se pueden utilizar para designar sólo un número fijo de repeticiones o un número no especificado de repeticiones de una determinada construcción. No se pueden comparar dos números arbitrarios para comprobar si son iguales.

1.6.3 Un lenguaje para la especificacion de analizadores lexicos

Se han desarrollado algunas herramientas para construir analizadores léxicos a partir de notaciones de propósito especial basadas en expresiones regulares. Ya se ha estudiado el uso de expresiones regulares en la especificación de patrones de componentes léxicos. Antes de considerar los algoritmos para compilar expresiones regulares en programas de concordancia de patrones, se da un ejemplo de una herramienta que pueda ser utilizada por dicho algoritmo.

En el mercado existen dos herramienta que son comlpatibles, la herramienta LEX es una herramienta para sistema opertivo UNIX, tambien existe una version para Window denominada FLEX, estas dos herramientas son compatibles.

En esta sección se describe la herramienta LEX, muy utilizada en la especificación de analizadores léxicos para varios lenguajes. Esa herramienta se denomina *compilador LEX,* y la especificación de su entrada, *lenguaje LEX.* El estudio de una herramienta existente permitirá mostrar cómo, utilizando expresiones regulares, se puede combinar la especificación de patrones con acciones, por, ejemplo, haciendo entradas en una tabla de símbolos, cuya ejecución se pueda pedir a un analizador léxico. Se pueden utilizar las especificaciones tipo LEX aunque no se disponga de un compilador LEX; las especificaciones se pueden transcribir manualmente a un programa operativo empleando las técnicas de diagramas de transiciones.

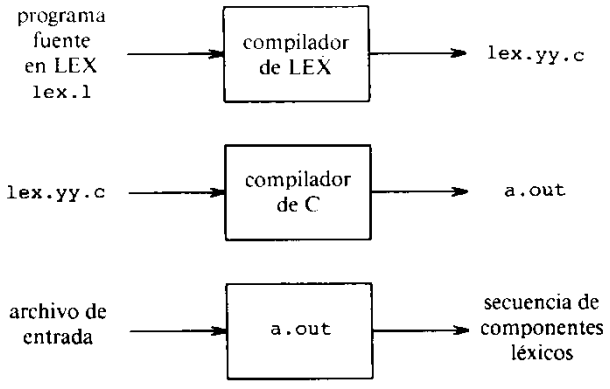

Figura 1-13 Creación de un analizador léxico con LEX.

Por lo general, se utiliza el LEX de la forma representada en la Figura 1-13. Primero, se prepara una especificación del analizador léxico creando un programa lex.l en lenguaje LEX. Después lex.1 se pasa por el compilador LEX para producir el programa en C lex.yy.c. El programa lex.yy.c consta de una representación tabular de un diagrama de transiciones construido a partir de las expresiones regulares de lex.l, junto con una rutina estándar que utiliza la tabla para reconocer lexemas. Las acciones asociadas a las expresiones regulares de lex.l son partes de código en C y se transfieren directamente a lex.yy.c. Por último, lex.yy.c se ejecuta en el compilador de C para producir un programa objeto a.out, que es el analizador léxico que transforma un archivo de entrada en una secuencia de componentes léxicos (Aho et al, 1990).

Especificaciones en LEX

Un programa en LEX consta de tres partes:

- declaraciones
- reglas de traducción
- procedimientos auxiliares

La sección de declaraciones incluye declaraciones de variables, constantes manifiestas y definiciones regulares. (Una constante manifiesta es un identificador que se declara para representar una constante.) Las definiciones regulares son proposiciones similares a las estudiadas, y se utilizan como componentes de las expresiones regulares que aparecen en las reglas de traducción.

Las reglas de traducción de un programa en LEX son proposiciones de la forma

P_1 { *acción*$_1$ }
P_2 { *acción*$_2$ }
. . . . . .
P_n { *acción*$_n$ }

Donde P_i es una expresión regular y cada *acción* es un fragmento de programa que describe cuál ha de ser la acción del analizador léxico cuando el patrón P_i concuerda con un lexema. En LEX, las acciones se escriben en C, en general, sin embargo, pueden estar en cualquier lenguaje de implantación.

La tercera sección contiene todos los procedimientos auxiliares que puedan necesitar las acciones. A veces, estos procedimientos se pueden compilar por separado y cargar con el analizador léxico.

Un analizador léxico creado por LEX se comporta en sincronía con un analizador sintáctico como sigue. Cuando es activado por el analizador sintáctico, el analizador léxico comienza a leer su entrada restante, un carácter a la vez, hasta que encuentre el mayor prefijo de la entrada que concuerde con una de las expresiones regulares P_i. Entonces, ejecuta la *acción*$_i$. Generalmente, *acción*$_i$ devolverá el contro1 al analizador sintáctico. Sin embargo, si no lo hace, el analizador léxico se dispone a encontrar más lexemas, hasta que una acción hace que el contro1 regrese al analizador sintáctico. La búsqueda repetida de lexemas hasta encontrar una instrucción **return** explícita permite al analizador léxico procesar espacios en blanco y comentarios de manera apropiada.

El analizador léxico devuelve una única cantidad, el componente léxico, al analizador sintáctico. Para pasar un valor de atributo con la información del lexema, se puede asignar una variable global llamada **yylval.**

Ejemplo. La Figura 1-14 es un programa en LEX que reconoce los componentes léxicos y devuelve el componente léxico encontrado. Algunas observaciones sobre el código servirán para introducir muchas características importantes de LEX.

```
%{
        /* definición de las constantes manifiestas MEN, MEI, IGU,
           DIF, MAY, MAI,
           IF, THEN, ELSE, ID, NUMERO, OPREL */ %}
/* definiciones regulares */
delim       [ \t\n]
eb          {delim}+
letra       [A-Za-z]
dígito      [0-9]
id          {letra}({letra}      {dígito})*
número      {dígito}+(\.{dígito}+)?(E[+\-]?{dígito}+)?
%%
{eb}        {/* no hay acción ni se devuelve nada */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval   =  instala_id(); return(ID);}
{número}    {yylval   =  instala_núm; return(NUMERO);}
"<"         {yylval   =  MEN; return(OPREL);}
"<="        {yylval   =  MEI; return(OPREL);}
"="         {yylval   =  IGU; return(OPREL);}
"<>"        {yylval   =  DIF; return(OPREL);}
">"         {yylval   =  MAY; return(OPREL);}
">="        {yylval   =  MAI; return(OPREL);}
%%
instala_id() {
        /* procedimiento para instalar el lexema, cuyo primer carácter está
           apuntado por yytexto y cuya longitud es yylong, dentro de la tabla de
           símbolos y devuelve un apuntador a él */
}
instala_núm() {
       /* procedimiento similar para instalar un lexema que es un número */
}
```

Figura 1-14 Programa en LEX para los componentes léxicos.

1.7 ANÁLISIS SINTÁCTICO

Todo lenguaje de programación tiene reglas que prescriben la estructura sintáctica de programas bien formados. Se puede describir la sintaxis de las construcciones de los lenguajes de programación por medio de gramáticas independientes del contexto o notación BNF (forma de Backus-Naur).

1.7.1 El papel del analizador sintáctico

En este modelo de compilador, el analizador sintáctico obtiene una cadena de componentes léxicos del analizador léxico, como se muestra en la Figura 1-15, y comprueba si la cadena pueda ser generada por la gramática del lenguaje fuente. Se supone que el analizador sintáctico informará de cualquier error de sintaxis de manera inteligible. También debería recuperarse de los errores que ocurren frecuentemente para poder continuar procesando el resto de su entrada.

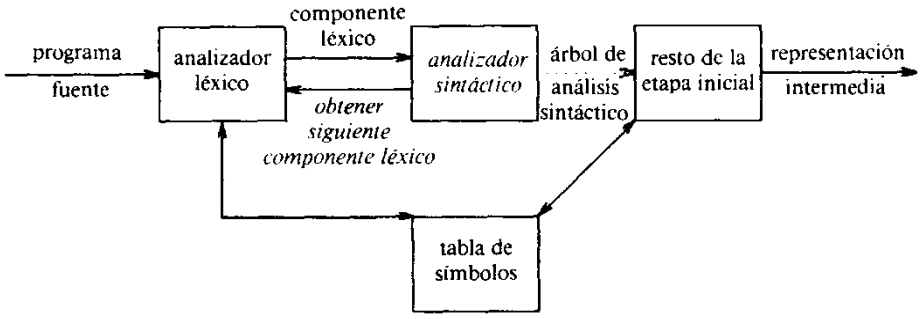

Figura 1-15 Posición del analizador sintáctico en el modelo del compilador.

Los métodos empleados generalmente en los compiladores se clasifican como descendentes o ascendentes. Como sus nombres indican, los analizadores sintácticos descendentes construyen árboles de análisis sintáctico desde arriba (la raíz) hasta abajo (las hojas), mientras que los analizadores sintácticos ascendentes comienzan en las hojas y suben hacia la raíz. En ambos casos, se examina la entrada al analizador sintáctico de izquierda a derecha, un símbolo a la vez.

Se asume que la salida del analizador sintáctico es una representación del árbol de análisis sintáctico para la cadena de componentes léxicos producida por el analizador léxico. En la práctica, hay varias tareas que se pueden realizar durante el análisis sintáctico, como recoger información sobre distintos componentes léxicos en la tabla de símbolos, realizar la verificación de tipo y otras clases de análisis semántico, y generar código intermedio (Aho et al, 1990).

Manejo de errores sintácticos

Los programadores a menudo escriben programas incorrectos, y un buen compilador debería ayudar al programador a identificar y localizar errores.

Se sabe que los programas pueden contener errores de muy diverso tipo. Por ejemplo, los errores pueden ser:

- léxicos, como escribir mal un identificador, palabra clave u operador

- sintácticos, como una expresión aritmética con paréntesis no equilibrados
- semánticos, como un operador aplicado a un operando incompatible
- lógicos, como una llamada infinitamente recursiva

El manejador de errores en un analizador sintáctico tiene objetivos fáciles de establecer.

- Debe informar de la presencia de errores con claridad y exactitud.
- Se debe recuperar de cada error con la suficiente rapidez como para detectar errores posteriores.
- No debe retrasar de manera significativa el procesamiento de programas correctos.

¿Cómo debe informar un manejador de errores de la presencia de un error? Al menos debe informar del lugar en el programa fuente donde se detecta el error, porque es muy probable que el error real se haya producido en alguno de los componentes léxicos anteriores.

Una vez detectado el error, ¿cómo se debe recuperar el analizador sintáctico? En la mayoría de los casos, no es adecuado que el analizador sintáctico abandone después de detectar el primer error, porque el posterior procesamiento de la entrada podría revelar más errores. Normalmente, hay alguna forma de recuperación del error donde el analizador sintáctico intenta volver a un estado en el que el procesamiento de la entrada pueda continuar.

Una estrategia de recuperación de errores son las *Producciones de error.* Si se tiene una buena idea de los errores comunes que pueden encontrarse, se puede aumentar la gramática del lenguaje con producciones que generen las construcciones erróneas. Entonces se usa esta gramática aumentada con las producciones de error para construir el analizador sintáctico. Si el analizador sintáctico usa una producción de error, se pueden generar diagnósticos de error apropiados para indicar la construcción errónea reconocida en la entrada.

1.7.2 Gramaticas independientes del contexto

Muchas construcciones de los lenguajes de programación tienen una estructura inherentemente recursiva que se puede definir mediante gramáticas independientes del contexto. Una gramática independiente del contexto (gramática, por brevedad) consta de terminales, no terminales, un símbolo inicial y producciones.

1. Los terminales son los símbolos básicos con que se forman las cadenas. "Componente léxico" es un sinónimo de "terminal" cuando se trata de gramáticas para lenguajes de programación. Las palabras clave **if, then y else** es un terminal.
2. Los no terminales son variables sintácticas que denotan conjuntos de cadenas. Los no terminales definen conjuntos de cadenas que ayudan a definir el lenguaje generado por la gramática. También imponen una estructura jerárquica sobre el lenguaje que es útil tanto para el análisis sintáctico como para la traducción.
3. En una gramática, un no terminal es considerado como el símbolo inicial, y el conjunto de cadenas que representa es el lenguaje definido por la gramática.
4. Las producciones de una gramática especifican cómo se pueden combinar los terminales y los no terminales para formar cadenas. Cada producción consta de un no terminal, seguido por una flecha, seguida por una cadena de no terminales y terminales.

Ejemplo. La gramática con las siguientes producciones define expresiones aritméticas simples.

expr —> expr op expr
expr —> (expr)
expr —> - expr
*expr —>***id**
op —> +
op —> -
*op —> **
op —> /
op —> ^

En esta gramática, los símbolos terminales son

id + - * / ^ ()

Los símbolos no terminales son *expr* y *op*, y *expr* es el símbolo inicial.

Derivaciones

La idea central es que se considera una producción como una regla de reescritura, donde el no terminal de la izquierda es sustituido por la cadena del lado derecho de la producción.

Por ejemplo, considérese la siguiente gramática para expresiones aritméticas, donde el no terminal *E* representa una expresión.

*E —> E + E | E*E* | (*E*) | *-E* | **id** **(1.1)**

La producción *E —>-E* significa que una expresión precedida por un signo menos es también una expresión. Esta producción se puede usar para generar expresiones más complejas a partir de expresiones más simples permitiendo sustituir cualquier presencia de *E* por *-E.* En el caso más simple, se puede sustituir una sola *E* por *-E.* Se puede describir esta acción escribiendo

E => -E

Que se lee "*E* deriva *-E*". Laproducción *E —>*(*E*)establece que también se podría sustituir una presencia de una *E* en cualquier cadena de símbolos gramaticales por (*E*); por ejemplo, *E*E =>*(*E*)**E* o *E*E => E**(*E*).

Se puede tomar una sola *E* y aplicar repetidamente producciones en cualquier orden para obtener una secuencia de sustituciones. Por ejemplo,

E => -E => - (*E*) => - (**id**)

A dicha secuencia de sustituciones se le llama *derivación* de - **(id)** a partir de *E.* Esta derivación proporciona una prueba de que un caso determinado de una expresión es la cadena - **(id)**.

De forma más abstracta, se dice que α*A*β => αγβ si *A —>* γes una producción y α y β son cadenas arbitrarias de símbolos gramaticales. Si α_1 => α_2 => . . . => α_n, se dice que α_1*deriva* a α_n. El símbolo => significa "deriva en un paso". A menudo se desea decir "deriva en cero o más pasos". Para este propósito se puede usar el símbolo $\overset{*}{\Rightarrow}$. Así:

1. $\alpha \overset{*}{\Rightarrow} \alpha$ para cualquier cadena α, y
2. Si $\alpha \overset{*}{\Rightarrow} \beta$ y β => γ, entonces $\alpha \overset{*}{\Rightarrow} \gamma$.

Del mismo modo se puede usar $\overset{+}{\Rightarrow}$ para expresar "deriva en uno o más pasos".

Dada una gramática *G* con símbolo inicial *S,* se puede utilizar la relación $\overset{+}{\Rightarrow}$ para definir *L*(*G*), el *lenguaje generado por G.* Las cadenas de *L*(*G*) pueden contener sólo símbolos terminales de *G.* Se dice que una cadena de terminales *w* está en *L*(*G*) si, y sólo si, $S \overset{+}{\Rightarrow} w$. A la cadena *w* se le llama *frase* de *G.* De un lenguaje que pueda ser generado por una gramática se dice que es un *lenguaje independiente del contexto.* Si dos gramáticas generan el mismo lenguaje, se dice que son *equivalentes.*

Si $S \overset{*}{\Rightarrow} \alpha$, donde α puede contener no terminales, entonces se dice que α es una *forma de frase* de *G.* Una frase es una forma de frase sin no terminales.

Para comprender cómo trabajan algunos analizadores sintácticos, hay que considerar derivaciones donde tan sólo el no terminal de más a la izquierda de cualquier forma de frase se sustituya a cada paso. Dichas derivaciones se denominan *por la izquierda.*

Si $\alpha \Rightarrow \beta$ mediante un paso en el que se sustituye el no terminal más a la izquierda de α, se escribe $\alpha \underset{mi}{\Rightarrow} \beta$.

Usando las convenciones de notación, todo paso por la izquierda se puede escribir $wA\gamma \underset{mi}{\Rightarrow} w\delta\gamma$, donde *w* consta sólo de terminales, $A \rightarrow \delta$ es la producción aplicada y γ es una cadena de símbolos gramaticales. Para subrayar el hecho de que α deriva a β por medio de una derivación por la izquierda, se escribe $\alpha \overset{*}{\underset{mi}{\Rightarrow}} \beta$. Si $S \overset{*}{\underset{mi}{\Rightarrow}} \alpha$, entonces se dice que α es una *forma de frase izquierda* de la gramática en cuestión.

Análogas definiciones se aplican a las derivaciones *derechas,* donde el no terminal más a la derecha se sustituye en cada paso. Las derivaciones derechas a menudo se denominan derivaciones *canónicas.*

Árboles de análisis sintáctico y derivaciones

Un árbol de análisis sintáctico se puede considerar como una representación gráfica de una derivación que no muestra la elección relativa al orden de sustitución. Cada nodo interior de un árbol de análisis sintáctico se etiqueta con algún no terminal *A,* y que los hijos de ese nodo se etiquetan, de izquierda a derecha, con los símbolos del lado derecho de la producción por la cual se sustituyó esta *A* en la derivación. Las hojas del árbol de análisis sintáctico se etiquetancon terminales o no terminales y, leídas de izquierda a derecha, constituyen una forma de frase, llamada el producto o frontera del árbol. Por ejemplo, en la Figura 1-16 se muestra el árbol de análisis sintáctico para –(id+id).

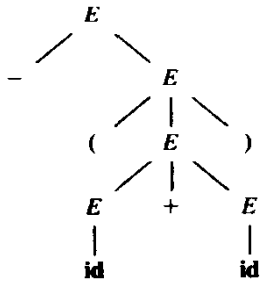

Figura 1-16 Arbol de análisis sintáctico para – (id + id).

Ejemplo**.** Se considera de nuevo la gramática (1.1) de expresiones aritméticas. La frase **id+ id*id** tiene las dos claras derivaciones por la izquierda:

E =>	*E* + *E*	*E* =>	*E***E*
=>	**id** + *E*	=>	*E* + *E***E*
=>	**id** + *E***E*	=>	**id** + *E***E*
=>	**id** + **id****E*	=>	**id** + **id****E*
=>	**id** + **id*id**	=>	**id** + **id*id**

Con los dos árboles de análisis sintáctico correspondientes que se muestran en la Figura 1-16.

Obsérvese que el árbol de análisis sintáctico de la Figura 1-17(a) refleja la precedencia comúnmente aceptada de + y *, mientras que el árbol de la Figura 1-17(b) no. Es decir, es habitual considerar que el operador * tiene mayor precedencia que +, lo cual corresponde al hecho de que una expresión como *a+b*c* normalmente se evaluaría como *a+(b*c)*, en lugar de como *(a+b)*c.*

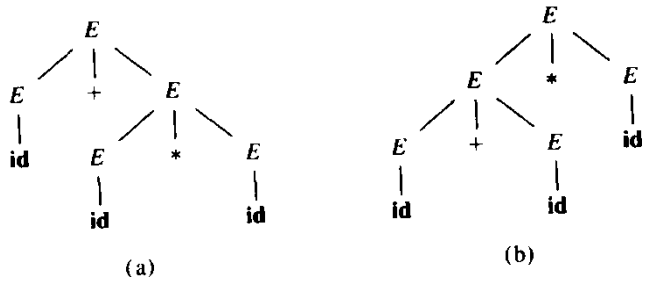

Figura 1-17 Dos árboles de análisis sintáctico para id + id*id.

Ambigüedad

Se dice que una gramática que produce más de un árbol de análisis sintáctico para alguna frase es *ambigua.* O, dicho de otro modo, una gramática ambigua es la que produce más de una derivación por la izquierda o por la derecha para la misma frase. Para algunos tipos de analizadores sintácticos es preferible que la gramática no sea ambigua, pues si lo fuera, no se podría determinar de manera exclusiva qué árbol de análisis sintáctico seleccionar para una frase. Para algunas aplicaciones se considerarán también los métodos mediante los cuales se puedan utilizar ciertas gramáticas ambiguas, junto con *reglas para eliminar*

ambigüedades que desechan árboles de análisis sintácticos indeseables, dejando sólo un árbol para cada frase.

1.7.3 Escritura de una gramática

Las gramáticas son capaces de describir la mayoría, pero no todas, de las sintaxis de los lenguajes de programación.

Esta sección se inicia considerando la división del trabajo entre un analizador léxico y un analizador sintáctico. Puesto que cada método de análisis sintáctico puede manejar sólo gramáticas de una cierta forma, quizá se deba reescribir la gramática inicial para hacerla analizable por el método elegido.

