UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

DESARROLLO DE COMPILADOR PARA UN PLC

T E S I S

QUE PARA OBTENER EL GRADO DE:
INGENIERO EN COMPUTACIÓN
P R E S E N T A

CUAUHTÉMOC GONZÁLEZ JUÁREZ

DIRECTOR DE TESIS: M. en I. MIGUEL ANGEL BAÑUELOS SAUCEDO

LABORATORIO DE ELECTRÓNICA
CENTRO DE CIENCIAS APLICADAS Y DESARROLLO TECNOLÓGICO

MÉXICO, D.F. 2006

Universidad Nacional Autónoma de México

Biblioteca Central

Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

ÍNDICE

Índice 1

INTRODUCCION 5

Antecedentes 5
Definición del problema 5
Objetivo 6
Metodología 6

CAPÍTULO 1. ANÁLISIS DE LOS REQUERIMIENTOS DEL SISTEMA 9
1.1 Antecedentes de los PLC 9
1.1.1 Definición de un PLC 9
1.1.2 Evolución al PLC actual 9
1.1.3 Ventajas de los PLC 10
1.1.4 Desventajas de los PLC 13
1.2 Conjunto de instrucciones 13
1.2.1 Programación 18
1.2.2 Definición de los principales tipos de bit 19
1.2.3 Instrucciones boolenas 20
1.2.4 Flancos ascendente y descendente 21
1.2.5 Instrucciones de carga 21
1.2.6 Instrucciones de asignación 22
1.2.7 Utilización de paréntesis 25
1.2.8 Funciones de bloque 27
1.3 Elementos que constituyen el sistema 38
1.3.1 Software 38
1.3.2 Hardware 38

1.3.3 Descripción de Java 38

CAPÍTULO 2. DISEÑO DEL SISTEMA 43
2.1 Lenguajes y gramáticas 43
2.1.1 Jerarquía de lenguajes 53
2.2 Autómatas 56
2.2.1 De gramáticas a máquinas computacionales 57
2.2.2 De máquinas computacionales a Programas 62
2.3 La arquitectura del compilador 63
2.4 Elementos del compilador 65
2.4.1 Identificación de elementos: análisis léxico 66
2.4.2 Estructura del lenguaje: análisis sintáctico 67
2.4.3 Del análisis al código 71

CAPÍTULO 3. DESARROLLO DEL SISTEMA 73
3.1 Estructura del analizador léxico 73
3.1.1 Simplificaciones para el analizador léxico 75
3.2 Estructura del analizador sintáctico 76
3.2.1 Simplificaciones para el analizador sintáctico 79
3.3 La estructura del intérprete 81
3.4 Estructuras de datos para el procesador de lenguajes 82
3.5 Proceso de compilación 91

CAPÍTULO 4. PRUEBAS DEL SISTEMA Y MANTENIMIENTO 97
4.1 Pruebas de caja negra 98
4.1.1 Limitaciones de las pruebas de caja negra 99
4.2 Pruebas de caja blanca 99
4.2.1 Limitaciones de las pruebas de caja blanca 99
4.3 Pruebas unitarias 99

4.4 Pruebas de integración 100
4.5 Pruebas de regresión 100
4.6 Consideraciones para la ejecución de las pruebas 100
4.7 Pruebas aplicadas 101
4.8 Mantenimiento 102

CONCLUSIONES 105

GLOSARIO 107

BIBLIOGRAFÍA 111

INTRODUCCIÓN

Antecedentes

En la vida actual y específicamente en el campo de la tecnología, la automatización ha alcanzado un nivel de empleo importantísimo de tal manera que en el desarrollo tecnológico no se puede prescindir de ella, dado que la automatización reduce significativamente tanto las horas hombre y los tiempos de manufactura como el material empleado, reduciendo los problemas y garantizando al final un producto de mayor calidad.

En el campo de la ingeniería, la automatización es fundamental porque determina procesos como los que realizan los sistemas de control y los PLC. Un PLC (acrónimo del inglés Programmable Logic Controller) es un dispositivo que fue inventado para reemplazar los circuitos con relevadores y temporizadores para el control de máquinas en diversos procesos. Con este dispositivo los cambios en el cableado son menores, porque las interconexiones, los temporizadores y contadores son implementados por software. Los sistemas PLC ayudan a reducir los tiempos en los procesos en los que son utilizados, trabajan de acuerdo a entradas y dependiendo del estado de éstas, se encienden o apagan sus salidas. Al PLC se le introduce un programa que ayuda a obtener los resultados deseados.