Supresión de la ambigüedad

A veces, una gramática ambigua se puede reescribir para eliminar la ambigüedad. Como ejemplo, se eliminará la ambigüedad de la siguiente gramática con **"else** ambiguo":

prop—> **if***expr* **then** *prop*

| **if***expr* **then** *prop* **else** *prop* (1.2)

| **otra**

Aquí, **otra** representa cualquier otra proposición. De acuerdo con esta gramática, la proposición condicional compuesta

if*E*$_1$**then** *S*$_1$**else if***E*$_2$**then** *S*$_2$**else** *S*$_3$

Tiene el árbol de análisis sintáctico que se muestra en la Figura 1-18. La gramática (1.2) es ambigua, puesto que la cadena

if*E*$_1$**then if** *E*$_1$ **then** *S*$_1$**else** *S*$_2$ (1.3)

Tiene los dos árboles de análisis sintáctico que se muestran en la Figura 1-19.

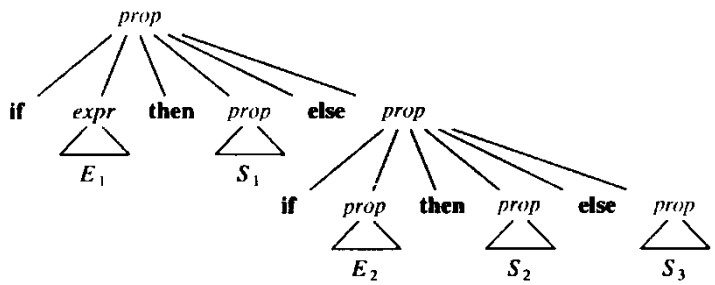

Figura 1-18 Arbol de análisis sintáctico para la proposición condicional.

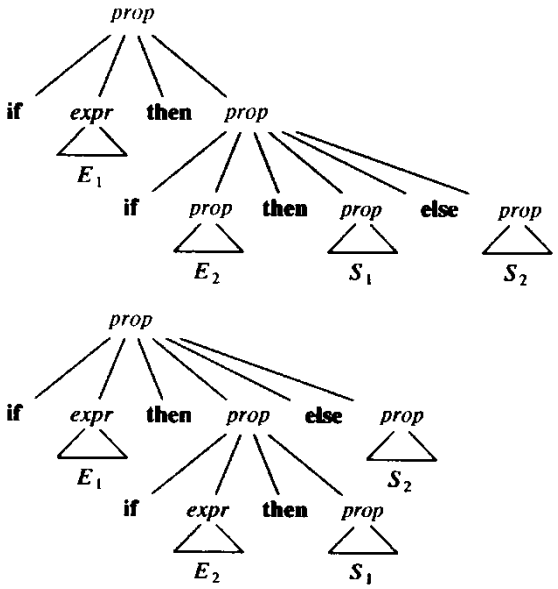

Figura 1-19 Dos árboles de análisis sintáctico para una frase ambigua.

En todos los lenguajes de programación con proposiciones condicionales de esta forma, se prefiere el primer árbol de análisis sintáctico. La regla general es, "emparejar cada **else** con el **then** sin emparejar anterior más cercano". Esta regla para eliminar ambigüedades se puede incorporar directamente a la gramática. Por ejemplo, se puede reescribir la gramática (1.2) como la siguiente gramática no ambigua. La idea es que una proposición que aparezca entre un **then** y un **else** debe estar "emparejada"; es decir, no debe terminar con un **then** sin emparejar seguido de cualquier proposición, porque entonces el **else** estaría obligado a concordar con este **then** no emparejado. Una proposición emparejada es o una proposición **if-then-else** que no contenga proposiciones sin emparejar o cualquier otra clase de proposición no condicional. Así, se puede utilizar la gramática

prop —> *prop_emparejada*
| *pro_no_emparejada*
prop_emparejada —> **if***expr***then***prop_emparejada***else***prop_emparejada*
| **otra** (1.4)
prop_no_emparejada —>**if***expr***then***prop*
| **if***expr***then***prop_emparejada***else***prop_no_emparejada*

Esta gramática genera el mismo conjunto de cadenas que (1.2), pero permite sólo un análisis sintáctico para la cadena (1.3), es decir, el que asocia cada **else** con el **then** sin emparejar anterior más cercano.

1.7.4 Analisis sintactico ascendente

El análisis sintáctico es el proceso de determinar si una cadena de componentes léxicos puede ser generada por una gramática. La mayoría de los métodos de análisis

sintáctico están comprendidos en dos clases, llamadas métodos descendente y ascendente.

En esta sección se introduce un estilo general de análisis sintáctico ascendente, conocido como análisis sintáctico por desplazamiento y reducción.

El análisis sintáctico por desplazamiento y reducción intenta construir un árbol de análisis sintáctico para una cadena de entrada que comienza por las hojas (el fondo) y avanza hacia la raíz (la cima). Se puede considerar este proceso como de "reducir" una cadena w al símbolo inicial de la gramática. En cada paso de reducción se sustituye una subcadena determinada que concuerde con el lado derecho de una producción por el símbolo del lado izquierdo de dicha producción y si en cada paso se elige correctamente la subcadena, se traza una derivación por la derecha en sentido inverso.

Mangos

Informalmente, un "mango" de una cadena es una subcadena que concuerda con el lado derecho de una producción y cuya reducción al no terminal del lado izquierdo de la producción representa un paso a lo largo de la inversa de una derivación por la derecha. En muchos casos, la subcadena situada más a la izquierda β que concuerda con el lado derecho de alguna producción *A* —>β no es un mango, porque una reducción por la producción *A* —>β produce una cadena no reducible al símbolo inicial. Por esta razón, se debe dar una definición más precisa de un mango.

Formalmente, un *mango* de una forma de frase derecha γ es una producción *A* —>β y una posición de γ donde la cadena β podría encontrarse y sustituirse por *A* para producir la forma de frase derecha previa en una derivación por la derecha de γ.

Implantación por medio de una pila del análisis sintáctico por desplazamiento y reducción

Hay dos problemas a resolver si se va a hacer el análisis sintáctico. El primero consiste en situar la subcadena a reducir en una forma de frase derecha, y el segundo, en determinar qué producción elegir en caso de que haya más de una producción con dicha subcadena en el lado derecho. Antes de considerar estas cuestiones, considérese primero el tipo de estructuras de datos que se debe utilizar en un analizador sintáctico por desplazamiento y reducción.

Un modo adecuado de implantar un analizador sintáctico por desplazamiento y reducción es mediante la utilización de una pila para manejar los símbolos gramaticales, y un *buffer* de entrada para manejar la cadena w que se ha de analizar. Se utiliza $ para marcar el fondo de la pila y el extremo derecho de la entrada. Al principio, la pila está vacía, y la cadena *w* está en la entrada, como sigue:

PILA	ENTRADA
	w $

El analizador sintáctico funciona desplazando cero o más símbolos de la entrada a la pila hasta que un mango β esté en su cima. Entonces, el analizador reduce β al lado izquierdo de la producción adecuada. El analizador repite este lazo hasta que detecta un error o hasta que la pila contiene el símbolo inicial y la entrada está vacía:

PILA	ENTRADA
$*S*	$

Después de esta configuración, el analizador se para y anuncia la terminación con éxito del análisis sintáctico.

Aunque las principales operaciones del analizador son el desplazamiento y la reducción, existen en realidad cuatro acciones posibles que un analizador por desplazamiento y reducción puede realizar: 1) desplazar, 2) reducir, 3) aceptar y 4) error.

1. En una acción de *desplazar,* el siguiente símbolo de entrada se desplaza a la cima de la pila.
2. En una acción de *reducir,* el analizador sabe que el extremo derecho del mango está en la cima de la pila. Entonces debe localizar el extremo izquierdo del mango dentro de la pila y decidir el no terminal con qué debe sustituir el mango.
3. En una acción de *aceptar,* el analizador anuncia la terminación con éxito del análisis sintáctico.
4. En una acción de *error,* el analizador descubre que se ha producido un error sintáctico y llama a una rutina de recuperación de errores.

Hay un hecho importante que justifica el uso de una pila en el análisis sintáctico por desplazamiento y reducción: el mango siempre aparecerá en la cima de la pila, nunca dentro.

Prefijos viables

Los prefijos de las formas de frase derecha que pueden aparecer en la pila de un analizador sintáctico por desplazamiento y reducción se denominan *prefijos viables.* Una definición equivalente de un prefijo viable es la de que es un prefijo de una forma de frase derecha que no continúa más allá del extremo derecho del mango situado más a la derecha de esta forma de frase. Con esta definición, siempre es posible añadir símbolos terminales al final de un prefijo viable para obtener una forma de frase derecha. Por tanto, aparentemente no hay error siempre que la porción examinada de la entrada hasta un punto dado pueda reducirse a un prefijo viable.

Conflictos durante el análisis sintáctico por desplazamiento y reducción

Existen gramáticas independientes del contexto para las cuales no se pueden utilizar analizadores sintácticos por desplazamiento y reducción. Todo analizador por desplazamiento y reducción para estas gramáticas puede alcanzar una configuración en la que el analizador sintáctico, conociendo el contenido total de la pila y el siguiente símbolo de entrada, no puede decidir si desplazar o reducir (un *conflicto de desplazamiento/reducción),* o no puede decidir qué tipo de reducción efectuar (un *conflicto de reducción/reducción).* Técnicamente, estas gramáticas no están dentro de la clase LR(*k*) de gramáticas; se les

denomina gramáticas no LR. La *k* de LR(*k*) se refiere al número de símbolos de preanálisis sobre la entrada. Por lo general, las gramáticas utilizadas en compilación se incluyen en la clase LR(1), con un símbolo de anticipación.

1.7.5 Generadores de analizadores sintacticos

Hay herramientas para la genración de analizaciones sintácticos, una de ellos es BISON, este generador se encuentra disponible como una orden del sistema UNIX, y se ha utilizado para facilitar la implantación de cientos de compiladores.

El generador de analizadores sintácticos YACC

Se puede construir un traductor utilizando YACC de la forma que se ilustra en la Figura 1-20. Primero, se prepara un archivo, por ejemplo traduce.y, que contiene una especificación en YACC del traductor. La orden

```
yacc traduce.y
```

Transforma al archivo traduce.y en un programa escrito en C llamado y.tab.c usando el método LALR. El programa y.tab.c es una representación de un analizador sintáctico escrito en C, junto con otras rutinas en C que el usuario pudo haber preparado. La tabla de análisis sintáctico LALR se comprime. Al compilar y.tab.c junto con la biblioteca ly que contiene el programa de análisis sintáctico LR utilizando la orden

```
cc y.tab.c -ly
```

Se obtiene el programa objeto deseado a.out que realiza la traducción especificada por el programa original en YACC. Si se necesitan otros procedimientos, se pueden compilar o cargar con y.tab.c, igual que en cualquier programa en C.

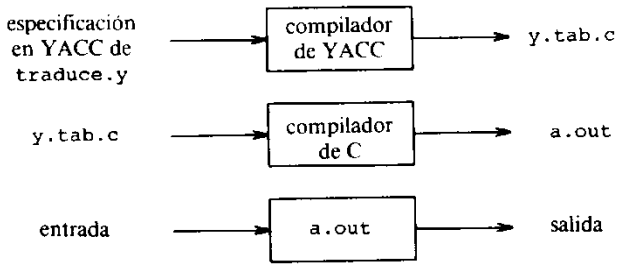

Figura 1-20 Creación de un traductor de entrada/salida con YACC.

Un programa fuente en YACC tiene tres partes:

```
declaraciones
%%
reglas de traducción
%%
rutinas en C de apoyo
```

1.7.6 Traducción dirigida por la sintaxis

Definiciones dirigidas por la sintaxis

Una *definición dirigida por la sintaxis* utiliza una gramática independiente del contexto para especificar la estructura sintáctica de la entrada. A cada símbolo de la gramática le asocia un conjunto de atributos y a cada producción, un conjunto de *reglas semánticas* para calcular los valores de los atributos asociados con los símbolos que aparecen en esa producción. La gramática y el conjunto de reglas semánticas constituyen la definición dirigida por la sintaxis.

Traducción dirigida por la sintaxis

Hay dos notaciones para asociar reglas semánticas con producciones, las definiciones dirigidas por la sintaxis y los esquemas de traducción. Las definiciones dirigidas por la sintaxis ocultan muchos detalles de la implantación y no es necesario que el usuario especifique explícitamente el orden en el que tiene lugar la traducción. Los esquemas de traducción indican el orden en que se deben evaluar las reglas semánticas, así que algunos detalles de la implantación quedan visibles.

Conceptualmente, tanto con las definiciones dirigidas por la sintaxis como con los esquemas de traducción, se analiza sintácticamente la cadena de componentes léxicos de entrada, se construye el árbol de análisis sintáctico y después se recorre el árbol para evaluar las reglas semánticas en sus nodos. La evaluación de las reglas semánticas puede generar código, guardar información en una tabla de símbolos, emitir mensajes de error o realizar otras actividades. La traducción de la cadena de componentes léxicos es el resultado obtenido al evaluar las reglas semánticas.

Hay casos especiales de definiciones dirigidas por la sintaxis que se pueden implantar en una sola pasada evaluando las reglas semánticas durante el análisis sintáctico, sin construir explícitamente un árbol de análisis.La implantación en una sola pasada es importante para la eficiencia en cuanto al tiempo de compilación. Una subclase importante, llamada las definiciones "con atributos por la izquierda", abarca prácticamentetodas las traducciones que se puedan realizar sin la construcción explícita de un árbol de análisis sintáctico.

Forma de una definición dirigida por la sintaxis

En una definición dirigida por la sintaxis, cada producción gramatical A —> α tiene asociado un conjunto de reglas semánticas de la forma $b := f(c_1, c_2, \ldots, c_k)$,donde *f* es una función, y, o bien

1. *b* es un atributo sintetizado de *A y* $c_1, c_2, \ldots, c_k$son atributos que pertenecen a los símbolos gramaticales de la producción, o bien
2. *b es un atributo heredado de uno de los símbolos gramaticales del lado derecho de la producción, y c1, c2, . . . , ckson atributos que pertenecen a los símbolos gramaticales de la producción.*

En cualquier caso, se dice que el atributo *b* depende de los atributos $c_1, c_2, \ldots, c_k$.

En una definición dirigida por la sintaxis, se asume que los terminales sólo tienen atributos sintetizados, ya que la definición no proporciona ninguna regla semántica para los terminales. El analizador léxico es el que proporciona generalmente los valores para los atributos de los terminales.

Atributos sintetizados

Se dice que un atributo está *sintetizado* si su valor en un nodo del árbol de análisis sintáctico se determina a partir de los valores de atributos de los hijos del nodo. Los atributos sintetizados tienen la atractiva propiedad de que se pueden calcular durante un solo recorrido ascendente del árbol de análisis sintáctico.

Ejemplo. En la Tabla 1-10 se muestra una definición dirigida por la sintaxis para traducir expresiones, formadas por dígitos separados por los signos más o menos, a notación postfija. A cada no terminal está asociado un atributo *t* con un valor de la cadena que representa la notación postfija de la expresión generada por ese no terminal en un árbol de análisis sintáctico.

PRODUCCIÓN	REGLA SEMÁNTICA
expr -> expr$_1$ + término	*expr.t := expr$_1$.t ||término.t || '+'*
expr -> expr$_1$ - término	*expr.t := expr$_1$.t || término.t || '-'*
expr -> término	*expr.t := término.t*
término ->0	*término.t := '0'*
término -> 1	*término.t := '1'*
. . .	. . .
término -> 9	*término.t := '9'*

Tabla 1-10 Definición dirigida por la sintaxis para traducción de infija a postfija

El operador || en las reglas semánticas representa la concatenación de cadenas.

Atributos heredados

Un atributo heredado es uno cuyo valor en un nodo de un árbol de análisis sintáctico está definido a partir de los atributos en el padre y/o de los hermanos de dicho nodo. Los atributos heredados sirven para expresar la dependencia de una construcción de un lenguaje de programación en el contexto en el que aparece.

Esquemas de traducción

Un esquema de traducción es una gramática independiente del contexto en la que se asocian atributos con los símbolos gramaticales y se insertan acciones semánticas encerradas entre llaves { } dentro de los lados derechos de las producciones. Se utilizarán esquemas de traducción como una notación útil para especificar la traducción durante el análisis sintáctico.

Cuando se diseña un esquema de traducción, se deben respetar algunas limitaciones para asegurarse de que el valor de un atributo esté disponible cuando una acción se refiera a él. Estas limitaciones, motivadas por las definiciones con atributos por la izquierda, garantizan que las acciones no hagan referencia a un atributo que aún no haya sido calculado.

El ejemplo más sencillo ocurre cuando sólo se necesitan atributos sintetizados. En este caso, se puede construir el esquema de traducción creando una acción que conste de una

asignación para cada regla semántica y colocando esta acción al final del lado derecho de la producción asociada. Por ejemplo, la producción y la regla semántica

PRODUCCIÓN	REGLA SEMÁNTICA
$T \longrightarrow T_1 * F$	$T.val := T_1.val \times F.val$

Dan como resultado la siguiente producción y acción semántica:

$T \longrightarrow T_1 * F \{ T.val := T_1.val \times F.val \}$

1.8 INGENIERÍA DE SOFTWARE

La ingeniería del software es una disciplina de la ingeniería, y una de sus metas es el desarrollo costeable de sistemas de software. Éste es abstracto e intangible, no está restringido por materiales, o gobernado por leyes físicas o por procesos de manufactura. De alguna forma, esto simplifica la ingeniería del software ya que no existen limitaciones físicas del potencial del software. Sin embargo, esta falta de restricciones naturales significa que el software puede llegar a ser extremadamente complejo y, por lo tanto, muy difícil de entender (Sommerville, 2005).

El Estándar 729/1993 de la IEEE define a la ingeniería del software como “La aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software, es decir, la aplicación de la ingeniería al software”.

Los objetivos generales de la ingeniería de software son:

- Desarrollo de software de calidad
- Aumento de la productividad
- Desarrollo de software económico

El desarrollo de software es un problema ingenieril ya que trata de crear soluciones efectivas y viables económicamente.

La Ingeniería de Software está conformada por (Pressman, 2005):

- Herramientas: soporte automático o semiautomático a los métodos, orientadas a etapas particulares en el diseño de un software (herramientas CASE[1]).
- Métodos: cómo se construye el software (planificación, análisis de los requisitos, diseño del sistema, codificación, prueba y mantenimiento).
- Procedimientos: secuencia en que se aplican los métodos, entregas y controles. Son los que unen los métodos con las herramientas.

1.8.1 Relevancia de la Ingeniería del Software

Las aplicaciones, sistemas y componentes de software son herramientas que ayudan a producir bienes y servicios que necesitan las personas. Actualmente la economía depende de la infraestructura y los sistemas de información, donde la ingeniería de software juega un rol principal.

[1] Computer Assisted Software Engineering

El tamaño de los problemas resueltos por software ha ido evolucionando desde los pequeños hasta los muy grandes, y este cambio de escala ha traído importantes cambios de complejidad, que, por un lado, afectan sin duda a las técnicas, pero que van mucho más allá y comprenden desde la naturaleza misma de los problemas, hasta la variedad multidisciplinar de los aspectos y áreas involucrados en los procesos.

1.8.2 Metodologias de desarrollo de software

En la literatura sobre este tema existen muchas definiciones sobre lo que es una metodología. Más o menos todas ellas coinciden en que debería tener al menos las siguientes características:

- Define como se divide un proyecto en fases y las tareas a realizar en cada una.
- Para cada una de las fases está especificado cuales son las entradas que reciben y las salidas que producen.
- Tienen alguna forma de gestionar el proyecto.

Teniendo esto en cuenta se establece que *metodología* es un modo sistemático de producir software.

Taxonomía de las metodologías

Existen metodologías en función de la mentalidad con la que se aborda el problema, las más conocidas y usadas son la metodología estructurada y la metodología orientada a objetos.

1.8.3 Metodología Estructurada

Aparecieron a fines de los 60’s con la programación estructurada, posteriormente a mediados de los 70’s extendidas con el diseño estructurado y a fines de los 70’s con el análisis estructurado. Ejemplos de metodologías estructuradas impulsadas por organismos gubernamentales lo constituyen: MERISE, METRICA (Perez Garcia, 2006). Otras metodologías estructuradas en el ámbito académico y comercial son las de Gane & Sarson, Ward & Mellor, Yourdon & De Marco entre otras.

Esta metodología crea los modelos del sistema de forma descendente. Son las orientadas a procesos, a datos y las mixtas. Intentan aplicar formas ingenieriles para solucionar problemas técnicos al obtener un sistema de información, proponen la creación de modelos, flujos y estructuras mediante un top-down. Está orientada a procesos, es decir, se centra en especificar y descomponer la funcionalidad del sistema.

1.8.3.1 Métodos Estructurados

Un método estructurado es una forma sistemática de elaborar modelos de un sistema existente o de un sistema que tienen que construirse. Los métodos estructurados proporcionan un marco que soporta el desarrollo de modelos del sistema (Sommerville, 2005). Fueron desarrollados en la década de 1970 para soportar el análisis y diseño de software y, en la décadas de 1980 y 1990, evolucionaron para soportar el desarrollo orientado a objetos.

La mayoría de los métodos estructurados tienen su propio conjunto de modelos. Normalmente, definen un proceso que puede utilizarse para obtener los modelos y un conjunto de reglas y guías que se aplican a dichos modelos.

Los métodos estructurados normalmente son soportados por herramientas CASE, que incluyen la edición de modelos y la comprobación y generación de código.

La metodología estructurada cuenta con métodos que definen las actividades a realizar para obtener un sistema software, a continuación se da una breve descripción de los métodos.