Definición del problema

El Laboratorio de Electrónica del CCADET tiene como una de sus líneas de investigación el desarrollo de sistemas electrónicos acorde con las necesidades del país; por lo que ha diseñado dentro de sus instalaciones un sistema PLC, el cual es programado mediante instrucciones que se envían desde la HyperTerminal del sistema operativo Windows a una memoria del dispositivo. Este sistema requiere de una persona experta en la programación del mismo.

Debido a que los procesos de automatización actuales requieren máquinas de fácil programación por usuarios no expertos en electrónica, se tiene la necesidad de un programa compilador que facilite esta tarea.

El modo más empleado actualmente para programar un PLC es mediante una computadora tipo PC. Esto supone herramientas más potentes: simulación, facilidad de programación, posibilidad de almacenamiento, impresión, transferencia de datos, monitorización, etc.

Con la ayuda del programa compilador, objeto de este ejercicio, la programación será más eficiente ya que antes de cargar el programa en memoria, se realizan varios pasos de análisis del código proporcionado por el usuario, asegurando con esto, que el código es correcto y que la máquina no tendrá comportamientos indeseados; se trabaja, además, en un ambiente propio, y se mejora el trabajo del programador ya que puede almacenar los archivos que se van cargando al PLC para posteriormente realizar análisis de los procesos que se efectúan e introducir cambios o mejoras al código.

Objetivo

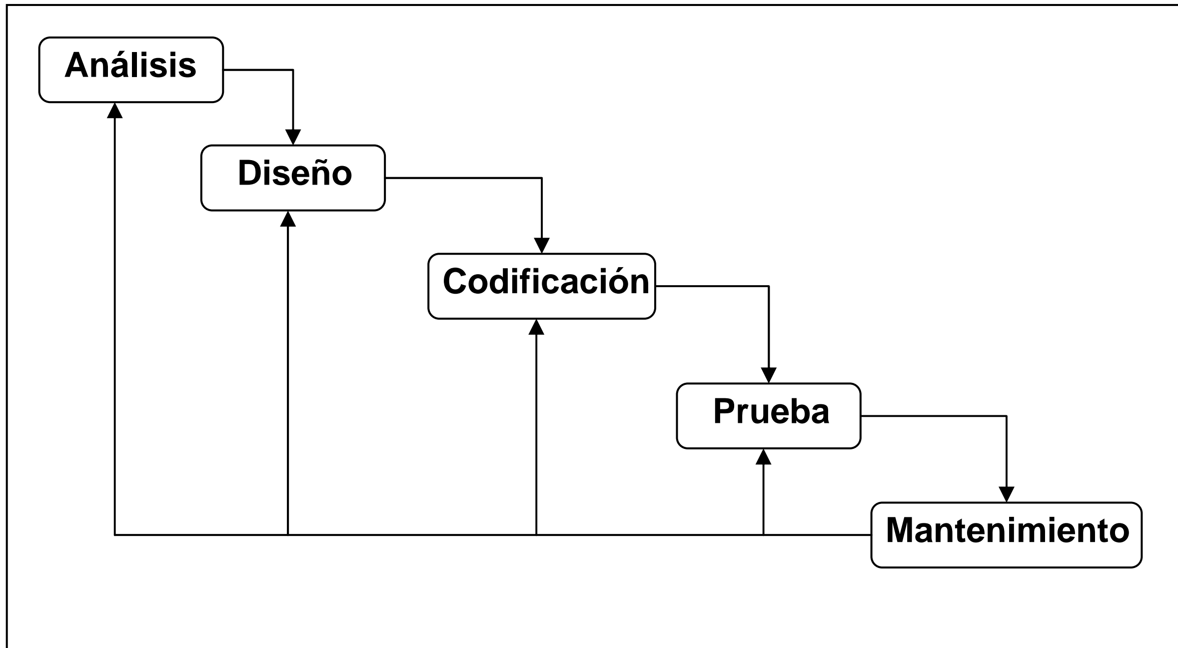
Contar con una aplicación que facilite el proceso de programación del PLC diseñado por el Laboratorio de Electrónica del CCADET.

Metodología

El método que se utilizado es el modelo conocido en la Ingeniería de Software como Ciclo de vida clásico o Modelo de cascada.

Este modelo especifica qué pasos y en qué orden deben llevarse a cabo cada una de las actividades para que el sistema se desarrolle de manera adecuada y en un corto lapso de tiempo.

El modelo de ciclo de vida clásico se muestra en la siguiente figura:



Modelo de cascada.