1.8.3.2 Análisis estructurado

El análisis estructurado se concentra en especificar lo que quiere que haga el sistema o la aplicación. No se establece cómo se cumplirán los requerimientos o la forma en que implantará la aplicación. Más bien permite que las personas observen los elementos lógicos (lo que hará el sistema) separados de los componentes físicos (computadoras, terminales, sistemas de almacenamiento, etc.) después de esto se puede desarrollar un diseño físico eficiente para la situación donde será utilizado (Rivas, 2012).

En el análisis estructurado se utilizan varias herramientas:

- Diagramas de flujo de datos (DFD): representan la forma en la que los datos se mueven y se transforman. Incluye procesos, flujos de datos y almacenes de datos.Los procesos individuales se pueden a su vez descomponer en otros DFD de nivel superior.
- Especificaciones de procesos: describe los procesos definidos en el DFD cuando no se puede descomponer más. Puede hacerse en pseudocódigo, con tablas de decisión o en un lenguaje de programación.
- Diccionario de datos: son todas las definiciones de los elementos en el sistema (entidades externas, los flujos de datos y almacenes de datos).
- Diagramas entidad-relación: los elementos del modelo E/R se corresponden con almacenes de datos en el DFD. En este diagrama se muestran las relaciones entre dichos elementos

Para desarrollar una descripción del sistema por el método de análisis estructurado se sigue un proceso descendente (TOP-down). El modelo original se detalla en diagramas de bajo nivel que muestran características adicionales del sistema. Cada proceso puede desglosarse en diagramas de flujo de datos cada vez más detallados. Esta secuencia se repite hasta que se obtienen suficientes detalles que permiten al analista comprender en su totalidad la parte del sistema que se encuentra bajo investigación (Yourdon, 1993).

1.8.3.3 El diseño estructurado

Se enfoca en el desarrollo de especificaciones del software. La meta del diseño estructurado es crear programas formados por módulos independientes unos de otros desde el punto de vista funcional, es una técnica específica para el diseño de programas y no un método de diseño de comprensión. Esta técnica conduce a la especificación de módulos de programa que son funcionalmente independientes.

La herramienta fundamental del diseño estructurado es el diagrama estructurado, los cuales son de naturaleza gráfica y evitan cualquier referencia relacionada con el hardware o detalles físicos. Su finalidad no es mostrar la lógica de los programas. Los diagramas estructurados describen la interacción entre módulos independientes junto con los datos que

un módulo pasa a otro cuando interacciona con él. Estas especificaciones funcionales para los módulos se proporcionan a los programadores antes que dé comienzo la fase de escritura de código (Rivas, 2012).

1.8.3.4 La Implementación

Una vez que se definió qué funciones debe realizar el sistema (análisis) y cómo estarán organizados sus componentes (diseño), es el momento de pasar a la etapa de implementación. Para esta fase se deben seleccionar las herramientas adecuadas, un lenguaje de programación apropiado para el tipo de sistema elegido (Berzal, 2015).

Las herramientas a usar dependerán en gran parte del diseño y del entorno en el que el sistema deberá funcionar. A la hora de programar, el código no debe ser indescifrable. Además de las tareas de programación, en la fase de implementación debemos encargarnos de la adquisición de los recursos necesarios para que el sistema funcione (por ejemplo, las licencias de uso del sistema gestor de bases de datos que vayamos a utilizar) (Guevara, 2015).

CAPÍTULO II

Marco Metodológico

2.1 INTRODUCCION

El presente capítulo presenta el enfoque metodológico y describe en forma general la labor llevada a cabo para la obtención del presente trabajo de tesis, en él se muestran aspectos como: el tipo de investigación, las técnicas y procedimientos que fueron utilizados para llevar a cabo la investigación.

2.2 PLANTEAMIENTO DEL PROBLEMA

El desarrollo de software educativo ha pasado de ser concebido como un "presentador de información" a ser un elemento didáctico interactivo que se elabora a partir de la representación de conocimiento (Maldonado, 1997) y que facilita en el usuario su construcción gracias a la utilización de elementos que permiten solucionar problemas e impactar su estructura cognitiva.

De acuerdo con lo anterior el papel de las herramientas software en la educación se caracteriza por ser un elemento de apoyo al proceso de enseñanza-aprendizaje y elemento didáctico que diseña ambientes escolares basados en los requerimientos de los estudiantes. Lo anterior implica que en su realización debe tener en cuenta no solo aspectos técnicos sino también aspectos de aprendizaje. El docente entonces, pasa de ser un transmisor de información que genera en el estudiante indiferencia hacia los procesos de aprendizaje, a ser un creador de ambientes de aprendizaje, por lo tanto a centrar su tarea pedagógica en la caracterización de las necesidades de sus estudiantes y en la implementación de soluciones apoyado en las TICs.

En la carrera Ingeniería en Informática, de la Universidad Nacional de Catamarca, los alumnos de 2^{do} año de la cátedra Sistemas Operativos, estudian dentro de los contenidos de estas materias, la teoría de micro-operaciones mediante un computador básico hipotético, el cual se describe en el libro Arquitectura de Computadoras del autor M. Morris Mano (1993). Por no existir una máquina física, ni virtual, los alumnos no pueden realizar sus trabajos prácticos interactuando con una computadora. Por lo antes expuesto en este trabajo se diseñan e implementas las máquinas virtuales que emulen el computador básico hipotético utilizado para el estudio de micro-operaciones, para construir estas máquinas virtuales, se utilizan generadores de analizadores léxicos y sintácticos.

2.3 ANTECEDENTES

En la Universidad Nacional de Catamarca no se encontraron antecedentes sobre la utilización de un emulador de un computador básico como herramienta de aprendizaje, sin embargo en el Departamento de Computación de la Facultad de Ingeniería de la Universidad de Buenos Aires existe un trabajo que puede indicarse como antecedente, denominado “Un simulador de una máquina computadora como herramienta para la enseñanza de la arquitectura de computadoras”. En este trabajo se presenta una experiencia educativa específica para la Enseñanza de la Arquitectura de Computadoras, basada en el uso de un Programa Simulador del funcionamiento de una máquina computadora genérica que permite a los alumnos resolver problemas adaptándose a las limitaciones del lenguaje de máquina. Se caracteriza la situación didáctica, se fundamentan

los propósitos de la utilización del programa simulador, se describen sus características y se exponen los resultados obtenidos (Grossi et al, 2005).

Además, se puede mencionar otro trabajo del Departamento de Tecnología Electrónica de la Universidad de Sevilla llamado “Emulador del Computador Simple 2” (Gomez Gonzalez, 2008).

2.4 JUSTIFICACIÓN

El siglo XXI impone a cualquier proyecto educativo que pretenda verdaderamente desarrollar competencias necesarias para la vida moderna, como es la *alfabetización digital* y la reducción de la *brecha digital*, siendo un gran desafío, sobre todo en los países en vías de desarrollo, se debe saber y reconocer que las TICs son instrumentos potenciales para el crecimiento científico, cultural y económico de los pueblos.

A través de la utilización de las TIC, se pueden diseñar elementos didácticos que apoyan el proceso de enseñanza–aprendizaje de las diferentes temáticas, aumentando así la motivación del estudiante en la adquisición del conocimiento.

El integrar las TIC al proceso educativo sirve como apoyo a la docencia y proporciona al proceso de enseñanza – aprendizaje las herramientas necesarias en la cual el alumno no solo trabaja a su propio ritmo como una respuesta positiva a la enseñanza a través de la tecnología, sino que también pueda verificar ciertos conceptos que hasta ahora solo se veía en papel.

Por lo antes expuesto el presente trabajo proporcionará una herramienta para la enseñanza-aprendizaje para que los alumnos tengan la posibilidad de ejecutar sus programas, depurarlos, observar las micro-operaciones ejecutadas e inspeccionar los valores de los registros del procesador, interactuando con un computador, debido a que actualmente los trabajos prácticos se realizan en papel. Además, que los alumnos conozcan la arquitectura y el funcionamiento de la computadora para que puedan aprovecharla como herramienta de trabajo.

Esta posibilidad de que todos puedan aportar, escribir y generar conocimiento, ciertamente puede ser un riesgo para el rigor y exactitud académica de los contenidos generados, como muchos critican o alertan, pero precisamente eso constituye a su vez, una oportunidad para justificar ahora más que nunca, el desarrollo de potencialidades cognitivas para el análisis racional de la información. Resulta necesario tomar conciencia de esta situación y asumir una posición crítica, lo cual implica la necesidad de contar con mayor conocimiento y con las competencias de análisis necesarias y complementarias para superar el simple acceso acrítico a la información.

2.5 OBJETIVOS

2.5.1 Objetivo general

El objetivo general del presente trabajo es desarrollar las máquinas virtuales necesarias para emular el funcionamiento del computador básico hipotético utilizado para el estudio de micro-operaciones.

2.5.2 Objetivos específicos

- Proveer a los docentes, de la Cátedra Sistemas Operativos de la Universidad Nacional de Catamarca, de una herramienta que contribuya a la enseñanza de micro-operaciones y además facilite a los alumnos el aprendizaje de micro-operaciones.
- Brindar una herramienta tecnológica como soporte a la asignatura en el contexto del aula y fuera de ella.
- Ampliar las posibilidades de intervención del profesor así como de autoaprendizaje del alumno.
- Adquirir experiencia práctica en la programación a bajo nivel en lenguaje ensamblador de un computador básico.

2.6 DISEÑO METODOLÓGICO

Como metodología de trabajo se utilizó una metodología estructurada, a pesar de que existen muchas otras metodologías, esta es la que mejor se adecuó al desarrollo de la herramienta debido principalmente a que los requerimientos son estables. Para la programación y desarrollo de las pantallas del sistema se usó el Lenguaje C y Builder C++.

2.6.1 Tipo de Estudio o Investigación

Este trabajo de tesis se basó en una investigación aplicada de tipo exploratorio-descriptivo con aplicación de caso. Con el estudio descriptivo se buscó especificar las propiedades importantes del fenómeno que se investigó, para conseguir la familiarización con la herramienta a desarrollar, se efectuó una adecuada revisión de la literatura y análisis de antecedentes. Por otro lado la investigación, estuvo especialmente orientada a describir, analizar y aplicar la ingeniería de software para el desarrollo de la herramienta.

2.6.2 Técnicas e Instrumentos

Para la recolección de datos se utilizaron las siguientes técnicas e instrumentos:

- Análisis de contenidos: permitió realizar la sistematización bibliográfica mediante el instrumento de documento de anotaciones.
- Observación ordinaria y/o participante: permitió acumular y sistematizar información sobre el contenido dictado en la asignatura donde se utiliza la herramienta. El Instrumento usado fue el cuaderno de notas y guion de observación documental.
- Entrevista: se emplearán entrevistas semiestructuradas a los docentes debido a que se elaborarán interrogantes previamente y otras preguntas serán formuladas en el momento. Para esta técnica se utilizo el instrumento de la hoja de entrevista.

Para el desarrollo de la herramienta de software se analizaron y utilizaron los principios y criterios generales de la ingeniería de software.

2.6.3 Procedimiento

El procedimiento general que se llevó a cabo se describe a modo general, el cual cubrió las siguientes tareas o fases:

Análisis Exploratorio

Consistió en la búsqueda, recolección, lectura comprensiva y análisis de las fuentes de información (bibliografía, publicaciones, sitios web, entre otros) referidas al tema que trata el trabajo, para poder expresar las bases teóricas y conceptuales en las cuales se apoya el trabajo.

Desarrollo de la herramienta

El método de desarrollo utilizado fue el Método Estructurado donde se identifican las siguientes fases genéricas.

- **a.** Requerimientos o Requisitos.
- **b.** Análisis
- **c.** Diseño
- **d.** Desarrollo donde se transformaron las especificaciones en software.

Elaboración del Informe Final del Trabajo

Esta etapa involucró la redacción del presente informe de trabajo final, y la aplicación de diferentes técnicas para la documentación de la totalidad de los recursos obtenidos y/o utilizados a lo largo del trabajo.

Capítulo III

Desarrollo de la Herramienta

3.1 INTRODUCCIÓN

En este capítulo se aborda las actividades realizadas en el proceso de desarrollo de la herramienta software que permite emular un computador básico descripto por Morris Mano (Morris Mano, 1993).

El método de desarrollo de software utilizado fue el Método Estructurado donde se identificaron fases genéricas, las cuales se encuentran descriptas en el punto “2.6.3. Procedimiento”.

3.2 REQUERIMIENTOS O REQUISITOS.

En esta fase se llevaron a cabo las actividades para recolectar los requerimientos que debe satisfacer la herramienta. Se recabó información utilizando las siguientes técnicas:

- Entrevistas: se realizaron entrevistas semiestructuradas al Jefe de Trabajo Práctico.
- Analisis de documentación: se analizó la bibliografía del computador básico, apuntes de la teoría y trabajos prácticos de la Cátedra Sistemas Operativos.

Con la información recabada se generó la especificación de requerimientos de software (ERS) tomando como guía la norma IEEE 830 (1998).

3.2.1 Propósito

El propósito de la ERS fue definir de manera clara y precisa todos los requerimientos funcionales y restricciones que debe reunir el ECB para apoyar el proceso enseñanza-aprendizaje de micro-operaciones de un procesador.

3.2.2 Alcance o Ámbito de la herramienta

El ECB se modela para brindar una herramienta que permita la enseñanza-aprendizaje de micro-operaciones, para lo cual proporcionará un compilador y un intérprete para la ejecución, depuración de programas que utilizan las mismas.

La herramienta no permite:

- La modificación en tiempo de ejecución, por parte de los usuarios de posiciones de memoria.
- El uso de etiquetas para representar posiciones de memoria.

3.2.3 Acrónimos

Se detalla en la Tabla 3-1 los acrónimos que se utilizan en la especificación de Requisistos.

Acrónimo	Descripción
ERS	Especificación de Requerimientos de software
IEEE	The Institute of Electrical and Electronics Engineers
ECB	Emulador de Computador Básico
RAM	Memoria de Acceso Aleatorio

Tabla 3-1 Acrónimos

3.2.4 Perspectiva del Producto

El ECB, en esta versión, no interactuará con ninguna otra aplicación.

3.2.5 Funciones de la herramienta

Las principales funciones que realiza el ECB son:

- Edición de código fuente.
- Ensamblar el código assembler.
- Ejecutar código máquina.
- Depurar código máquina.

3.2.6 Restricciones

Dado que la herramienta implementa la mayoría de los procesos que actualmente se realizan en forma manual, es de esperar que futuros cambios en los modos de trabajo o en las políticas, ejerzan un impacto sobre la actual herramienta y entorno.

3.2.7 Requerimientos de Hardware/Software

En cuanto a las restricciones Hardware/Software, la herramienta funciona bajo el paradigma escritorio.

El entorno de programación es Borland Builder C++ 6.

3.2.8 Requerimientos Funcionales

- El alumno necesita crear un nuevo código fuente en assembler.
- El alumno necesita modificar el código fuente en assembler.
- El alumno necesita imprimir el código fuente.
- El alumno necesita localizar y/o reemplazar texto en el código fuente.
- El alumno requiere colocar marcadores (Bookmark) en el código fuente para volver a una posición determinada.
- El alumno requiere ejecutar el código fuente.
- El alumno requiere depurar el código fuente.

- El alumno requiere compilar (Ensamblar) el código fuente.
- El alumno requiere colocar y/o eliminar puntos de interrupción (Breakpoint) en el código fuente.
- El alumno necesita interrumpir la ejecución del código objeto.
- El alumno necesita continuar con la ejecución del código objeto, luego de haber interrumpido la ejecución.
- El alumno necesita reestablecer la ejecución del código objeto (al reestablecer la ejecución, se vuelve al estado inicial la memoria, registros y banderas del procesador).
- El alumno necesita inspeccionar posiciones de memoria.
- El alumno necesita establecer el punto de ejecución.
- El alumno necesita localizar el punto de ejecución.
- El alumno necesita modificar los valores de los registros de entrada (INPR) y salida (OUTR).
- El alumno necesita modificar loa valores de las banderas de habilitación de interrupción (IEN), de entrada (FGI) y salida (FGO).

3.2.9 Requerimientos No Funcionales

- La herramienta debe correr en sistema Operativo Windows XP o superior.
- La interfaz del usuario debe ser simple y clara.

3.3 ANÁLISIS

En este punto se presenta la actividad de análisis realizada, donde se tuvo en cuenta los componentes que forman el ECB:

- Compilador Assembler
- Interprete de código de máquina
- Emulador

3.3.1 Compilador Assembler

Se realizó análisis en base a las instrucciones del lenguaje assembler descriptas en el punto “1.2 Organización y Funcionamiento de un Computador Básico”.

También se analizó las herramientas que se usaron para la implementación del compilador de assembler.

Para el desarrollo del **Analizador lexico** que usa el emulador se utilizó FLEX debido a que esta herramienta es equivalente a LEX y funciona sobre Windows y la mayoria de los posibles usuarios de la herramienta utilizan el sistema operativo Windows.

Para generar el **Analizador sintáctico** se utilizó la herramienta BISON, la cual es compatible con el generador de analizadores sintácticos YACC.

En la figura 3-1 se muestra la estructura de generación del compilador que usa el ECB según las herramientas empleadas.

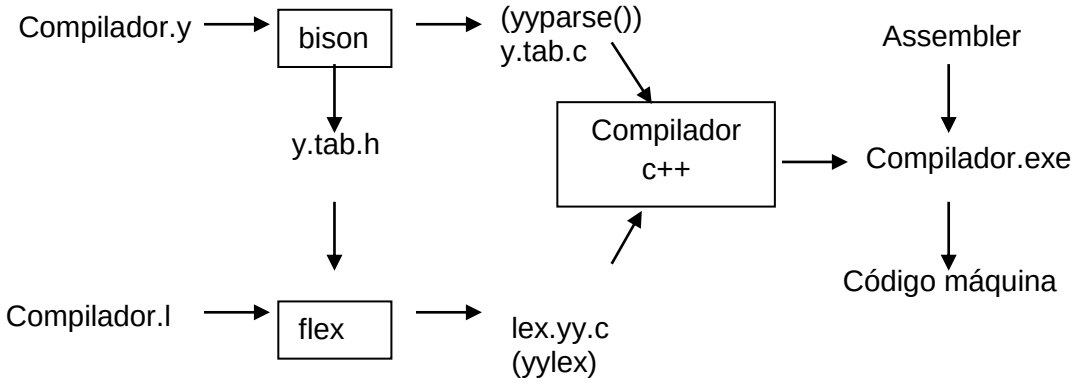

Figura 3-1 Construcción de un compilador con Flex /Bison

La Figura 3-1 ilustra la convencion de nombres de archivos utilizados por flex y bison. En primer lugar, se especifico todas las reglas de coincidencia de patrones para flex (Compilador.l) y las reglas gramaticales para bison (Compilador.y).

Los comandos para crear el compilador, Compilador.exe, se enumeran a continuación:

```
bison –d Compilador.y                  # crea y.tab.h, y.tab.c
flexCompilador.l                       # crea lex.yy.c
cc lex.yy.c y.tab.c –o Compilador.exe  # compila/link
```

Bison lee las descripciones gramaticales en Compilador.y y genera un analizador sintáctico, la función yyparse, en el archivo y.tab.c. Incluido en el archivo Compilador.y están las declaraciones de tokens. La opción -d hace que Bison genere definiciones para los tokens y las coloca en el archivo y.tab.h.

Flex lee las descripciones de patrónes en Compilador.l, incluye el archivo heder y.tab.h, y genera un analizador léxico, la función yylex, en archivo lex.yy.c.

Por último, el analizador léxico y sintáctico son compilados y enlazados entre sí para formar el ejecutable, Compilador.exe. Desde main, llamamos a yyparse para ejecutar el compilador. La función yyparse automáticamente llama a yylex para obtener cada token.

3.3.1.1 Analizador Léxico (Patrones)

A continuación se describen las expresiones regulares para el ensamblador, instrucciones y otros patrones usados en los programas.

- **Expresiones Regulares para ensamblador**

Expresion Regular	Descripcion
data	Area de memoria destinada para los datos
program	Area de instrucciones del programa a ejecutar

Tabla 3-2 Expresiones Regulares para el ensamblador.

- **Expresiones Regulares de instrucciones**

  Estas expresiones son las instrucciones assembler del computador basico de Morris (1993).

Expresion Regular	Descripcion
cla	Aclara registro AC
cle	Aclara bandera E
cma	Complementa registro AC
cme	Complementa E
cir	Circula a la derecha E y AC
cil	Circula a la izquierda E y AC
inc	Incrementa AC
spa	Salta si AC es positivo
sna	Salta si AC es negativo
sza	Salta si AC es cero
sze	Salta si E es cero
hlt	Para el computador
inp	Entre el carácter a AC
out	Saque el carácter de AC
ski	Salte en la bandera de entrada
sko	Salte en la bandera de salida
ion	Encendido de interrupción
iof	Apagado de interrupción
and	AND la palabra de memoria a AC
add	SUMA de la palabra de memoria a AC
lda	Carga AC a partir de la memoria
sta	Almacena AC en la memoria
bun	Ramifica incondicionalmente
bsa	Ramifica y mantiene la dirección de retorno
isz	Incrementa y salta si es cero

Tabla 3-3 Expresiones Regulares de instrucciones

- **Reconocimiento de otros patrones**

Expresion Regular	Descripcion
&[[:xdigit:]]+	Valores hexadecimales
[():;]	Caracteres [():;]
[\t]+	Uno ó mas caracteres de tabulación
\n	Salto de línea
'[^\n]*	comentarios
.	Cualquier carácter excepto nueva línea

Tabla 3-4 Reconocimiento de otros patrones

3.3.1.2 Analizador Sintáctico (Gramática)

A continuación se muestra la gramática desarrollada para el analizador sintáctico realizado.

```
source_code
-> DATA_SECT memory_init_list PROGRAM_SECT code_list
  | DATA_SECT PROGRAM_SECT code_list

memory_init_list
  -> memory_init_list memory_init
  | memory_init

memory_init
  -> address_data_list ':' data_list ';'

address_data_list
  -> HEXNUM

data_list
  -> data_list HEXNUM
  | HEXNUM

code_list
  -> code_list code
  | code

code
  -> address_instruction_list ':' instruction_list

address_instruction_list
  -> HEXNUM

instruction_list
  -> instruction_list instruction_dir_mem_ref
  | instruction_list instruction_indir_mem_ref
  | instruction_list instruction_reg_io
  | instruction_dir_mem_ref
  | instruction_indir_mem_ref
  | instruction_reg_io

instruction_dir_mem_ref
  -> label_inst_mem_ref HEXNUM ';'

instruction_indir_mem_ref
  -> label_inst_mem_ref '(' HEXNUM ')' ';'
```

```
instruction_reg_io
  -> label_inst_reg_io ';'

label_inst_mem_ref
  -> AND
  | ADD
  | LDA
  | STA
  | BUN
  | BSA
  | ISZ

label_inst_reg_io
  -> CLA
  | CLE
  | CMA
  | CME
  | CIR
  | CIL
  | INC
  | SPA
  | SNA
  | SZA
  | SZE
  | HLT
  | INP
  | OUT
  | SKI
  | SKO
  | ION
  | IOF
```

3.3.2 Interprete de Código Maquina

La funcionalidad del Interprete de Código de Maquina se analizó en la descripción del computador básico del punto “1.2 Organización y Funcionamiento de un Computador Básico”.

3.3.3 Análisis del Emulador

Para el desarrollo del emulador se utilizó la metodología de Analisis Estructurado de Edward Yordon (1993). Este autor propone el desarrollo de un Modelo Esencial el cual presenta la funcionalidad que el sistema debe hacer para satisfacer los requerimientos del usuario.

El *modelo esencial* consiste de dos componentes principales:

- El Modelo Ambiental
- El Modelo de Comportamiento: para modelar la funcionalidad del sistema se usó Diagrama de Flujo de Datos (DFD), diccionario de datos y especificación de procesos.

3.3.3.1 Modelo Ambiental

Este modelo define la frontera entre el sistema y el contexto; es decir, el ambiente o entorno en el cual existe el sistema.

El *modelo Ambiental* consiste de:

- Propósitos del Sistema
- Diagrama de Contexto
- Lista de Acontecimientos o Eventos

Propósitos del Sistema

Maquina virtual para emular el computador básico utilizado para el aprendizaje de micro-operaciones.

Diagrama de Contexto

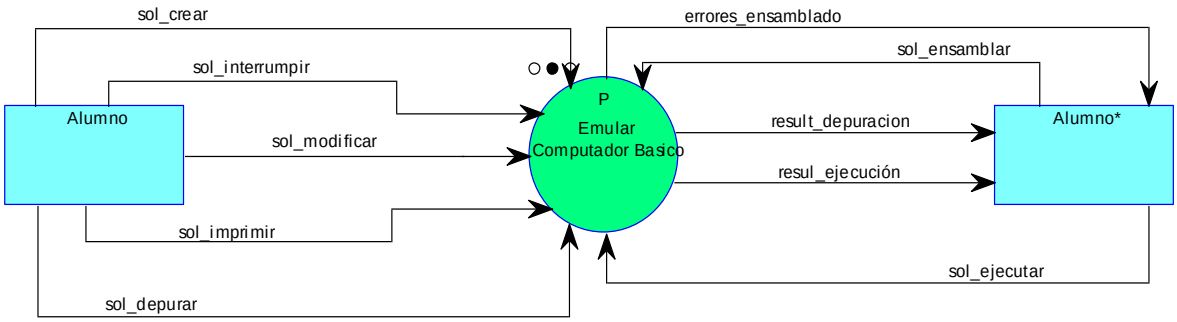

Figura 3-2 Diagrama de Contexto

Lista de Acontecimientos o Eventos

1) El alumno necesita editar código fuente en assembler.
2) El alumno necesita imprimir el código fuente.
3) El alumno requiere compilar (Ensamblar) el código fuente.
4) El alumno requiere ejecutar el código fuente.
5) El alumno requiere depurar el código fuente.

3.3.3.2 Modelo de Comportamiento

Describe el comportamiento que se requiere del sistema para que interactúe de manera exitosa con el ambiente.

El modelo de comportamiento consiste de:

- DFD de nivel de sistema: muestra las funciones que debe realizar el sistema.
- Diccionario de datos: herramienta textual, que explica o apoya a los DFD, en la que se describen los flujos, los elementos de datos, los almacenes y las entidades externas.
- Especificación de procesos: herramienta textual que explica los procesos.

Diagrama de Flujos de Datos

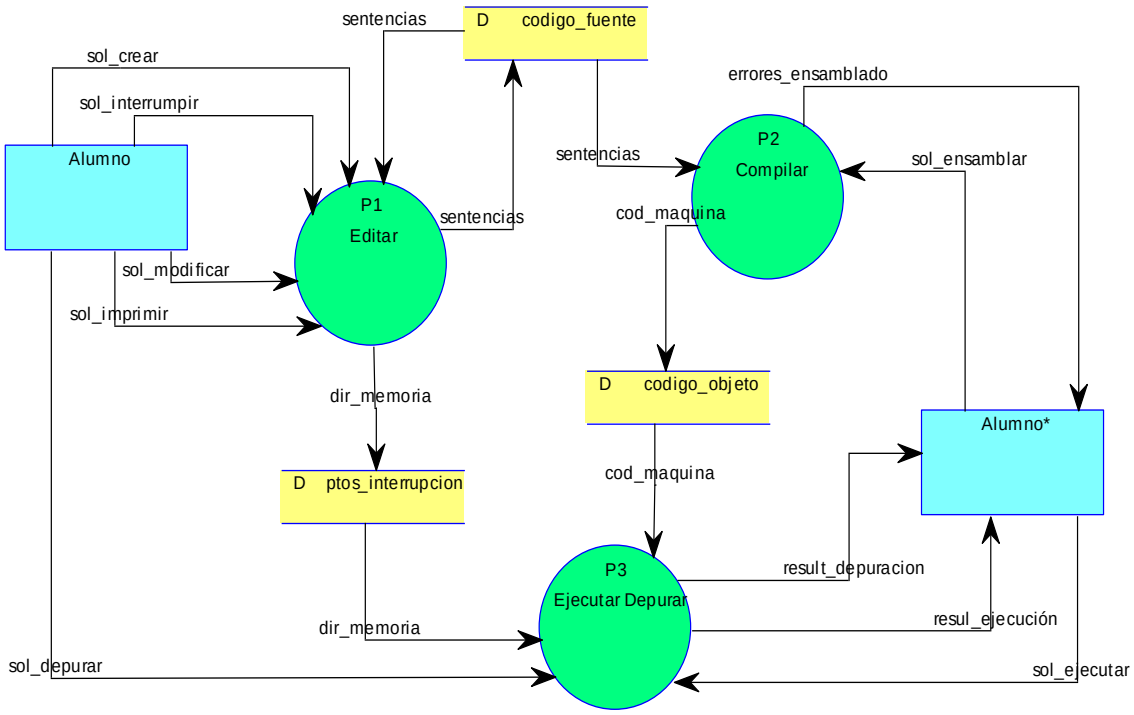

Figura 3-3 Diagrama de Flujos de Datos

Diccionario de Datos

El diccionario de datos describe los elementos del DFD siguientes:

- Entidades externas
- Flujo de datos
- Estructura de datos
- Almancenes de datos
- Elementos de datos

Entidades Externas

Entidad Externa	Descripción
Alumno	Persona que asiste a la Universidad

Tabla 3-5 Entidad Externa.

Flujos de Datos

Flujo de Dato	Especificación
Cod_maquina	{byte}
dir_memoria	** valor de dirección de memoria representada en hexadecimal
Errores ensamblado	{Nro_linea +columna+ Descrip_error}
resul_depuracion	** Alias de resul_ejecucion
resul_ejecucion	dir_memoria + Cod_maquina + cod_assembler + registros + banderas + vuelco_memoria + micro_op + ({inspec_memoria})
Sentencias	{Etiqueta | Comentario | Instruccion}
Sol_crear	Nom_programa + sentencias
Sol_depurar	** depura el programa abierto y ensamblado*
Sol_ejecutar	** ejecutar el programa abierto y ensamblado*
Sol_ensamblar	Nom_programa+ (puntos_interrupcion)
Sol_imprimir	Nom_programa
Sol_interrumpir	Nro_linea
Sol_modificar	** igual datos que Cod_fuente

Tabla 3-6 Flujos de Datos.

Estructuras de Datos

Estructuras de Datos	Especificación
banderas	Nom_bandera + valor_bandera
cod_assembler	{ instrucción_assembler +(dir_memoria) }
instrucción	Dir_memoria + instrucción_assembler + (Dir_memoria) + ";"
registros	Nom_reg + valor_reg
vuelco_memoria	Dir_memoria + ":" + {palabra_memoria} + {represent_ASCII}
micro_op	Ciclo + {función_ctrol + micro_op}
inspec_memoria	Dir_memoria + palabra_memoria
puntos_interrupcion	{Dir_memoria}

Tabla 3-7 Estructuras de Datos.

Elementos de Datos

Elemento	Descripción	Tipo dato	Longitud		Observación
byte	Números hexadecimales	hexadecimal	2		
Ciclo	Descripción del ciclo del procesador [Ciclo Fetch | Ciclo Indirecto | Ciclo Ejecución | Ciclo Interrupción]	String	18		
columna	Numero de columna	entero	2		
Comentario	Son líneas ignoradas por el ensamblador o compilador de la herramienta				Comienzan con “ ' ” ' Main program '
Descrip_error	*Descripción del error producido* [Se esperaba una dirección | se esperaba una instrucción | Se esperaba ';', ….]	String	200		
dir_memoria	Dirección memoria en hexadecimal	Hexadecimal	3		
Etiqueta	Palabra que indica una sección del programa				DATA: indica que el bloque que sigue son datos PROGRAM: indica que las líneas siguientes son instrucción
función_ctrol	Función de control de la micro-operación	String			
instrucción_assembler	Etiqueta que identifica una instruccion	char	3		ej. LDA, ADD. BUM
micro_op	Micro-operación a ser ejecutada y el valor que tienen los registros	String			
Nom_bandera	Nombre de la bandera	Char	3		
Nom_programa	Nombre del archivo del programa	String	100		
Nom_reg	Nombre del registro	char	4		
Nro_linea	Numero de línea	Entero	2		
palabra_memoria	Valor en hexadecimal	Hexadecimal	4		
represent_ASCII	Representación ASCII de un byte de memoria	Char	1		
valor_bandera	Valor en hexadecimal	Hexadecimal	1		
valor_reg	Valor en hexadecimal	Hexadecimal	4		

Tabla 3-8 Elementos de Datos.

Almacenes de Datos

Almacén	Descripción
Código Fuente	{ Sentencias }
Código Objeto	{cod maquina}
Ptos_Interrupcion	{ Dir_memoria }

Tabla 3-9 Almacenes de Datos.

Especificaciones de Procesos

A continuación se presenta la especificación de procesos, se utilizó la herramienta textual de "lenguaje estructurado".

Proceso P1: Editar

```
Entrada: sol_crear, sol_modificar, sol_interrumpir, sol_imprimir
Salida: sentencias, dir_memoria, código
INICIO
        HACER CASO
        CASO existe sol_crear
                crear archivo
                guardar sentencias CODIGO_FUENTE
        CASO existe sol_modificar
                obtener sentencias CODIGO_FUENTE
                editar codigo
                guardar sentencias CODIGO_FUENTE
        CASO existe sol_interrumpir
                agregar dir_memoria PTOS_INTERRUPCION
        CASO existe sol_imprimir
                imprimir codigo
        FIN CASO
TERMINA
```

ProcesoP2: Compilar

```
Entrada: sol_ensamblar, sentencias
Salida : cod_maquina, errores_ensamblado
INICIO
        SI existe sol_ensamblar
                OBTEBER sentencias CODIGO_FUENTE
                HACER compilar
                SI existen errores
                        mostrar errores_ensamblado
                SI NO
                        guardar cod_maquina CODIGO_OBJETO
                FIN SI
        FIN SI
TERMINA
```

Proceso P3: Ejecutar Depurar

Entrada: cod_maquina, ptos_interrupcion
Salida : resul_ejecucion, resul_depuracion

```
INICIO
        OBTENER cod_maquina
        OBTENER ptos_interrupcion

        SI existe sol_ejecutar
                HACER MIENTRAS existen elementos CODIGO_OBJETO
                        SI direccion instruccion existe en PTOS_INTERRUPCION
                                detener ejecucion
                        SI NO
                                ejecutar instruccion
                                mostrar resul_ejecucion
                        FIN SI
                FIN HACER
        FIN SI

        SI existe sol_depurar
                HACER MIENTRAS existen elementos CODIGO_OBJETO
                        SI evento continuar ejecucion
                                ejecutar instruccion
                                mostrar resul_ejecucion
                        FIN SI
                FIN HACER
        FIN SI
TERMINA
```

3.4 DISEÑO

En este punto se presenta lo realizado en la actividad de diseño.

3.4.1 Intérprete Código Maquina

En la Figura 3-4 se muestra el esquema del intérprete de código de máquina realizado.

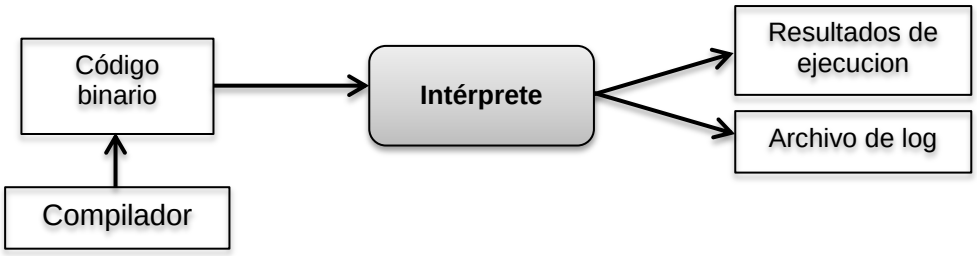

Figura 3-4 Intérprete Código Maquina.

El código binario es generado por el compilador y sirve como entrada al Intérprete. El intérprete es una funcion o modulo que ejecuta un programa binario. Los resultados de la ejecución son mostrados por pantalla y loggeados en un archivo.

Por pantalla se muestra los ciclos, las micro-operaciones, los valores de registros y banderas.
En el archivo se registran los ciclos y las micro-operaciones necesarias para la ejecución de las instrucciones. Ver definición del archivo en el punto “3.4.2.3 Definición de archivos”.

3.4.2 Emulador

En este punto se muestran los artefactos realizados durante la actividad de diseño del ECB, los cuales son:

- Diagrama de estructura y descripción de los flujos de datos y módulos.
- Interfaces gráficas.
- Definición de archivos.

3.4.2.1 Diagrama de estructura

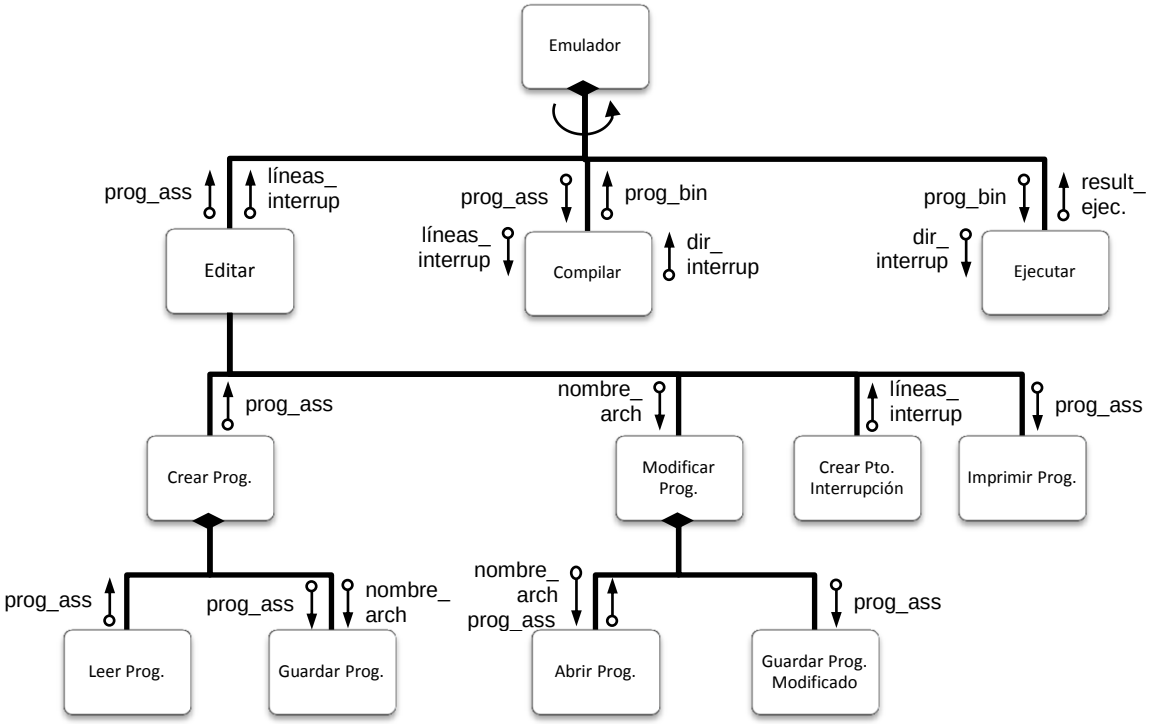

Figura 3-5 Diagramas de estructura.

Descripción de los flujos de datos

A continuación se describen los flujos de datos del diagrama de estructura de la Figura 3-5.

Flujo de Datos	Especificación
prog_ass	Programa en assembler que ingresa el usuario para ser compilado y ejecutado
líneas_interrup	Líneas de programa correspondientes a las marcas de interrupción definidas por el usuario
nombre_arch.	Nombre de archivo del programa assembler
prog_bin	Programa binario a ser ejecutado
dir_interrup	Direcciones de memoria correspondientes a las marcas de interrupción
result_ejec.	Resultados de la ejecución del programa binario

Tabla 3-10 Flujo de Datos.

Especificación de Módulos

A continuación se especifican los módulos que forman el diagrama de estructura de la Figura 3-5.

Nombre de Módulo	Emulador
Entrada	Opción ingresada por el alumno
Salida	Resultados de la ejecución
Descripción	Comenzar Según la selección de la opcion ingresada Hacer: Caso 1:Llamar Módulo “Editar” Caso 2:Llamar Módulo “Compilar” Caso 3: Llamar Módulo “Ejecutar” Fin

Tabla 3-11 Módulo Emulador.

Nombre de Módulo	Editar
Entrada	ninguna
Salida	prog_ass, lineas_interrup
Descripción	Comenzar Según la selección de la opcion ingresada Hacer: Caso 1:Llamar Módulo “Crear Prog.” Caso 2: Llamar Módulo “Modificar Prog.” Caso 3: Llamar Módulo “Crear Pto. Interrupción” Caso 4: Llamar Módulo “Imprimir Prog.” Fin

Tabla 3-12 Módulo Editar.

Nombre de Módulo	Crear Prog.
Entrada	ninguna
Salida	prog_ass
Descripción	Comenzar Llamar Módulo “Leer Prog.” Llamar Módulo “Guardar Prog.” Fin

Tabla 3-13 Módulo Crear Prog..

Nombre de Módulo	Leer Prog.
Entrada	Nombre archivo
Salida	prog_ass
Descripción	Comienza Ingresa por pantalla programa assembler prog_ass Fin.

Tabla 3-14 Módulo Leer Prog.

Nombre de Módulo	Guardar Prog.
Entrada	prog_ass, nombre_arch
Salida	ninguna
Descripción	Comienzo Ingresa por pantalla nombre de archivo Grabar programa assembler prog_ass Fin

Tabla 3-15 Módulo Guardar Prog.

Nombre de Módulo	Modificar Prog.
Entrada	nombre_arch
Salida	
Descripción	Comenzar Llamar Módulo “Abrir Prog.” Ingresa por pantalla modificaciones al programa assembler prog_ass Llamar Módulo “Guardar Prog.” Fin

Tabla 3-16 Módulo Modificar Prog.

Nombre de Módulo	Abrir Prog.
Entrada	nombre_arch (archivo de texto con extension ".ecb")
Salida	prog_ass
Descripción	Comienza Leer prog_ass que se corresponde con nombre archivo "nombre_arch". Fin.