Etapa 1. Análisis

Recabar información sobre los PLC y su programación, que permita conocer a fondo las características y el funcionamiento que el sistema debe tener.

Analizar el PLC del Laboratorio de Electrónica y su forma de programación.

Analizar la interfaz requerida.

Analizar el software y el hardware que se utilizarán.

Recabar información sobre la construcción de compiladores.

Etapa 2. Diseño

Diseñar el conjunto de instrucciones.

Diseñar la gramática del lenguaje que se usará.

Diseñar la estructura general del compilador en base a los requerimientos encontrados en la Etapa 1.

Diseñar los elementos del compilador.

Diseño del analizador léxico.

Diseño del analizador sintáctico.

Diseño del intérprete.

Diseño de la interfaz.

Documentar el diseño del compilador.

Etapa 3. Codificación

Generar la codificación de la interfaz.

Generar la codificación del analizador léxico.

Generar la codificación del analizador sintáctico.

Generar la codificación del intérprete.

Etapa 4. Pruebas

Realizar pruebas del código generado por el PLC.

Generar la documentación pertinente.

Etapa 5. Mantenimiento

Realizar cambios, bien por errores que no se hayan detectado antes, por cambios en el entorno o por ampliaciones a petición de los usuarios. Se llevan a cabo tres tipos de mantenimiento:

1. Mantenimiento correctivo.
2. Mantenimiento adaptativo.
3. Mantenimiento perfectivo.

CAPÍTULO 1. ANÁLISIS DE LOS REQUERIMIENTOS DEL SISTEMA

1.1 Antecedentes de los PLC

1.1.1 Definición de un PLC

Un PLC (Controlador Lógico Programable, por sus siglas en inglés) es una computadora electrónica de uso amigable que realiza funciones de control de muchos tipos y niveles de profundidad. Puede ser programado, controlado y operado por una persona inexperta en PLCs, pero con conocimientos básicos de operación de computadoras. En el PLC básicamente se dibujan las líneas y los dispositivos de diagramas de escalera. El dibujo resultante se transforma dentro de la computadora en el equivalente del cableado requerido para controlar un proceso. El PLC opera cualquier tipo de sistemas que tengan como salida de los dispositivos interruptores de encendido y apagado, así como salidas variables. En el lado de la entrada también pueden presentarse entradas de encendido/apagado o entradas variables.

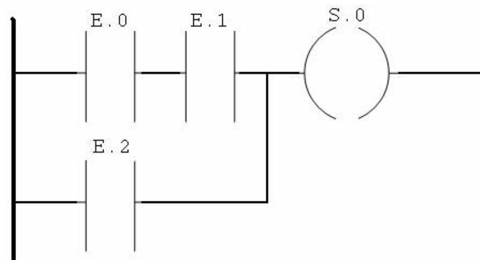


Figura 1.1.
Ejemplo de diagrama de escalera.

1.1.2 Evolución al PLC actual

Los primeros sistemas PLC evolucionaron de las computadoras convencionales de finales de los 60s y principios de los 70s. Estos sistemas fueron instalados en su mayoría en plantas automotrices. Las plantas de autos debían detenerse cerca de un mes mientras se hacían los cambios concernientes a los nuevos modelos. Los primeros PLC fueron usados

junto con otras técnicas de automatización para acortar el tiempo de reajuste. Uno de los procesos de mayor consumo de tiempo en el cambio era el alambrado nuevo o la revisión de los relevadores y los páneles de control. El procedimiento de reprogramación vía teclado del PLC reemplazó el cableado de los páneles llenos de cables, relevadores, temporizadores y otros componentes. Los nuevos PLC ayudaron a reducir el tiempo de reprogramación a unos cuantos días.

Había un gran problema con los procedimientos de programación de las computadoras/PLC de los 70s. Los programas eran complicados y se requería de un programador entrenado para hacer los cambios. A lo largo de los 70s se hicieron cambios a los programas de los PLC para hacerlos más amigables; en 1978, con la introducción de los microprocesadores a los PLC se incrementó su poder de cómputo y se redujeron los costos. Como consecuencia de esto, los PLC tuvieron muchas mejoras, los programas llegaron a ser más entendibles, y aumentó su capacidad.

En los 80s y 90s, con mayor poder de cómputo por menos precio, los PLC incrementaron exponencialmente su uso. Los PLC llegaron a ser el más grande volumen de producción de algunas de las grandes empresas de electrónica y computadoras.

Actualmente los PLC son usados en una amplia variedad de sistemas de automatización, que van desde generación de energía hasta usos en casa habitación y equipo médico. El uso de estos sistemas continúa creciendo.