Tabla 3-17 Módulo Abrir Prog.

Nombre de Módulo	Guardar Prog.Modificado
Entrada	prog_ass
Salida	
Descripción	Comienzo Grabar programa assembler prog_ass Fin

Tabla 3-18 Módulo Guardar Prog. Modificado.

Nombre de Módulo	Crear Pto. Interrupción
Entrada	
Salida	lineas_interrup
Descripción	Comienzo Ingresa por pantalla marcas de interrupción lineas_interrup en las lineas de programa Resaltar con rojo lineas marcadas Agregar numero de linea en lineas_interrup Fin

Tabla 3-19 Módulo Crear Pto. Interrupción.

Nombre de Módulo	Imprimir Prog.
Entrada	prog_ass
Salida	
Descripción	Comienzo Ingresar por pantalla intervalo de impresion Ingresar por pantalla número de copias Ingresar por pantalla configuración de la impresora Imprimir programa assembler prog_ass Fin

Tabla 3-20 Módulo Imprimir Prog.

Nombre de Módulo	Compilar
Entrada	prog_ass, lineas_interrup
Salida	prog_bin, dir_interrup
Descripción	Comienzo // Se utiliza la funcion generada por la herramienta Bison para compilar Compilar programa assembler prog_ass Fin

Tabla 3-21 Módulo Compilar.

Nombre de Módulo	Ejecutar
Entrada	prog_bin, dir_interrup
Salida	result_ejec.
Descripción	Comienzo Ejecutar programa binario prog_bin Generar resultados result_ejec.(micro-operaciones, registros procesador, banderas procesador, vuelco de memoria e inspección de memoria) Fin

Tabla 3-22 Módulo Ejecutar.

3.4.2.2 Diseño de Interfaces gráficas

El emulador esta constituido por un conjunto de ventanas, las cuales se describen brevemente a continuación.

Ventana de edición del código assembler

Esta ventana proporciona un editor de código fuente. Este es usado para mostrar y editar código assembler. Se provee funciones de edición, búsqueda y reemplazo de texto, establecer y borrar marcadores de texto, ejecutar y depurar el código assembler, establecer y borrar puntos de interrupción, opciones de tipo y color de fuente y ayuda del sistema.

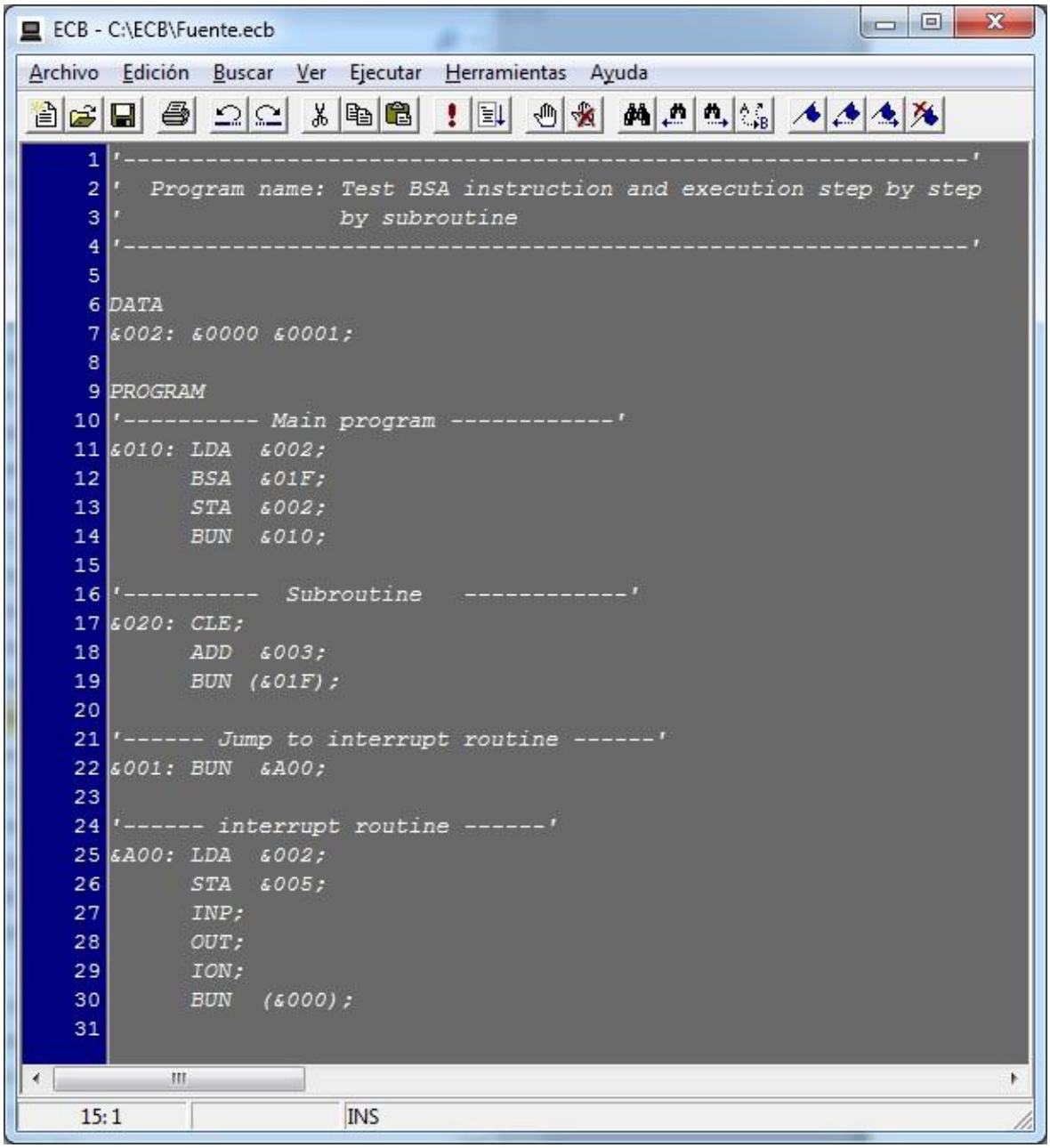

Figura 3-6 Ventana de edición del código assembler.

Cuando se ensambla y existen errores se muestra el cuadro de dialogo de la Figura 3.7 donde indica el archivo ensamblado, la cantidad de líneas que posee y la cantidad de errores.

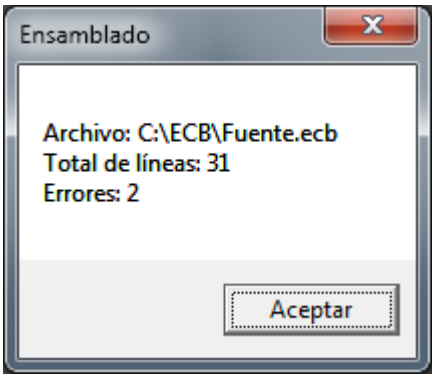

Figura 3-7 Cuadro de dialogo del comando ensamblar

En caso de producirse **errores de compilación**, lo mismos se muestran en el panel inferior al ejecutar los comandos ejecutar ó depurar.

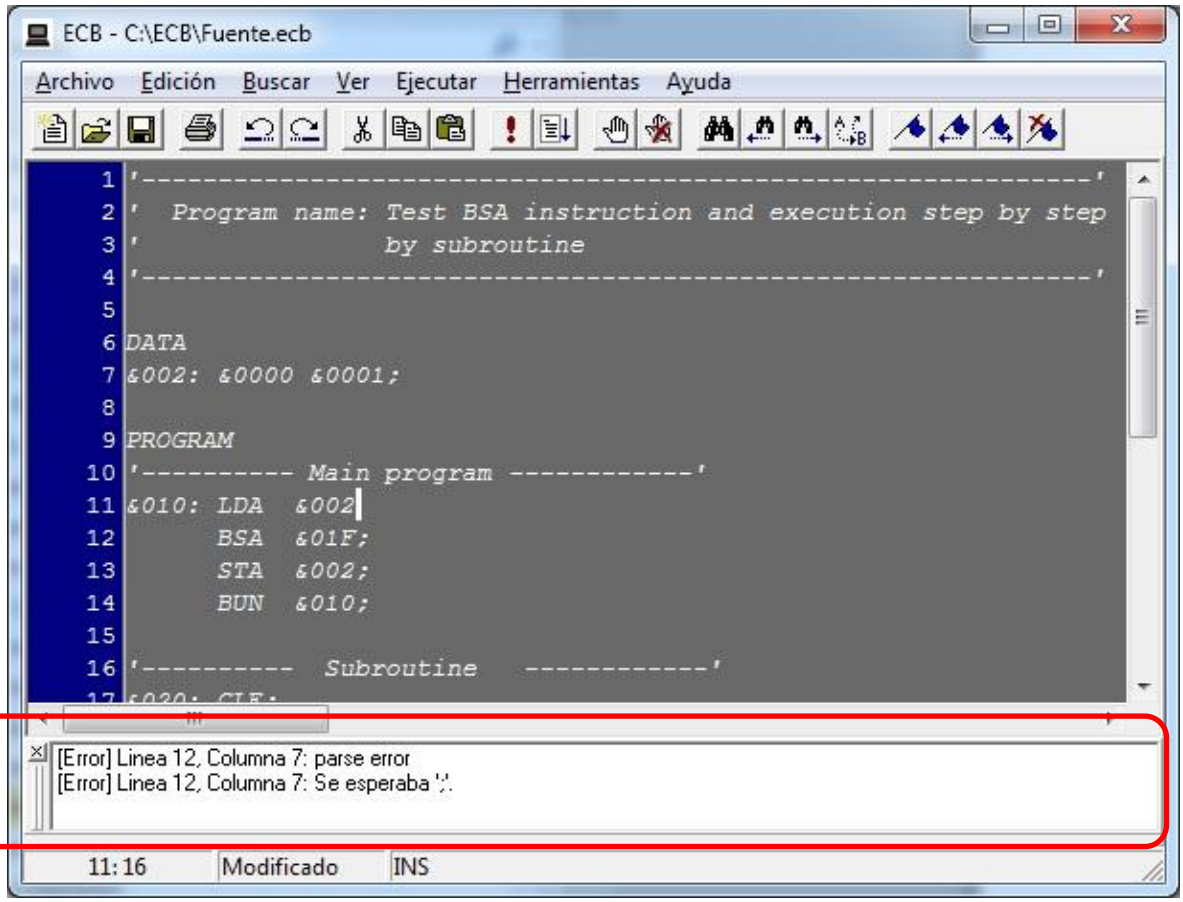

Figura 3-8 Ventana de edición del código assembler con errores de compilación.

Ventana del intérprete

Esta ventana esta compuesta por el panel de código máquina y assembler, registros del procesador, banderas del procesador, pestañas de vuelco de memoria y micro-operaciones e inspección de posiciones de memoria.

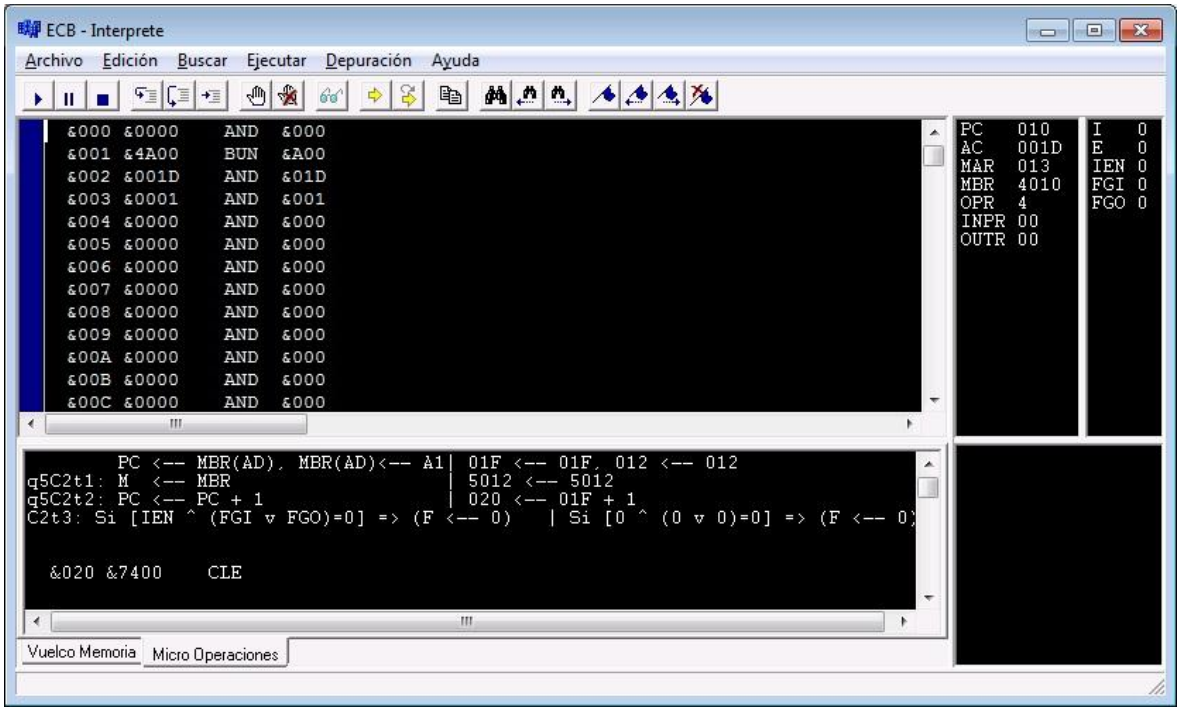

Figura 3-9 Ventana del intérprete con micro-operaciones.

La pestaña vuelco de memoria, que se encuentra en la parte inferior de la ventana, muestra el contenido de la memoria en formato hexadecimal y ASCII.

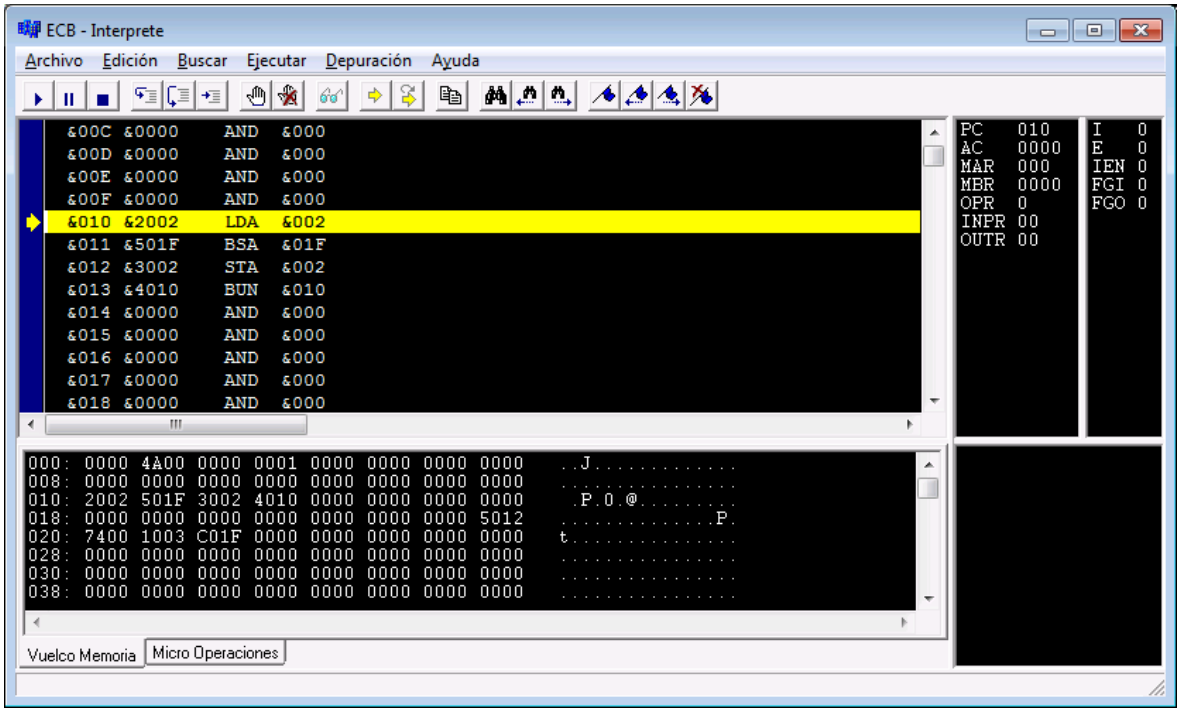

Figura 3-10 Ventana del intérprete con vuelco de memoria.

3.4.2.3 Definición de archivos

El ECB utiliza dos archivos:

- Archivo de código assembler.
- Archivo de loggeo de la ejecución.

Archivo de código assembler

El código assembler se divide en dos secciones, una sección para datos "DATA", y otra para programa "PROGRAM".
La sección de **DATA** está compuesta por una ó más lista de inicialización de memoria. Cada lista de inicialización de memoria está compuesta de una dirección de memoria seguida de dos puntos, una o más palabras de memoria y un punto y coma al final de la lista.

La sección de **PROGRAM** está compuesta por una ó más lista de código. Cada lista de código está compuesta de una dirección de memoria seguida de dos puntos, una o más instrucciones y un punto y coma al final de la lista.

Tanto las direcciones de memoria como su contenido, se deben expresar en hexadecimal, anteponiendo el símbolo ampersand "&" antes del valor, por ejemplo, "&FFF" para una dirección de memoria y "&FFFF" para una palabra de memoria.

Las direcciones de memoria encerradas entre parentisis, indican un direccionamiento indirecto. Las líneas de comentarios deben comenzar con el símbolo " ' ".

La Figura 3-11 muestra un ejemplo de archivo de código assembler.

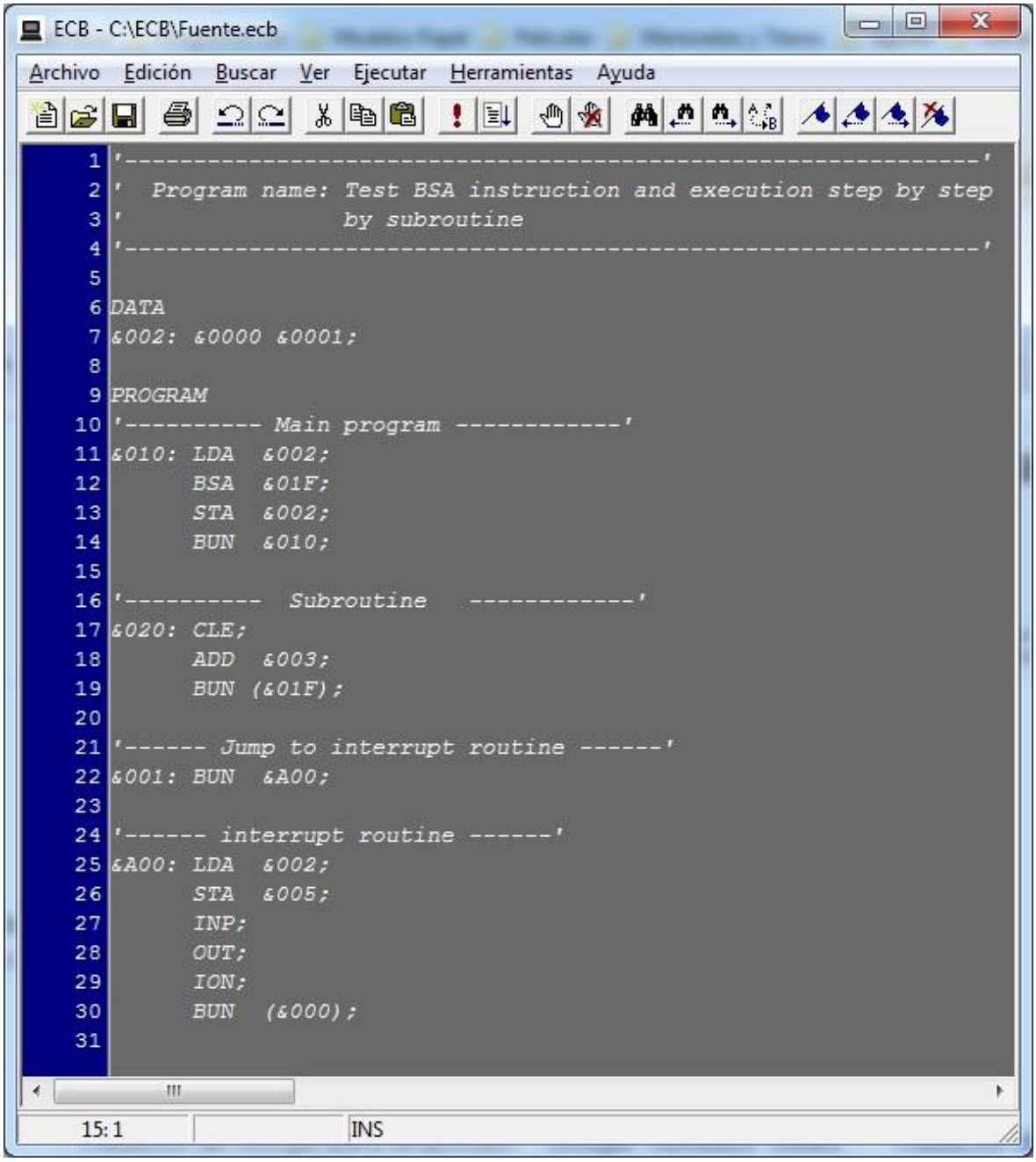

Figura 3-11 Archivo de código assembler.