1.1.3 Ventajas de los PLC

Las siguientes son algunas ventajas de usar un controlador lógico programable:

Flexibilidad. En el pasado, cada máquina de producción controlada electrónicamente requería su propio controlador. 15 máquinas requerían 15 diferentes controladores. Ahora es posible usar sólo un PLC para correr una o varias máquinas. Además, se requerirán menos de 15 controladores porque un PLC puede fácilmente operar

varias máquinas. Cada una de las 15 máquinas bajo control del PLC debe tener su propio programa.

Menor tiempo en implementación de cambios y corrección de errores. Con un panel cableado tipo relevador, cualquier alteración del programa requiere tiempo para volver a cablear el panel y los dispositivos. Cuando se hace un cambio en el circuito del programa o en la secuencia de diseño, el programa del PLC puede hacerlo en unos cuantos minutos. No se necesita volver a cablear en un sistema controlado por PLC. También, si un error de programación debe corregirse en el diagrama de escalera de control del PLC, puede ser escrito rápidamente.

Cantidades grandes de contactos. El PLC tiene una gran cantidad de contactos para cada bobina disponible en su programación. Consideremos que un panel cableado con relevadores tiene cuatro contactos y todos están siendo usados cuando un cambio en el diseño requiere tres contactos más. Esto significaría tiempo para instalar nuevos relevadores. Usando sistemas PLC, sin embargo, requerirá simplemente escribir en el programa tres contactos más. Los contactos estarán disponibles automáticamente en el PLC. Pueden usarse mil contactos de un relevador, sólo si hay memoria disponible.

Bajo costo. La tecnología ha hecho posible compactar más funciones dentro de paquetes más pequeños y menos caros. Actualmente es posible conseguir un PLC con varios relevadores, temporizadores, contadores y otras funciones por un precio menor a los primeros dispositivos PLC.

Simulación. El programa de un circuito de PLC puede ser simulado y evaluado previamente. El programa puede escribirse, probarse, observarse y modificarse si es necesario, ahorrando un considerable tiempo de fábrica. En contraste, los sistemas de relevadores deben ser probados en la fábrica, lo cual puede consumir mucho tiempo de producción.

Observación. La operación correcta o incorrecta de un circuito de PLC puede observarse durante la operación en una pantalla mientras sucede. Las rutas lógicas se encienden en la pantalla mientras son energizadas. Los problemas pueden corregirse más rápido durante la observación.

Velocidad de operación. Los relevadores toman en ocasiones cantidades de tiempo de reacción para actuar que podrían resultar inaceptables. La velocidad de operación del PLC es muy rápida y está determinada por su tiempo de procesamiento, el cual es del orden de milisegundos.

Método de programación booleano o de escalera. La programación del PLC puede realizarse en modo escalera por un técnico electricista. De manera alternativa, puede programarse de manera booleana o digital.

Confianza. Los dispositivos de estado sólido son más confiables, en general, que los mecánicos o los relevadores eléctricos.

Seguridad. Un cambio en los programas PLC no puede hacerse a menos que el PLC esté propiamente en fase de programación. Los páneles con relevadores tienden a someterse a cambios no documentados por descuido, cuando la gente del área de trabajo no documenta los cambios al terminar la jornada de trabajo.

Facilidad en los cambios en la programación. Ya que el PLC puede programarse rápidamente, el procesamiento mixto de la producción puede ser perfecto. Por ejemplo, si la parte B baja a la línea de ensamblado mientras la parte A está siendo procesada aún, un programa para el procesamiento de la parte B puede ser reprogramado en la máquina de producción en unos cuantos segundos.

Estas características son algunas de las ventajas de usar controladores lógicos programables. Hay, por supuesto, otras ventajas en aplicaciones individuales o industriales.

1.1.4 Desventajas de los PLC

Las siguientes son algunas desventajas, o más bien precauciones, al usar PLCs.

Programas de accesorio. Algunas aplicaciones son funciones sencillas. No es necesario pagar un PLC que incluye múltiples capacidades de programación si no son necesarias. Un ejemplo es el uso de tambores controladores/secuenciadores. Algunos equipos de manufactura aun usan un tambor mecánico de pasos con una gran ventaja en el costo. La secuencia de operación casi no cambia o nunca cambia, por lo que la programación incluida en el PLC no es necesaria.

Consideraciones ambientales. Algunos procesos en el ambiente, como las altas temperaturas o las vibraciones, interfieren en los dispositivos del PLC, lo que limita su uso.