Archivo LOG

En este archivo se almacena el nombre del programa, fecha y hora de ejecución. Además, para cada instrucción, se guarda los ciclos y micro-operaciones utilizados para la ejecución. Para cada micro-operación, se muestra los valores de los registros y banderas asociadas.

La Figura 3-12 muestra un ejemplo de archivo log.

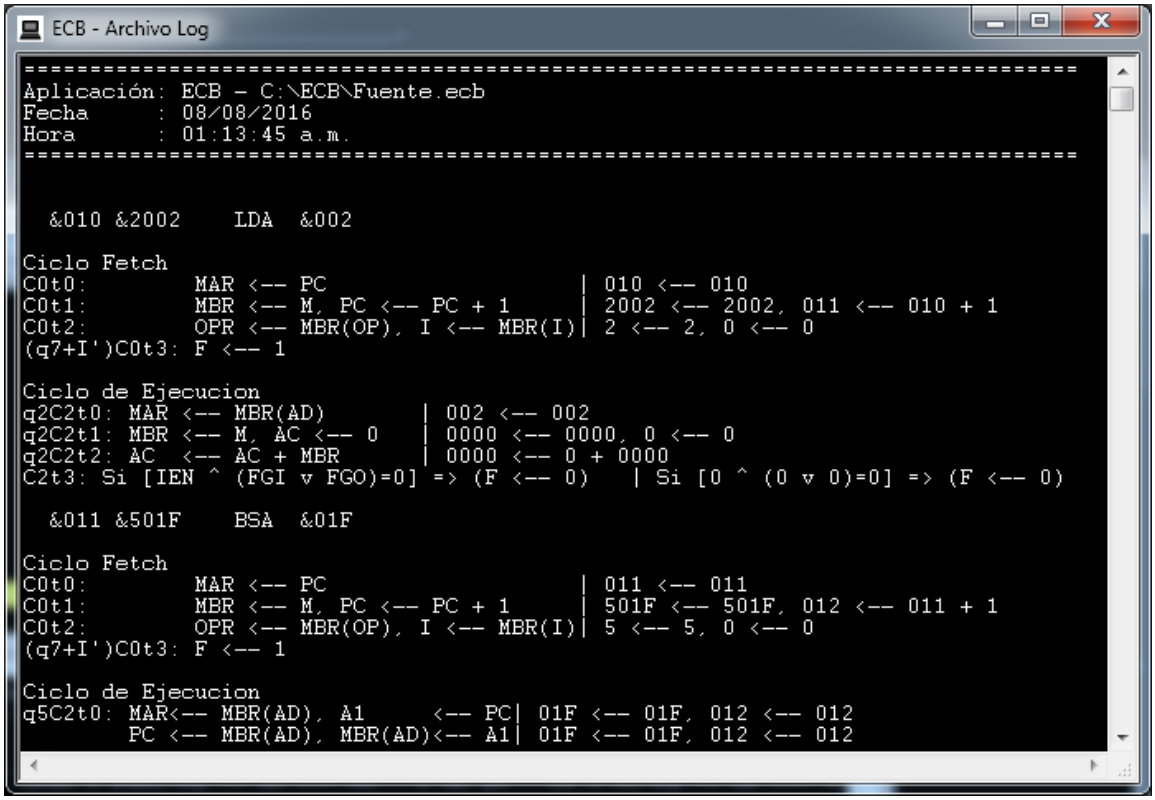

Figura 3-12 Archivo LOG.

3.5 DESARROLLO DE LA HERRAMIENTA

En este punto se muestra lo realizado en la actividad de desarrollo de la herramienta.

3.5.1 Compilador

Para generar el compilador e interprete se utilizaron las herramientas Bison para el analizador Sintactico y Flex para el analizador lexico.

3.5.1.1 Analizador lexico

Este codigo esta escrito según la especificación de Flex para reconocer los patrones o expresiones regulares.

```
%{
#include "Global.h"
#include "ytab~1.h"
#include <string.h>

externvoid yyerror(char*);

char buffer[256];
unsignedint Column =1;
staticunsignedint ScanColumn =1;
staticvoid GetColumn(unsignedint*Column,unsignedint*ScanColumn );
%}
%option noyywrap
```

```
%option never-interactive
%option yylineno

%%

data{ GetColumn(&Column,&ScanColumn );return DATA_SECT;}
program{ GetColumn(&Column,&ScanColumn );return PROGRAM_SECT;}
cla      { GetColumn(&Column,&ScanColumn );return CLA;}
cle      { GetColumn(&Column,&ScanColumn );return CLE;}
cma      { GetColumn(&Column,&ScanColumn );return CMA;}
cme      { GetColumn(&Column,&ScanColumn );return CME;}
cir      { GetColumn(&Column,&ScanColumn );return CIR;}
cil      { GetColumn(&Column,&ScanColumn );return CIL;}
inc      { GetColumn(&Column,&ScanColumn );return INC;}
spa      { GetColumn(&Column,&ScanColumn );return SPA;}
sna      { GetColumn(&Column,&ScanColumn );return SNA;}
sza      { GetColumn(&Column,&ScanColumn );return SZA;}
sze      { GetColumn(&Column,&ScanColumn );return SZE;}
hlt      { GetColumn(&Column,&ScanColumn );return HLT;}
inp      { GetColumn(&Column,&ScanColumn );return INP;}
out      { GetColumn(&Column,&ScanColumn );return OUT;}
ski      { GetColumn(&Column,&ScanColumn );return SKI;}
sko      { GetColumn(&Column,&ScanColumn );return SKO;}
ion      { GetColumn(&Column,&ScanColumn );return ION;}
iof      { GetColumn(&Column,&ScanColumn );return IOF;}
and      { GetColumn(&Column,&ScanColumn );return AND;}
add      { GetColumn(&Column,&ScanColumn );return ADD;}
lda      { GetColumn(&Column,&ScanColumn );return LDA;}
sta      { GetColumn(&Column,&ScanColumn );return STA;}
bun      { GetColumn(&Column,&ScanColumn );return BUN;}
bsa      { GetColumn(&Column,&ScanColumn );return BSA;}
isz      { GetColumn(&Column,&ScanColumn );return ISZ;}
&[[:xdigit:]]+ {
                  GetColumn(&Column,&ScanColumn );
if( yyleng -1> LEN_MAX_MEMORY_WORD +1)
{
/* Trunc hexa string to store in array */
memcpy( yylval.sHexStr, yytext +1, LEN_MAX_MEMORY_WORD +1);
                     yylval.sHexStr[ LEN_MAX_MEMORY_WORD +2]='\0';
}
else
{
/* Store hexa string in array without trunc */
memcpy( yylval.sHexStr, yytext +1, yyleng -1);
                     yylval.sHexStr[ yyleng -1]='\0';
}
return HEXNUM;
}
[():;]   { GetColumn(&Column,&ScanColumn );return*yytext;}
[ \t]+    GetColumn( &Column, &ScanColumn );/* ignora espacios en blanco */
\nColumn = ScanColumn = 1;/* restart column */
'[^\n]*   GetColumn( &Column, &ScanColumn );/* ignora comentarios */
.{
          GetColumn(&Column,&ScanColumn );
sprintf( buffer,"Error Lexico: Simbolo '%s' desconocido", yytext );
yyerror( buffer );
return*yytext;
}

%%

voidGetColumn( unsignedint *Column, unsignedint *ScanColumn )
{
*Column     =*ScanColumn;
*ScanColumn =*ScanColumn + yyleng;
}
```

3.5.1.2 Analizador sintactico

Este codigo se escribió según la especificación de Bison para implementar la gramatica.

```
%{
#include <stdio.h>
#include <malloc.h>
#include <alloc.h>
#include <conio.h>
#include <string.h>
#include <math.h>
#include "Global.h"

#define YYDEBUG 0

const TRUE  =1;
const FALSE =0;

externint yylex(void);
externFILE*yyin;
externint yylineno;
externunsignedint Column;

unsignedshort MemPtr =0;

unsignedshort NDatas            =0;
unsignedshort NInstructions     =0;
unsignedshort NBrkPointsAddress =0;
unsignedshort NBrkPointsLines   =0;
unsignedshort NErrors           =0;
unsignedshort NMaxErrors        =10;

struct TypeMemoryCell {
unsignedshort Address;
unsignedshort Word;
};
struct TypeError {
unsignedint Row;
unsignedint Column;
char Message[ LEN_ERROR_MSG ];
};

unsignedshort BrkPointAddress[ MAX_MEMORY_ADDRESS ];
int BrkPointsLines[ MAX_MEMORY_ADDRESS ];
struct TypeMemoryCell Data[ MAX_MEMORY_ADDRESS ];
struct TypeMemoryCell Instruction[ MAX_MEMORY_ADDRESS ];
struct TypeError *Error =NULL;

int Compiler(void);
int IsBrkPointLine(int LineNumber );
int IsMAddress(char*MemoryAddress );
int IsMWord(char*MemoryWord );
unsignedint Dec(char*str_hex);
void AddData(struct TypeMemoryCell *Data,unsignedshort*NDatas, \
unsignedshort*Address,unsignedint Word );
void AddInstruction(struct TypeMemoryCell *Instruction,unsignedshort*NInstructions,
\
struct TypeMemoryCell *Data,unsignedshort NDatas, \
unsignedshort*Address,unsignedint Word, \
unsignedshort*BrkPointAddress,unsignedshort*NBrkPointsAddress, \
int NSourceLine );
void CheckMemoryOverWrite(unsignedshort*Address,unsignedint Word, \
struct TypeMemoryCell *MBlock,unsignedshort*NCell );
void yyerror(char*s);
```

```
%}

%start source_code

%union {
char sHexStr[ LEN_MAX_MEMORY_WORD +2];
unsignedshort iValue;
}

%token<sHexStr> HEXNUM
%token<iValue> DATA_SECT PROGRAM_SECT CLA CLE CMA CME CIR CIL INC SPA SNA SZA
%token<iValue> SZE HLT INP OUT SKI SKO ION IOF AND ADD LDA STA BUN BSA ISZ

%type<iValue> instruction_dir_mem_ref instruction_indir_mem_ref instruction_reg_io
%type<iValue> label_inst_mem_ref label_inst_reg_io

%%

source_code
:DATA_SECTmemory_init_listPROGRAM_SECTcode_list
{}
|DATA_SECTmemory_init_listerror
{ yyerror("Se esperaba 'PROGRAM'.");}
code_list
|error
{ yyerror("Se esperaba 'DATA'.");}
memory_init_listPROGRAM_SECTcode_list


|DATA_SECTPROGRAM_SECTcode_list
{}
|DATA_SECTerror
{ yyerror("Se esperaba 'PROGRAM'.");}
code_list
|error
{ yyerror("Se esperaba 'DATA'.");}
PROGRAM_SECTcode_list
;

memory_init_list
:memory_init_listmemory_init
|memory_init
;

memory_init
:address_data_list':'data_list';'
|address_data_list':'data_list':'
{ yyerror("Se esperaba ';'.");}
|address_data_list':'data_listPROGRAM_SECT
{ yyerror("Se esperaba ';'.");}
/*
  | address_data_list ':' error ';'
      { yyerror( "Se esperaba una palabra de memoria." ); }
*/
|address_data_listerror
{
        yyerror("Se esperaba ':'.");
        yyclearin;
}
';'
|error';'
{ yyerror("Se esperaba una direccion de memoria.");}
;

address_data_list
:HEXNUM
```

```
{
if( IsMAddress($1))
MemPtr = Dec($1);
else
{
            yyerror("La direccion de la palabra de memoria debe estar en el rango
&0-&FFF.");
}
}
;

data_list
:data_listHEXNUM
{
if( IsMWord($2))
          AddData( Data,&NDatas,&MemPtr, Dec($2));
else
{
            yyerror("La palabra de memoria debe estar en el rango &0-&FFFF.");
}
}
|HEXNUM
{
if( IsMWord($1))
          AddData( Data,&NDatas,&MemPtr, Dec($1));
else
{
            yyerror("La palabra de memoria debe estar en el rango &0-&FFFF.");
}
}
|error
{
        yyerror("Se esperaba una palabra de memoria.");
yyclearin;
}
;

code_list
:code_listcode
|code
;

code
:address_instruction_list':'instruction_list
|address_instruction_listerror
{ yyerror("Se esperaba ':'.");
        yyclearin;
}
instruction_list
|error
{
        yyerror("Se esperaba una direccion de memoria.");
}
|address_instruction_list':'instruction_list';'
{ yyerror("Se esperaba una instruccion o direccion de memoria.");}
|address_instruction_list';'
{ yyerror("Antes del ';' anterior se esperaba una instruccion o ':' en lugar de
';'.");}
label_inst_mem_ref
|address_instruction_list';'
{ yyerror("Antes del ';' anterior se esperaba una instruccion o ':' en lugar de
';'.");}
label_inst_reg_io
;

address_instruction_list
```

```
:HEXNUM
{
if( IsMAddress($1))
MemPtr = Dec($1);
else
{
            yyerror("La direccion de la instruccion debe estar en el rango &0-
&FFF.");
}
}
;

instruction_list
:instruction_listinstruction_dir_mem_ref
{
        AddInstruction( Instruction,&NInstructions, Data, NDatas,&MemPtr,$2, \
                        BrkPointAddress,&NBrkPointsAddress, yylineno );
}
|instruction_listinstruction_indir_mem_ref
{
        AddInstruction( Instruction,&NInstructions, Data, NDatas,&MemPtr,$2, \
                        BrkPointAddress,&NBrkPointsAddress, yylineno );
}
|instruction_listinstruction_reg_io
{
        AddInstruction( Instruction,&NInstructions, Data, NDatas,&MemPtr,$2, \
                        BrkPointAddress,&NBrkPointsAddress, yylineno );
}

|instruction_dir_mem_ref
{
        AddInstruction( Instruction,&NInstructions, Data, NDatas,&MemPtr,$1, \
                        BrkPointAddress,&NBrkPointsAddress, yylineno );
}
|instruction_indir_mem_ref
{
        AddInstruction( Instruction,&NInstructions, Data, NDatas,&MemPtr,$1, \
                        BrkPointAddress,&NBrkPointsAddress, yylineno );
}
|instruction_reg_io
{
        AddInstruction( Instruction,&NInstructions, Data, NDatas,&MemPtr,$1, \
                        BrkPointAddress,&NBrkPointsAddress, yylineno );
}
|error';'
{ yyerror("Se esperaba una instruccion.");}
;

instruction_dir_mem_ref
:label_inst_mem_refHEXNUM';'
{
if( IsMAddress($2))
$$=$1| Dec($2);
else
{
            yyerror("La direccion a la que hace referencia la instruccion no es
valida.");
            YYERROR;
}
}
|label_inst_mem_refHEXNUMerror
{ yyerror("Se esperaba ';'.");}
|label_inst_mem_referror';'
{ yyerror("Se esperaba una direccion de memoria.");}
|label_inst_mem_refHEXNUM')';'
{ yyerror("Se esperaba '('.");}
```

```
;

instruction_indir_mem_ref
:label_inst_mem_ref'('HEXNUM')'';'
{
if( IsMAddress($3))
$$=($1|0x8000)| Dec($3);
else
{
yyerror("La direccion a la que hace referencia la instruccion no es valida.");
            YYERROR;
}
}
|label_inst_mem_ref'('HEXNUM')'error
{ yyerror("Se esperaba ';'.");}
|label_inst_mem_ref'('HEXNUMerror';'
{ yyerror("Se esperaba ')'.");}
|label_inst_mem_ref'('error
{ yyerror("Se esperaba una direccion de memoria.");}
')'';'
;

instruction_reg_io
:label_inst_reg_io';'
{$$=$1;}
;

label_inst_mem_ref
:AND{$$=0x0000;}
|ADD     {$$=0x1000;}
|LDA{$$=0x2000;}
|STA{$$=0x3000;}
|BUN{$$=0x4000;}
|BSA{$$=0x5000;}
|ISZ{$$=0x6000;}
;

label_inst_reg_io
:CLA{$$=0x7800;}
|CLE{$$=0x7400;}
|CMA{$$=0x7200;}
|CME{$$=0x7100;}
|CIR{$$=0x7080;}
|CIL{$$=0x7040;}
|INC{$$=0x7020;}
|SPA{$$=0x7010;}
|SNA{$$=0x7008;}
|SZA{$$=0x7004;}
|SZE{$$=0x7002;}
|HLT{$$=0x7001;}
|INP{$$=0xF800;}
|OUT{$$=0xF400;}
|SKI{$$=0xF200;}
|SKO{$$=0xF100;}
|ION{$$=0xF080;}
|IOF{$$=0xF040;}
;

%%

/* * * * * * ------------- Code entry point ------------- * * * * * */
int Compiler(void)
{
int Index;

  #if YYDEBUG
```

```
    yydebug=1;
  #endif

char buffer[ LEN_ERROR_MSG ];

/* Counters initialization */
  NBrkPointsAddress =0;
  NDatas            =0;
  NInstructions     =0;
  NBrkPointsAddress =0;
  NErrors           =0;
  yylineno          =1;

/* Pointers initialization */
  Error =NULL;

/* Parse source code */
  yyin =fopen( NAME_TEMP_FILE,"r");
  yyparse();
fclose(yyin);

if( NErrors ==0)
return1;
else
return0;
}


void yyerror(char*s){
if( NErrors <= NMaxErrors )
{
if( Error ==NULL)
{
          Error =(struct TypeError *)malloc(sizeof(struct TypeError ));
if( Error !=NULL)
{
              Error[ NErrors ].Row     = yylineno;
              Error[ NErrors ].Column  = Column;
memcpy( Error[ NErrors ].Message, s, LEN_ERROR_MSG -1);
              Error[ NErrors ].Message[ LEN_ERROR_MSG -1]='\0';
              NErrors =1;
}
}
else
{
          Error  =(struct  TypeError  *)realloc(  Error,(  NErrors  +1)*sizeof(struct
TypeError ));
if( Error !=NULL)
{
              Error[ NErrors ].Row     = yylineno;
              Error[ NErrors ].Column  = Column;
memcpy( Error[ NErrors ].Message, s, LEN_ERROR_MSG -1);
              Error[ NErrors ].Message[ LEN_ERROR_MSG -1]='\0';
              NErrors++;
}
}
}
}


int IsMAddress(char*MemoryAddress )
{
if(strlen( MemoryAddress )> LEN_MAX_MEMORY_ADDRESS )
return FALSE;
else
return TRUE;
```

```
}


int IsMWord(char*MemoryWord )
{
if(strlen( MemoryWord )> LEN_MAX_MEMORY_WORD )
return FALSE;
else
return TRUE;
}


unsignedint Dec(char*str_hex){
        char longitud_str, i, dig_dec;
        unsignedint d=0;
        longitud_str=strlen(str_hex);
        for(i=longitud_str-1;i>=0;i--){
                switch(str_hex[i]){
                        case '0':;
                        case '1':;
                        case '2':;
                        case '3':;
                        case '4':;
                        case '5':;
                        case '6':;
                        case '7':;
                        case '8':;
                        case '9':dig_dec=str_hex[i]-'0';break;
                        case 'a':;
                        case 'b':;
                        case 'c':;
                        case 'd':;
                        case 'e':;
                        case 'f':dig_dec=str_hex[i]-'a'+10;break;
                        case 'A':;
                        case 'B':;
                        case 'C':;
                        case 'D':;
                        case 'E':;
                        case 'F':dig_dec=str_hex[i]-'A'+10;break;
                }
                d=d+dig_dec*pow(16.0,longitud_str-i-1);
        }
        return d;
}


void AddData(struct TypeMemoryCell *Data,unsignedshort*NDatas, \
unsignedshort*Address,unsignedint Word )
{
  Data[*NDatas ].Address =*Address;
  Data[*NDatas ].Word    = Word;
(*NDatas)++;
(*Address)++;

}


void AddInstruction(struct TypeMemoryCell *Instruction,unsignedshort*NInstructions,
\
struct TypeMemoryCell *Data,unsignedshort NDatas, \
unsignedshort*Address,unsignedint Word, \
unsignedshort*BrkPointAddress,unsignedshort*NBrkPointsAddress, \
int NSourceLine )
{
  Instruction[*NInstructions ].Address =*Address;
```

```
  Instruction[*NInstructions ].Word    = Word;

/* Check break point line */
if( IsBrkPointLine( yylineno ))
{
      BrkPointAddress[*NBrkPointsAddress ]=*Address;
(*NBrkPointsAddress)++;
}

/* Check memory overwrite */
  CheckMemoryOverWrite( Address, Word, Data,&NDatas );

(*NInstructions)++;
(*Address)++;
}


void CheckMemoryOverWrite(unsignedshort*Address,unsignedint Word, \
struct TypeMemoryCell *MBlock,unsignedshort*NCell )
{
int Index;
char buffer[100];
if(*NCell >0)
{
for( Index =0; Index <*NCell; Index++)
if( MBlock[ Index ].Address ==*Address )
{
sprintf( buffer,"Sobre escritura de memoria, %03X:%04X, %03X:%04X", \
*Address, Word, MBlock[ Index ].Address, MBlock[ Index ].Word );
            yyerror( buffer );
}
}
}

int IsBrkPointLine(int LineNumber )
{
int Index;
if( NBrkPointsLines >0)
for( Index =0; Index < NBrkPointsLines; Index++)
if( BrkPointsLines[ Index ]== LineNumber )
return TRUE;
return FALSE;
}
```