Operación a prueba de fallas. En los sistemas de relevadores, el botón de apagado desconecta eléctricamente el circuito; si la alimentación falla, el sistema se detiene. Sin embargo, el sistema relevador no reinicia automáticamente cuando la alimentación regresa. Esto, puede programarse en un PLC; sin embargo, en algunos programas, es necesario aplicar un voltaje de entrada para lograr que el dispositivo se detenga. Estos sistemas no son a prueba de fallas. Esta desventaja puede solucionarse agregando relevadores al PLC.

Operación de circuito fijo. Si el circuito en operación no se altera nunca, un sistema de control fijo como un tambor mecánico, por ejemplo, es menos costoso que un PLC. El PLC es más efectivo cuando son necesarios cambios periódicos.

1.2 CONJUNTO DE INSTRUCCIONES

Para que el PLC realice sus diversas funciones es necesario introducir en su memoria un programa. Esta tarea se realiza mediante una PC, con un software asociado al dispositivo. La selección del lenguaje a utilizar para la programación del PLC está basada en facilitar la programación del dispositivo, lo cual es uno de los requerimientos mínimos en los sistemas PLC actuales. Por otra parte, existen ciertas características de los diferentes

lenguajes que deben tenerse en consideración para elegir el lenguaje. Existen diversas maneras de programar un PLC. Las principales son:

- El Gráfico de Funciones Secuenciales o Grafcet.
- Diagramas de bloques funcionales (DBF).
- Diagramas de contactos en escalera (Ladder).
- Texto estructurado (Java, C, Pascal, etc.).
- Lista de instrucciones (AWL).

A continuación describiremos brevemente cada uno de los diferentes lenguajes.

El ***Gráfico de Funciones Secuenciales***, o Grafcet, es un lenguaje gráfico que ofrece estructura general y coordinación a las secuencias del programa. Fue derivado de las técnicas que se usan para describir comportamiento secuencial, describiendo el comportamiento de un sistema en términos de estados y transiciones. Una secuencia en Grafcet es descrita como una serie de pasos mostrados como cajas rectangulares conectadas por líneas verticales. Cada paso representa un particular estado del sistema a controlar. Cada flecha conectora posee una barra horizontal representando una transición. Una transición es asociada con una condición, la cual, cuando resulta verdadera, causa que el estado anterior a la transición se desactive y que el que le sigue a la transición sea activado. El flujo de control es generalmente hacia abajo, sin embargo pueden ser usadas ramas para dirigirse a pasos anteriores. Las principales características del Grafcet son:

- Soporta selecciones alternativas y secuencias paralelas.
- Las etapas o estados implican acciones asociadas.
- Las transiciones gobiernan los cambios de estado.
- Las flechas indican la dirección del cambio.
- Pueden darse esquemas menos lineales.

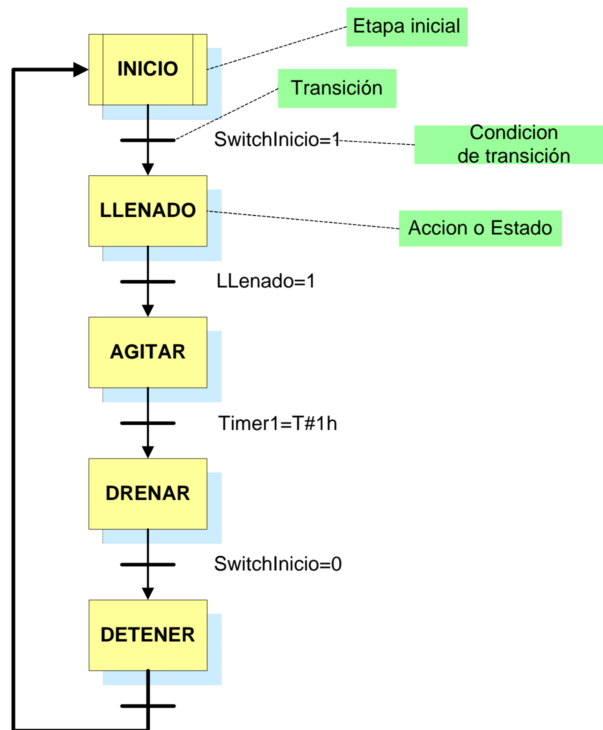


Figura 1.2.
Ejemplo de gráfico de funciones secuenciales.