3.5.2 Intérprete de Código de Máquina

La siguiente función fue generada en lenguaje C, e implementa el intérprete de código maquina.

```
void ExecInstruction(unsignedshort m[],unsignedshort MaxAddress, \
                     TypeRegister &Reg, DynamicArray<int>&InspectAddress, \
enum TypeUpdateRequest UpdateRequest, \
unsignedshort MWordByDumpLine, TScintilla *CodeEditor, \
                     TMemo *MemoryDump, TMemo *MicroOPEditor, \
                     TListBox *RegisterListBox, TListBox *FlagsListBox, \
                     TListBox *InspectListBox, \
FILE*LogFile )
{
const MAX_LINES_MICROOP_EDITOR =100;
unsignedchar q;
unsignedint aux1,aux2;
long EAC;
unsignedshort InstAddress, AddressChangeMemory;
```

```
  AnsiString MicroOPTxt;
int ExcessLines, Index;

/* ### ciclo fetch  C=0, FR=00 ### */
if( Reg.c ==0)
{
    Reg.MAR = InstAddress = Reg.PC;
Reg.MBR =m[ Reg.MAR ];
Reg.PC  =( Reg.PC > MaxAddress -1)?0: Reg.PC +1;
    q = Reg.OPR = Reg.mbr.op;
Reg.I = Reg.mbr.i;
if( q !=7&& Reg.I ==1) Reg.flip_flop.R =1;
if( q ==7|| Reg.I ==0) Reg.flip_flop.F =1;
if( UpdateRequest == UPDATE_INTERFACE )
{
GetDisAssembler( InstAddress, m[ InstAddress ], MicroOPTxt );
      MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf("\r\n\r\n");
      MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"Ciclo Fetch\r\n"
"C0t0:        MAR <-- PC                    | %03X<-- %03X\r\n"
"C0t1:        MBR <-- M, PC <-- PC + 1      | %04X<-- %04X, %03X<-- %03X + 1\r\n"
"C0t2:        OPR <-- MBR(OP), I <-- MBR(I)| %01X<-- %01X, %01X<-- %01X\r\n", \
      Reg.MAR, Reg.PC -1, \
      Reg.MBR,m[ Reg.MAR ], Reg.PC, Reg.PC -1, \
      Reg.OPR, Reg.mbr.op, Reg.I,Reg.mbr.i );
if( q !=7&& Reg.I ==1)
        MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf("q7'IC0t3:    R <--
%01X\r\n", Reg.flip_flop.R );
else
        MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf("(q7+I')C0t3: F <--
%01X\r\n", Reg.flip_flop.F );
      MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf("\r\n");
}
}

/* ### ciclo indirecto  C=1, FR=01 ### */
if( Reg.c ==1)
{
    aux1 = Reg.MAR = Reg.mbr.ad;
    Reg.MBR =m[ Reg.MAR ];
Reg.flip_flop.F =1; Reg.flip_flop.R =0;
if( UpdateRequest == UPDATE_INTERFACE )
{
      MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"Ciclo Indirecto\r\n"
"C1t0: MAR <-- MBR(AD)    | %03X<-- %03X\r\n"
"C1t1: MBR <-- M          | %04X<-- %04X\r\n"
"C1t2:\r\n"
"C1t3: F   <-- %01X, R <-- %01X\r\n\r\n", \
Reg.MAR, aux1, \
      Reg.MBR,m[ Reg.MAR ], \
Reg.flip_flop.F, Reg.flip_flop.R );
}
}

/* ### ciclo ejecucion  C=2, FR=10 ### */
if( Reg.c ==2)
{
if( UpdateRequest == UPDATE_INTERFACE )
      MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf("Ciclo de Ejecucion\r\n");
switch( q )/* INST_REFERENCIA_MEMORIA */
{
case0:/* AND */
aux1 = Reg.MAR = Reg.mbr.ad;
        Reg.MBR =m[ Reg.MAR ]; aux2 = Reg.AC;
        Reg.AC = Reg.AC & Reg.MBR;
```

```
if( UpdateRequest == UPDATE_INTERFACE )
{
          MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"q0C2t0: MAR <-- MBR(AD)     | %03X<-- %03X\r\n"
"q0C2t1: MBR <-- M           | %04X<-- %04X\r\n"
"q0C2t2: AC  <-- AC ^ MBR    | %04X<-- %04X ^ %04X\r\n", \
Reg.MAR, aux1, \
          Reg.MBR,m[ Reg.MAR ], \
          Reg.AC, aux2,Reg.MBR );
}
break;

case1:/* ADD */
        aux1 = Reg.MAR = Reg.mbr.ad;
Reg.MBR =m[ Reg.MAR ];
EAC =long( Reg.AC )+long( Reg.MBR ); aux2 = Reg.AC;
        Reg.AC = EAC &0xFFFF; Reg.E =( EAC>>16)?1:0;
if( UpdateRequest == UPDATE_INTERFACE )
{
          MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"q1C2t0: MAR <-- MBR(AD)     | %03X<-- %03X\r\n"
"q1C2t1: MBR <-- M           | %04X<-- %04X\r\n"
"q1C2t2: EAC <-- AC + MBR    | %0X%04X<-- %04X + %04X\r\n", \
Reg.MAR, aux1, \
          Reg.MBR,m[ Reg.MAR ], \
          Reg.E, Reg.AC, aux2,Reg.MBR );
}
break;

case2:/* LDA */
        aux1 = Reg.MAR = Reg.mbr.ad;
Reg.MBR =m[ Reg.MAR ]; Reg.AC =0;
        Reg.AC = Reg.AC + Reg.MBR;
if( UpdateRequest == UPDATE_INTERFACE )
{
          MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"q2C2t0: MAR <-- MBR(AD)        | %03X<-- %03X\r\n"
"q2C2t1: MBR <-- M, AC <-- 0    | %04X<-- %04X, 0 <-- 0\r\n"
"q2C2t2: AC  <-- AC + MBR       | %04X<-- 0 + %04X\r\n", \
Reg.MAR, aux1, \
          Reg.MBR,m[ Reg.MAR ], \
Reg.AC,Reg.MBR );
}
break;

case3:/* STA */
        aux1 = Reg.MAR = Reg.mbr.ad;
        Reg.MBR = Reg.AC;
m[ Reg.MAR ]= Reg.MBR;
        AddressChangeMemory = Reg.MAR;
RefreshEditors( AddressChangeMemory, m, MWordByDumpLine, MaxAddress, CodeEditor,
MemoryDump );
if( UpdateRequest == UPDATE_INTERFACE )
{
          MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"q3C2t0: MAR <-- MBR(AD)    | %03X<-- %03X\r\n"
"q3C2t1: MBR <-- AC         | %04X<-- %04X\r\n"
"q3C2t2: M   <-- MBR        | %04X<-- %04X\r\n", \
Reg.MAR, aux1, \
          Reg.MBR, Reg.AC, \
m[ Reg.MAR ], Reg.MBR );
}
break;

case4:/* BUN */
        Reg.PC = Reg.mbr.ad;
```

```
if( UpdateRequest == UPDATE_INTERFACE )
{
          MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"q4C2t0: PC  <-- MBR(AD)    | %03X<-- %03X\r\n"
"q4C2t1:\r\n"
"q4C2t2:\r\n", \
          Reg.PC,Reg.mbr.ad );
}
break;

case5:/* BSA */
        Reg.MAR = Reg.mbr.ad; aux1 = Reg.PC; aux2 = Reg.PC = Reg.mbr.ad; Reg.mbr.ad
= aux1;
m[ Reg.MAR ]= Reg.MBR;
AddressChangeMemory = Reg.MAR;
        Reg.PC = Reg.PC +1;
RefreshEditors( AddressChangeMemory, m, MWordByDumpLine, MaxAddress, CodeEditor,
MemoryDump );
if( UpdateRequest == UPDATE_INTERFACE )
{
          MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"q5C2t0: MAR<-- MBR(AD), A1     <-- PC| %03X<-- %03X, %03X<-- %03X\r\n"
"        PC <-- MBR(AD), MBR(AD)<-- A1| %03X<-- %03X, %03X<-- %03X\r\n"
"q5C2t1: M  <-- MBR                   | %04X<-- %04X\r\n"
"q5C2t2: PC <-- PC + 1                | %03X<-- %03X + 1\r\n", \
Reg.MAR, Reg.MAR, aux1, aux1, \
          aux2, aux2, Reg.mbr.ad, aux1, \
m[ Reg.MAR ], Reg.MBR, \
Reg.PC,aux2 );
}
break;

case6:/* ISZ */
aux1 = Reg.MAR = Reg.mbr.ad;
        aux2 = Reg.MBR =m[ Reg.MAR ];
        Reg.MBR = Reg.MBR +1;
m[ Reg.MAR ]= Reg.MBR;
AddressChangeMemory = Reg.MAR;
if( Reg.MBR ==0) Reg.PC = Reg.PC +1;
RefreshEditors( AddressChangeMemory, m, MWordByDumpLine, MaxAddress, CodeEditor,
MemoryDump );
if( UpdateRequest == UPDATE_INTERFACE )
{
          MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"q6C2t0: MAR <-- MBR(AD)      | %03X<-- %03X\r\n"
"q6C2t1: MBR <-- M            | %04X<-- %04X\r\n"
"q6C2t2: MBR <-- MBR + 1      | %04X<-- %04X + 1\r\n"
"q6C2t3: M   <-- MBR          | %04X<-- %04X\r\n", \
Reg.MAR, aux1, \
          aux2, aux2, \
          Reg.MBR, aux2, \
m[ Reg.MAR ], Reg.MBR );
if( Reg.MBR ==0)
            MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
" si(MBR=0) => (PC<-- PC + 1) | si(%04X=0) => (%03X<-- %03X + 1)\r\n", \
            Reg.MBR, Reg.PC, Reg.PC -1);
else
            MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"si(MBR=0) => (PC<-- PC + 1) | si(%04X=0) es Falso\r\n", Reg.MBR );
}
break;

case7:
if( Reg.I ==0)
switch( Reg.mbr.ad )/* INST_REFERENCIA_REG */
{
```

```
caseMBR_bit5:/* CLA */
                Reg.AC =0;
if( UpdateRequest == UPDATE_INTERFACE )
                  MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf("rB5: AC <-- 0
\r\n");
break;

caseMBR_bit6:/* CLE */
                Reg.E =0;
if( UpdateRequest == UPDATE_INTERFACE )
                  MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf("rB6: E  <-- 0
\r\n");
break;

caseMBR_bit7:/* CMA */
                Reg.AC =~Reg.AC;
if( UpdateRequest == UPDATE_INTERFACE )
                  MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"rB7: AC <-- AC'   | %04X<-- %04X'\r\n", Reg.AC,~Reg.AC );
break;

caseMBR_bit8:/* CME */
                aux1 = Reg.E;
                Reg.E =~Reg.E &1;
if( UpdateRequest == UPDATE_INTERFACE )
                  MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"rB8: E  <-- E'    | %01X<-- %d'\r\n", Reg.E,aux1 );
break;

caseMBR_bit9:/* CIR */
                aux1 = Reg.E; aux2 = Reg.AC;
                Reg.AC = Reg.AC >>1; Reg.ac.bit1 = Reg.E;
if( UpdateRequest == UPDATE_INTERFACE )
                  MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"rB9: EAC <-- cir EAC   | %01X%04X<-- cir %01X%04X\r\n", \
                  Reg.E, Reg.AC, aux1,aux2 );
break;

caseMBR_bit10:/* CIL */
                aux1 = Reg.E; aux2 = Reg.AC;
                Reg.AC = Reg.AC <<1; Reg.AC = Reg.AC | Reg.E;
if( UpdateRequest == UPDATE_INTERFACE )
                  MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"rB10: EAC <-- cil EAC   | %01X%04X<-- cil %01X%04X\r\n", \
                  Reg.E, Reg.AC, aux1,aux2 );
break;

caseMBR_bit11:/* INC */
                aux1 = Reg.AC;
                EAC =long( Reg.AC )+1; Reg.AC = EAC &0xFFFF; Reg.E =( EAC
>>16)?1:0;
if( UpdateRequest == UPDATE_INTERFACE )
                  MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"rB11: EAC <-- AC + 1   | %01X%04X<-- %04X + 1\r\n", \
                  Reg.E, Reg.AC,aux1 );
break;

caseMBR_bit12:/* SPA */
if( Reg.ac.bit1 ==0) Reg.PC = Reg.PC +1;
if( UpdateRequest == UPDATE_INTERFACE )
if( Reg.ac.bit1 ==0)
                    MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"rB12: si(AC(1)=0) => (PC <-- PC + 1)   | si(0=0) => (%03x<-- %03X + 1)\r\n", \
Reg.PC, Reg.PC -1);
else
                    MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
```

```
"rB12: si(AC(1)=0) => (PC <-- PC + 1)    | si(1=0) es Falso\r\n");
break;

caseMBR_bit13:/* SNA */
if( Reg.ac.bit1 ==1) Reg.PC = Reg.PC +1;
if( UpdateRequest == UPDATE_INTERFACE )
if( Reg.ac.bit1 ==1)
                    MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"rB13: si(AC(1)=1) => (PC <-- PC + 1)    | si(1=1) => (%03x<-- %03X + 1)\r\n", \
Reg.PC, Reg.PC -1);
else
                    MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"rB13: si(AC(1)=1) => (PC <-- PC + 1)    | si(0=1) es Falso\r\n");
break;

caseMBR_bit14:/* SZA */
if( Reg.AC ==0) Reg.PC = Reg.PC +1;
if( UpdateRequest == UPDATE_INTERFACE )
if( Reg.AC ==0)
                    MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"rB14: si(AC=0) => (PC <-- PC + 1)    | si(0=0) => (%03x<-- %03X + 1)\r\n", \
Reg.PC, Reg.PC -1);
else
                    MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"rB14: si(AC=0) => (PC <-- PC + 1)    | si(%04X=0) es Falso\r\n", \
Reg.AC );
break;

caseMBR_bit15:/* SZE */
if( Reg.E ==0) Reg.PC = Reg.PC +1;
if( UpdateRequest == UPDATE_INTERFACE )
if( Reg.E ==0)
                    MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"rB15: si(E=0) => (PC <-- PC + 1)    | si(0=0) => (%03X<-- %03X + 1)\r\n", \
Reg.PC, Reg.PC -1);
else
                    MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"rB15: si(E=0) => (PC <-- PC + 1)    | si(1=0) es Falso\r\n");
break;

caseMBR_bit16:/* HLT */
                Reg.S =0;
if( UpdateRequest == UPDATE_INTERFACE )
                  MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf("rB16: S <--
0\r\n");
break;

default:
                Reg.S =0;
DisplayRegisters( AllRegisters, Reg, RegisterListBox );
DisplayFlags( AllFlags, Reg, FlagsListBox );
                Application->MessageBox("Codigo de instruccion no valido", \
"Error de ejecución", MB_OK );
}
else
switch( Reg.mbr.ad )/* INST_E/S */
{
caseMBR_bit5:/* INP */
                Reg.ac.bits9_16 = Reg.INPR; Reg.FGI =0;
if( UpdateRequest == UPDATE_INTERFACE )
                  MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"pB5: AC(9-16) <-- INPR, FGI <-- 0    | %02X<-- %02X, 0 <-- 0\r\n", \
                  Reg.ac.bits9_16,Reg.INPR );
break;

caseMBR_bit6:/* OUT */
```

```
                Reg.OUTR = Reg.ac.bits9_16; Reg.FGO =0;
if( UpdateRequest == UPDATE_INTERFACE )
                  MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"pB6: OUTR <-- AC(9-16), FGO <-- 0    | %02X<-- %02X, 0 <-- 0\r\n", \
                  Reg.OUTR, Reg.ac.bits9_16 );
break;

caseMBR_bit7:/* SKI */
if( Reg.FGI ==1) Reg.PC = Reg.PC +1;
if( UpdateRequest == UPDATE_INTERFACE )
if( Reg.FGI ==1)
                    MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"pB7: si(FGI=1) => (PC <-- PC + 1)    | si(1=1) => (%03X<-- %03X + 1)\r\n", \
Reg.PC, Reg.PC -1);
else
                    MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"pB7: si(FGI=1) => (PC <-- PC + 1)    | si(0=1) es Falso\r\n");
break;

caseMBR_bit8:/* SKO */
if( Reg.FGO ==1) Reg.PC = Reg.PC +1;
if( UpdateRequest == UPDATE_INTERFACE )
if( Reg.FGO ==1)
                    MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"pB8: si(FGO=1) => (PC <-- PC + 1)    | si(1=1) => (%03X<-- %03X + 1)\r\n", \
Reg.PC, Reg.PC -1);
else
                    MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"pB8: si(FGO=1) => (PC <-- PC + 1)    | si(0=1) es Falso\r\n");
break;

caseMBR_bit9:/* ION */
                Reg.IEN =1;
if( UpdateRequest == UPDATE_INTERFACE )
                  MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"pB9:  IEN <-- 1                      | 1 <-- 1\r\n");
break;

caseMBR_bit10:/* IOF */
                Reg.IEN =0;
if( UpdateRequest == UPDATE_INTERFACE )
                  MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"pB10: IEN <-- 0                      | 0 <-- 0\r\n");
break;

default:
                Reg.S =0;
DisplayRegisters( AllRegisters, Reg, RegisterListBox );
DisplayFlags( AllFlags, Reg, FlagsListBox );
                Application->MessageBox("Codigo de instruccion no valido", \
"Error de ejecución", MB_OK );
}
}
if(( Reg.IEN &&( Reg.FGI || Reg.FGO ))==1)
{
        Reg.flip_flop.R =1;
if( UpdateRequest == UPDATE_INTERFACE )
          MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"C2t3: Si [IEN ^ (FGI v FGO)=1] => (R <-- 1)    | "
"Si [%01X ^ (%01X v %01X)=1] => (R <-- %01X)\r\n", \
          Reg.IEN, Reg.FGI, Reg.FGO, Reg.flip_flop.R );
}
else/* (IEN && (FGI || FGO))==0 */
{
        Reg.flip_flop.F =0;
if( UpdateRequest == UPDATE_INTERFACE )
```

```
          MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"C2t3: Si [IEN ^ (FGI v FGO)=0] => (F <-- 0)    | "
"Si [%01X ^ (%01X v %01X)=0] => (F <-- %01X)\r\n", \
          Reg.IEN, Reg.FGI, Reg.FGO, Reg.flip_flop.F );
}
    MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf("\r\n");
}

/* ### ciclo interrupcion C=3, FR=11 ### */
if( Reg.c ==3)
{
    Reg.mbr.ad = Reg.PC; aux1 = Reg.PC; Reg.PC =0;
    Reg.MAR = Reg.PC; Reg.PC = Reg.PC +1;
m[ Reg.MAR ]= Reg.MBR;
AddressChangeMemory = Reg.MAR;
    Reg.IEN =0;
    Reg.flip_flop.F =0; Reg.flip_flop.R =0;
RefreshEditors( AddressChangeMemory, m, MWordByDumpLine, MaxAddress, CodeEditor,
MemoryDump );
if( UpdateRequest == UPDATE_INTERFACE )
{
      MicroOPTxt =AnsiString( MicroOPTxt ).cat_sprintf( \
"Ciclo de Interrupcion\r\n"
"C3t0: MBR(AD) <-- PC, PC <-- 0     | %03X<-- %03X, %03X<-- 0\r\n"
"C3t1: MAR <-- PC, PC <-- PC + 1    | %03X<-- %03X, %03X<-- %03X + 1\r\n"
"C3t2: M   <-- MBR, IEN <-- 0       | %04X<-- %04X, %01X<-- 0\r\n"
"C3t3: F <-- %01X, R <-- %01X\r\n\r\n", \
aux1, aux1, Reg.PC -1, \
      Reg.MAR, Reg.PC -1, Reg.PC, Reg.PC -1, \
m[ Reg.MAR ], Reg.MBR, Reg.IEN, \
Reg.flip_flop.F, Reg.flip_flop.R );
}
}

if( UpdateRequest == UPDATE_INTERFACE )
{
/* Add micro-operations of executed instruction to editor */
    MicroOPEditor->Lines->Add( MicroOPTxt );

/* Check max lines of micro-operations editor */
if( MicroOPEditor->Lines->Count > MAX_LINES_MICROOP_EDITOR )
{
/* Get number of excess lines and delete lines from start */
      ExcessLines = MicroOPEditor->Lines->Count - MAX_LINES_MICROOP_EDITOR;
for( Index =0; Index < ExcessLines; Index++)
        MicroOPEditor->Lines->Delete(0);
}

/* Update values of registers, flags and inspect list  */
DisplayRegisters( AllRegisters, Reg, RegisterListBox );
DisplayFlags( AllFlags, Reg, FlagsListBox );
DisplayInspect( m, MaxAddress, InspectAddress, InspectListBox );
}

/* Save micro-operations to log file */
if( LogFile !=NULL)
{
    MicroOPTxt =StringReplace( MicroOPTxt,"\r","", TReplaceFlags()<< rfReplaceAll
);
fprintf( LogFile,"%s", MicroOPTxt.c_str());
}
}
```

3.5.3 Emulador

Para el desarrollo del emulador se utilizó el lenguaje C y el compilador Builder C++ 6. Además, para el editor de código se usó el componente **Scintilla** que es un componente libre para la edición de código fuente.