Los ***Diagramas de Bloques Funcionales*** (DBF) son un lenguaje gráfico usado para construir procedimientos complejos a partir de una librería de funciones, bloques de funciones y programas así como un juego de bloques interconectados. Pueden ser usados donde el problema involucre un flujo de señales entre bloques de control. Una red de DBF puede considerarse análoga a un diagrama de un circuito eléctrico dónde las conexiones eléctricas describen trayectorias entre componentes. Los usos típicos de los DBF incluyen la descripción de lazos de control y lógica booleana.

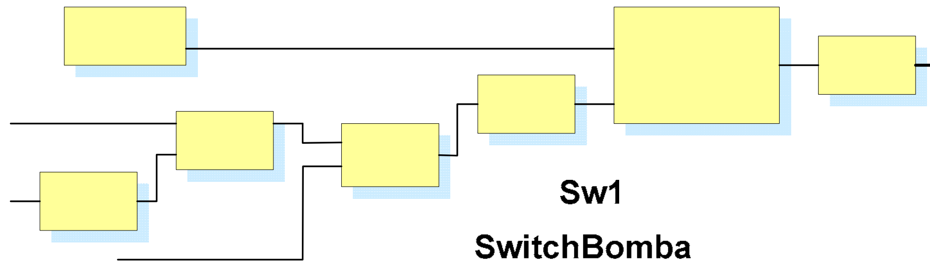


Figura 1.3.

Ejemplo de diagrama de bloques secuenciales.

El ***Diagrama de Contactos*** (LADDER) es un excelente lenguaje gráfico para lógicas discretas. También tiene la habilidad de incluir instrucciones de bloques funcionales dentro de una línea. El lenguaje de diagrama de contactos es considerado el más comúnmente usado para la programación de los PLC, está basado en el diseño lógico con relevadores. Un diagrama de escalera o ladder siempre tiene a su izquierda la barra de alimentación que provee de energía a los dispositivos como se distribuyan horizontalmente hacia la derecha. Dichos dispositivos pueden ser contactos, interruptores, bobinas etc. donde se tienen símbolos asociados a estos.

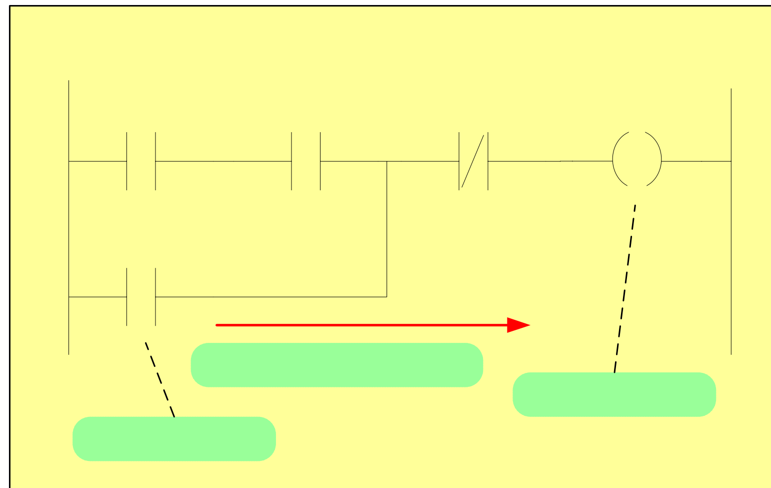


Figura 1.4.

Ejemplo de diagrama de escalera.

El ***Texto Estructurado*** (ST o TE) es un lenguaje similar a Java, Pascal o C (desde bajo hasta alto nivel) que se emplea para procedimientos complejos o cálculos que no pueden ser fácilmente implementados utilizando lenguajes gráficos. En este tipo de programación se pueden expresar funciones de diversos comportamientos, bloques de funciones y programas. El estándar define un rango de operadores tanto para operadores aritméticos como booleanos.

La ***lista de Instrucciones*** (AWL, IL o LI) es un lenguaje de bajo nivel, similar al código máquina o ensamblador, y puede ser usado para expresar el comportamiento de funciones, bloques de funciones, transiciones en diagramas de funciones secuenciales. Es útil para pequeñas aplicaciones que requieran de una rápida ejecución para optimizar el tiempo de proceso. La estructura básica de la lista de instrucciones es muy simple y fácil de aprender. El lenguaje de LI consiste en una serie de instrucciones donde cada instrucción está sobre una nueva línea. Una instrucción consiste en un operador seguido por uno o más operandos que pueden estar separados por comas. Cada instrucción puede cambiar o usar algún valor almacenado en un solo registro. El estándar refiere a este registro como el resultado de esta instrucción. El resultado puede ser sobrescrito con un nuevo valor, modificado o almacenado en alguna variable. Algunas veces a este registro donde se halla dicho resultado se le hace referencia como acumulador.

Muchos fabricantes de PLC´s ofrecen sistemas que soportan la programación de lista de instrucciones prefiriendo este sobre el texto estructurado. Para un diseñador de PLC´s, una de las principales ventajas de LI sobre el TE es que es mucho mas fácil implementar y desarrollar un PLC que pueda interpretar tanto su forma de programación gráfica así como además listas de instrucciones directamente, asimismo con algunos sistemas es posible descargar programas en LI al PLC sin compilar el programa del proceso dada su simpleza para programar además de su estandarización de la LI. Contrastando al TE que generalmente tiene que ser compilado al ensamblador nativo del microprocesador del PLC. Las Listas de instrucciones son consideradas como el lenguaje al cual todos los otros lenguajes pueden ser traducidos, aunque con limitantes tienden ser destinadas tales traducciones a pequeños PLC´s.

Considerando lo anterior, el lenguaje elegido para la programación del PLC es la lista de instrucciones. Un punto a favor de LI sobre el TE es que es mucho más fácil de implementar, sin embargo, la mayor ventaja es la característica de reversibilidad. Esta se refiere a la capacidad del compilador con que se programa el PLC para convertir en LI los programas gráficos escritos en LADDER y viceversa, aunque es necesario aclarar que no siempre es posible obtener una equivalencia exacta.

Una vez elegido el lenguaje de programación, y conociendo las funciones existentes en el dispositivo para el cual se desarrolla el compilador, definimos el conjunto de instrucciones, el cuál a grandes rasgos representa:

- las entradas/salidas del PLC (botones pulsadores, sensores, relevadores, indicadores de funcionamiento, interruptores, etc.)
- las funciones avanzadas (bloques temporizadores, bloques contadores, etc.)
- las operaciones matemáticas y lógicas (suma, división, y, o exclusiva...)
- las variables internas (bits, palabras, etc.).

A continuación se presenta con detalle el conjunto de instrucciones y la manera de realizar la programación de las diferentes tareas del PLC.

1.2.1 Programación

En los PLC el símbolo interno para cualquier entrada es un contacto. Cada terminal de entrada tiene un número en el módulo de entrada. Internamente, el PLC representa los estados de cada entrada por el estado del contacto del mismo número. Similarmente, el símbolo interno para las salidas es una bobina. También existen otros bloques funcionales para representar salidas como son los contadores o los temporizadores. Cada bobina interna tiene un número. Cuando se enciende, la salida correspondiente del mismo número se activa en el módulo de salida.

1.2.2 Definición de los principales tipos de bits

Bits de entrada (Contactos)

E.0 a E.15

Los contactos en un sistema PLC están relacionados directamente con los bits de entrada. Cada entrada al módulo tiene su correspondiente bit de entrada asociado (contacto). Sin embargo, no todos los bits en un programa interno tienen una entrada asociada. Se cuentan con 16 contactos de entrada los cuales se denotan con una E seguida de un punto y finalizando con el número de contacto que se quiere utilizar.

Supongamos que alimentamos la entrada 5. Todos los contactos programados en el PLC como E.5 cambiarán sus estados. La entrada es examinada y la acción se ejecuta. Es muy importante mencionar que la actualización de los contactos en la memoria del PLC se ejecuta en cada exploración de las entradas.

Bits de salida (Bobinas)

S.0 A S.15

Las bobinas en un programa interno en el PLC están relacionadas con las señales de salida que son enviadas a dispositivos externos. Una salida es activada o encendida en el módulo de salida cuando su correspondiente número de bobina está encendido en el diagrama de escalera del PLC.

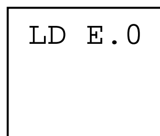
Bits de memoria

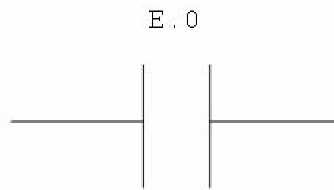
M.0 A M.63

Los bits de memoria registran los estados intermedios durante la ejecución del programa.

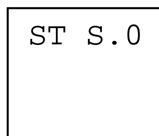
1.2.3 Instrucciones booleanas

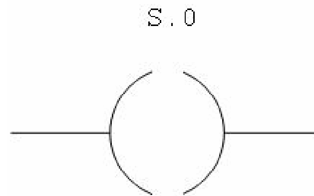
Elementos de comprobación. Ejemplo. La instrucción LD equivale a un contacto abierto y conduce cuando el objeto que lo controla se encuentra en el estado de encendido.