Conclusiones

A lo largo del presente trabajo se desarrolló una máquina virtual para emular el funcionamiento del computador básico hipotético especificado por Morris Mano (1993) utilizado para el estudio de micro-operaciones. El computador emulado no es compatible con los actuales. Por tal motivo, los alumnos no disponen de un computador sobre el cual ejecutar sus programas y realizan sus prácticas en forma manual, mediante pruebas de escritorio.

Mediante el emulador del computador básico, los alumnos pueden ejecutar sus programas y depurarlos, permitiendo la interactividad con los alumnos, retroalimentando y evaluando lo aprendido. Además, los alumnos pueden ver los registros y banderas del procesador, las micro-operaciones ejecutadas y el vuelco de memoria.

Esta máquina virtual, no tiene las limitaciones de una máquina física, en cuanto a que puede ser copiada y distribuida en forma sencilla sin costo alguno.

Con el trabajo realizado:

- Se pone a disposición de los docentes, de la Cátedra Sistemas Operativos de la Universidad Nacional de Catamarca, una herramienta que contribuye a la enseñanza de micro-operaciones y además facilita a los alumnos el autoaprendizaje.
- Se brinda una herramienta tecnológica como soporte a la asignatura en el contexto del aula y fuera de ella, de fácil instalación y puede utilizarse disponiendo de una PC con requerimientos míninos sobre el cual se pueda ejecutar el emulador sin depender de una maquina física.
- Se amplia las posibilidades de intervención del profesor así como de autoaprendizaje del alumno, al proveer funcionalidades que permiten la corrección de errores y ejecución del código paso a paso, mostrando los valores de registros y banderas del procesador.

Anexos

ANEXO I: GENERADOR DE ANALIZADORES LEXICOS FLEX

Hasta 1975, escribir un compilador era un proceso que consumía mucho tiempo. Entonces Lesk[1975] y Johnson [1975] publicaron artículos sobre lex y yacc. Estas utilidades simplificaron en gran medida la escritura de un compilador. Los detalles de implementación para lex y yacc se pueden encontrar en Aho [1986].

Lex y Yacc están disponibles en

- Mortice Kern Systems (MKS), at http://www.mks.com,
- GNU flex and bison, at http://www.gnu.org,
- Ming, at http://agnes.dida.physik.uni-essen.de/~janjaap/mingw32,
- Cygnus, at http://www.cygnus.com/misc/gnu-win32, and
- MiversióndeLex yYacc, at http://epaperpress.com.

La versión de MKS es un producto comercial de alta calidad es software GNU es libre. La salida de Flex se puede utilizar en un producto comercial, y, a partir de la versión 1.24, esto es también aplicable a bison. Ming y Cygnus son ports de 32-bit de Windows 95/NT del software GNU.

La versión usada se basa en Ming, pero está compilado con Visual C++, e incluye una corrección de errores de menor importancia enla rutina demanejo de archivos. Si descargas mi versión, asegúrese de mantener la estructura de directorios cuando se descomprima.

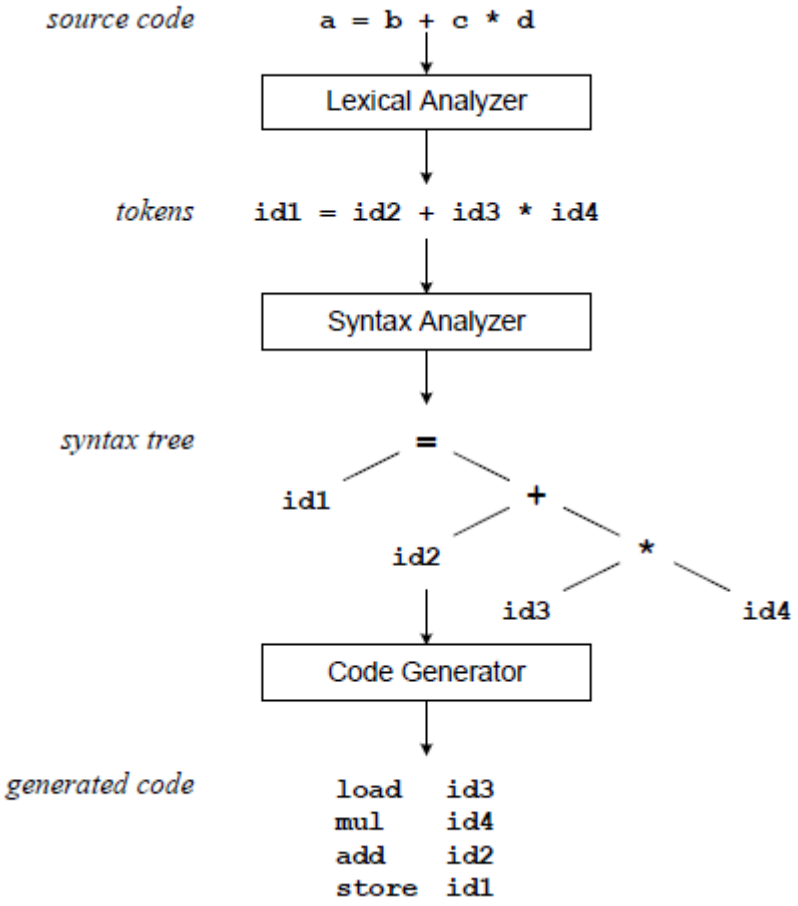

Figura 5-1 Secuencia de Compilación.

Lex genera código C para un analizador léxico, o escáner. Utiliza patrones que coincidan con las cadenas en la entrada y convierte las cadenas en tokens. Los tokens son representaciones numéricas de cadenas, y simplifican el proceso. Esto se ilustra en la Figura 5-1. A medida que lex encuentra identificadores en el flujo de entrada, los introduce en una tabla de símbolos. La tabla de símbolos también puede contener otra información tal como tipo de datos (entero o real) y ubicación de la variable en la memoria.Todas las referencias posteriores a identificadores se refieren al índice correspondiente de la tabla de símbolos.

Yacc genera código C para un analizador sintáctico o parser. Yacc utiliza reglas gramaticales que le permiten analizar tokens desde lex y crear un árbol sintáctico. Un árbol sintáctico impone una estructura jerárquica sobre los tokens. Por ejemplo, la precedencia y asociatividad de un operador son evidentes en el árbol sintáctico. El siguiente paso, la generación de código, hace un recorrido en profundidad del árbol sintáctico para generar código. Algunos compiladores producen código máquina, mientras que otros, como se muestra arriba, producen como salida asembler.

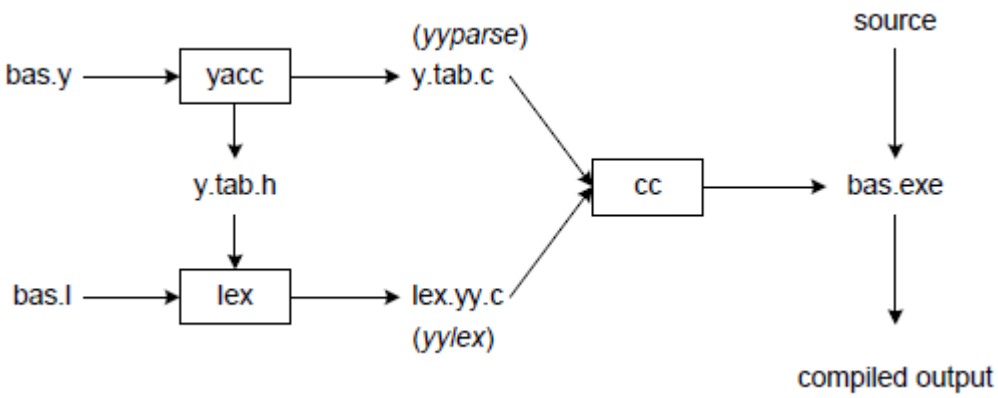

Figura 5-2 Construcción de un compilador con Lex /Yacc.

La Figura 5-2 ilustra las convenciones de nombres de archivo usadas por lex y yacc.Vamos a suponer que nuestro objetivo es escribir un compilador BASIC. En primer lugar, tenemos que especificar todas las reglas de coincidencia de patrones para lex (bas.l) y las reglas gramaticales para yacc (bas.y). Los comandos para crear nuestro compilador, bas.exe, se enumeran a continuación:

```
yacc -d bas.y                 #creay.tab.h,y.tab.c
lexbas.l                      #crealex.yy.c
cclex.yy.cy.tab.c-obas.exe    #compila/ link
```

Yacc lee las descripciones gramaticales en bas.yy genera un parser, la función yyparse, en el archivo y.tab.c. Las declaraciones de tokens están incluidas en el archivo bas.y. La opción –d hace que yacc genere definiciones para los tokens y las coloque en el archivo y.tab.h. Lex lee las descripciones de los patrónes en bas.l, incluye el archivo y.tab.h, y genera un

analizador léxico, la función yylex, en el archivo lex.yy.c. Finalmente, el analizador léxico y el parser se compilan y enlazan para formar el ejecutable, bas.exe. Desde main, llamamos yyparse para ejecutar el compilador. La función yyparse llama automáticamente a yylex para obtener cada token.

Lex

La primera fase de un compilador lee el código fuente de entrada y convierte las cadenas en el código fuente en tokens. Utilizando expresiones regulares, podemos especificar patrones para lex que permitan explorar y hacer coincidir cadenas en la entrada. Cada patrón en lex tiene una acción asociada. Típicamente, una acción devuelve un token, representando la cadena coincidente, para su uso posterior por el parser.

Para empezar, sin embargo, simplemente se imprimirá la cadena coincidente en lugar de devolver un valor de token. Podemos buscar los identificadores utilizando la expresión regular

letra(letra | dígito) *

Este patrón coincide con una cadena de caracteres que comienza con una letra, y es seguida por cero o más letras o dígitos. Este ejemplo ilustra muy bien las operaciones permitidas en expresiones regulares:

- repetición, expresada por el operador "*"
- alternancia, expresada por el operador "|"
- concatenación

Cualquier expresión de expresiones regulares puede expresarse como un autómata de estado finito (FSA). Podemos representar una FSA utilizando estados y transiciones entre estados. Hay un estado inicial, y uno o más finales o estados de aceptación.

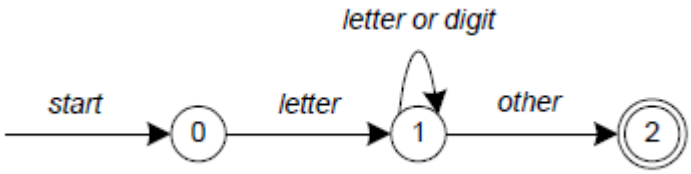

Figura 5-3 Autómata de Estado Finito.

En la Figura 5-3, el estado 0 es el estado inicial y el estado 2 es el estado de aceptación. A medida que los caracteres se leen, hacemos una transición de un estado a otro.Cuando la primera letra se lee, hacemos una transición al estado1. Seguimos en el estado 1cuando más letras o dígitos se leen. Cuando leemos un carácter que no sea una letra o un dígito, hacemos la transición al estado 2, el estado de aceptación. Cualquier FSA se puede expresar como un programa de ordenador. Por ejemplo, nuestra máquina de 3 estados es fácilmente programada:

```
start:     goto state0

state0:    read c
     if c = letter goto state1
     goto state0

state1:    read c
     if c = letter goto state1
     if c = digit goto state1
     goto state2

state2:    accept string
```

Esta es la técnica utilizada por Lex. Las expresiones regulares son traducidas por lex a un programa de ordenador que imita a una FSA. Usando el siguiente carácter de entrada, y el estado actual, el siguiente estado se determina fácilmente mediantela indización en una tabla de estados generada por ordenador.

Ahora se puede entender algunas de las limitaciones de Lex. Por ejemplo, Lex no puede ser utilizado para reconocer estructuras anidadas como paréntesis. Las estructuras anidadas son manejadas por la incorporación de una pila. Cada vez que encontramos un "(", empujamos este a la pila. Cuando un ")" es encontrado, emparejamos este con el de la parte superior de la pila, y lo sacamos de la pila. Lex sin embargo, sólo cuenta con los estados y las transiciones entre estados. Ya que no tiene pila, no es muy adecuado para analizarlas estructuras anidadas.Yacc aumenta el FSA con una pila, y puede procesar construcciones tales como los paréntesis con facilidad. Lo importante es usar la herramienta adecuada para el trabajo. Lex es bueno en la coincidencia de patrones.Yacc es apropiado para tareas más complejas.

ANEXO II: ANALIZADOR SINTÁCTICO BISON YACC

Las gramáticas para yacc son descriptas utilizando una variante de la Forma de Backus Naur (BNF). Esta técnica fue desarrollada por John Backus and Peter Naur, y se utilizó para describir ALGOL60. Una gramática BNF puede ser utilizada para expresar lenguajes libres de contexto.La mayoría de las construcciones en los lenguajes de programación modernos pueden ser representadas en BNF. Por ejemplo, la gramática de una expresión que multiplica y suma números es

```
1 E -> E + E
2 E -> E * E
3 E -> id
```

Tres producciones han sido especificadas. Los términos que aparecen en el lado izquierdo (LHS) de un producción, tales como E (expresión) son no terminales. Términos tales como id (identificador) son terminales (tokens devueltos por lex) y sólo aparecen en el lado derecho (RHS) de un producción. Esta gramática especifica que una expresión puede ser la suma de dos expresiones, el producto de dos expresiones, o un identificador. Podemos utilizar esta gramática para generar expresiones:

```
E -> E * E          (r2)
  -> E * z          (r3)
  -> E + E * z      (r1)
  -> E + y * z      (r3)
  -> x + y * z      (r3)
```

En cada paso que se expandió un término, reemplazando el LHS de una producción con el RHS correspondiente. Los números de la derecha indican qué regla fue aplicada. Para analizar una expresión, en realidad necesitamos hacer la operación inversa. En lugar de comenzar con un simple no terminal (símbolo inicial) y generar una expresión desde una gramática, tenemos que reducir una expresión a un solo no terminal. Esto se conoce como análisis de abajo hacia arriba o desplazamiento-reduccion, y utiliza una pila para almacenar los términos. Aquí está la misma derivación, pero en orden inverso:

```
1 . x + y * z     shift
2 x . + y * z     reduce(r3)
3 E . + y * z     shift
4 E + . y * z     shift
5 E + y . * z     reduce(r3)
6 E + E . * z     shift
7 E + E * . z     shift
8 E + E * z .     reduce(r3)
9 E + E * E .     reduce(r2) emit multiply
10 E + E .        reduce(r1) emit add
11 E .            accept
```

Los términos a la izquierda del punto están en la pila, mientras que la entrada restante está a la derecha del punto. Empezamos desplazando tokens a la pila. Cuando el tope de la pila coincide con el RHS de una producción, reemplazamos los tokens coincidentes en la pila con el LHS de la producción. Conceptualmente, los tokens coincidentes del RHS se extraen de la pila, y el LHS de la producción se inserta en la pila. Los tokens coincidentes se

conocen como mangos, y estamos reduciendo el mango a el LHS de la producción. Este proceso continúa hasta haber desplazado toda la entrada a la pila, y sólo queda en la pila el no terminal inicial. En el paso 1 desplazamos x a la pila. En el paso 2 se aplicala regla r3 a la pila, cambiando x a E. Seguimos desplazando y reduciendo, hasta que un solo no terminal, el símbolo inicial, permanezca enla pila. En el paso 9, cuando reducimos la regla r2, emitimos la instrucción de multiplicación. De manera similar, la instrucción suma se emite en el paso 10. Por lo tanto, la multiplicación tiene una mayor precedencia que la suma. Considere, sin embargo, el desplazamiento en el paso 6. En lugar de desplazar, podríamos haber reducido, aplicando la reglar 1. Esto resultaría en que la adición tendría un aprecedencia mayor que la multiplicación. Esto se conoce como un conflicto desplazamiento - reducción. Nuestra gramática es ambigua, ya que hay más de una posible derivación que producirá la expresión. En este caso, la precedencia de operadores se ve afectada. Como otro ejemplo, la asociatividad en la regla

```
E -> E + E
```

Es ambigua, porque se puede efectuar recursividad a la izquierda ó a la derecha. Para remediar la situación, se puede reescribir la gramática, o suministrar a yacc con directivas que indiquen que operador tiene precedencia. El último método es más simple. La siguiente gramática tiene un conflicto reducción - reducción. Con un id en la pila, podríamos reducir a T, o reducir a E.

```
E -> T
E -> id
T -> id
```

Yacc toma una acción predeterminada cuando se produce un conflicto. Para los conflictos desplazamiento-reducción, yacc desplazará. Para los conflictos reducción-reducción, se utilizará la primera regla en la lista. También emite un mensaje de advertencia cada vez que existe un conflicto. Los avisos pueden ser suprimidos al hacer la gramática no ambigua.

REFERENCIAS

Agilis IT Team Australia (2015). Online syntax highlighting. Sitio Web: http://www.tohtml.com/. Accedido el 07/12/15.

Aho Alfred V., Ravi Sethi, Jeffrey D. Ullman; (1990). Compiladores Principios, técnicas y herramientas; Ed. Addison Wesley Longman, 1990.

Berzal, F. (2015) . El ciclo de vida de un sistema de información. Recuperado de http://elvex.ugr.es/idbis/db/docs/lifecycle.pdf

Fennema Cristina. (1993); Notas de Clase: Aspectos de Diseño e Implementación de los Sistemas Operativos, 1993.

Gomez Gonzalez, Isabel. M. (2008). Emulador del computador simple 2. Departamento de Tecnología Electrónica de la Universidad de Sevilla — Última modificación 17/04/2008 12:06 [On-line]. Disponible en: http://150.214.141.196/docencia/etsii/itis/ec/Emuladores/cs2/view

Grossi, M. D., Jiménez Rey, M. E., Servetto, Ar. C., Perichinsky, G. (2005). Un Simulador de una Maquina Computadora como Herramienta Para La Enseñanza de la Arquitectura de Computadoras. Departamento de Computación de la Facultad de Ingeniería de la Universidad de Buenos Aires [On-line]. Disponible en: http://cs.uns.edu.ar/jeitics2005/Trabajos/pdf/06.pdf.

Guevara, J. (2015) *Analisis y Diseño detallado de aplicaciones informaticas de gestion*. Principales metodologías de desarrollo europeas: Recuperado de https://sites.google.com/site/adai6jfm/principales-metodologas-de-desarrollo-europeas

IEEE Recommended Practice for Software Requeriments Specification. IEEE std. 830, 1998.

IEEE Std 729 (1993). IEEE Software Engineering Standard: Glossary of Software Engineering Terminology. IEEE Computer Society Press, 1993

Maldonado, Vargas Y Monroy (1997), Pedagogía e informática - hacia el diseño de ambientes de aprendizaje, en revista educación y cultura**,** No. 44, Julio de 1997, pp.

Morris Mano M. (1993); Arquitectura de Computadoras; Ed. Prentice-Hall Hispanoamericana S.A, 1993.

Perez Garcia, A. (2006). *Metodologías para el Desarrollo de Sistemas. Casos de Estudio: Métrica II y Merise*. Tesis de grado, México. Recuperado de http://dgsa.uaeh.edu.mx:8080/bibliotecadigital/bitstream/handle/231104/321/Metodologias%20para%20el%20desarrollo%20de%20sistemas.pdf?sequence=1&isAllowed=y

Pressman, R. S. (2005). *Ingeniería del software: Un enfoque práctico*. 6ta Edición –McGraw-Hil / Interamericana. España

Rivas, Abdel (2012). *Análisis, análisis estructurado y orientado a objeto*. 29 de noviembre de 2012. Recuperado de http://mundoinformatico321.blogspot.com.ar/2012/11/analisis-analisis-estructurado-y.html.

Sommerville L. (2005). *Ingeniería del Software*. Séptima edición. PEARSON EDUCACIÓN. S. A. Madrid. 2005. ISBN: 84-7829-074-5.

Wikipedia la Enciclopedia Libre. Sitio Web: http://es.wikipedia.org/. Accedido el 02/03/15.

Yourdon, E. (1993). *Análisis Estructurado Moderno.* Prentice-Hall Hispanoamericana. México, 1993.

BIBLIOGRAFÍA

Hopcroft John E., Ullman Jeffrey D.; Introducción a la teoría de autómatas, lenguajes y computación; Ed. Compañía Editorial Continental, 1998.

NEUFERT, Ernst. *Arte de proyectar en arquitectura*. 14a. ed. Barcelona: Gustavo Gili,1999. 580 p.ISBN: 8425200539

Pressman, R. S. (2005). *Ingeniería del software: Un enfoque práctico*. 6ta Edición –McGraw-Hil / Interamericana. España

Sommerville L. (2005). *Ingeniería del Software*. Séptima edición. PEARSON EDUCACIÓN. S. A. Madrid. 2005. ISBN: 84-7829-074-5.

Yourdon, Edward . (1993). Análisis Estructurado Moderno. Prentice-Hall Hispanoamericana. México, 1993.