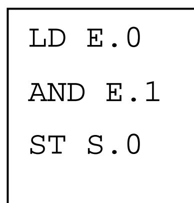


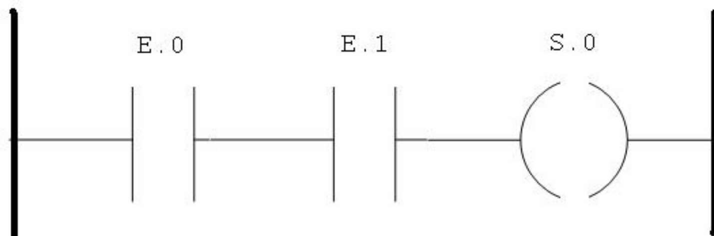
Elementos de acción. Ejemplo. La instrucción ST equivale a una bobina directa. El objeto asociado toma el valor lógico del resultado lógico del elemento de prueba.





Ecuación booleana: El resultado booleano de los elementos de comprobación se aplica al elemento de acción.





1.2.4 Flancos ascendente y descendente

Las instrucciones de comprobación permiten detectar los flancos ascendente y descendente en las entradas del autómata. Se detecta un flanco cuando el estado de una entrada ha cambiado entre el ciclo n-1 y el ciclo n en curso, y permanece detectado durante el ciclo en curso.

Flanco ascendente: detección del paso de 0 a 1 de la entrada que lo controla. La instrucción LDR (Load Rising edge) equivale a un contacto de detección de flanco ascendente:

```
LDR E.0
```

Flanco descendente: detección del paso de 1 a 0 de la entrada que lo controla. La instrucción LDF (Load Falling edge) equivale a un contacto de detección de flanco descendente:

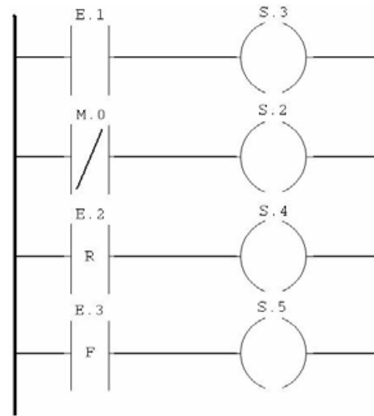
```
LDF E.0
```

1.2.5 Instrucciones de carga

Las instrucciones **LD**, **LDN**, **LDR** y **LDF** corresponden respectivamente a los contactos abierto, cerrado, de flanco ascendente y flanco descendente.

código	Operando
LD	0/1, E, M
LDN	E, M,
LDR	E
LDF	E

```
LD E.1
ST S.3
LDN M.0
ST S.2
LDR E.2
ST S.4
LDF E.3
ST S.5
```

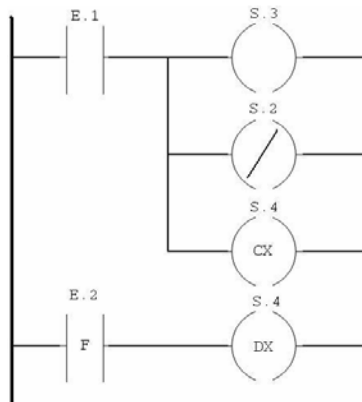


1.2.6 Instrucciones de asignación

Las instrucciones **ST**, **STN**, **BCX** y **BDX** corresponden respectivamente a las bobinas directa, inversa, conexión de bobina y desconexión de bobina.

Código	Operando
ST	S, M
STN	S, M
BCX	S, M
BDX	S, M

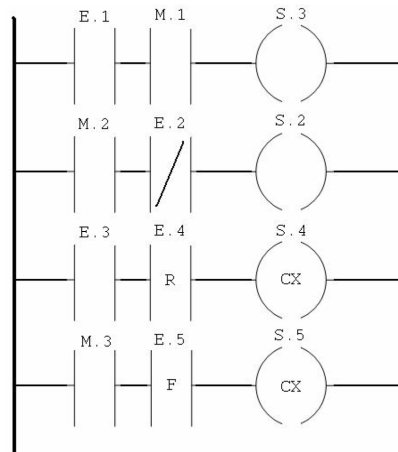
```
LD E.1
ST S.3
STN S.2
BCX S.4
LD E.2
BDX S.4
```



Instrucciones Y. Estas instrucciones realizan una Y lógica entre el operando (o su inverso, o frente ascendente o frente descendente) y el resultado booleano de la instrucción anterior.

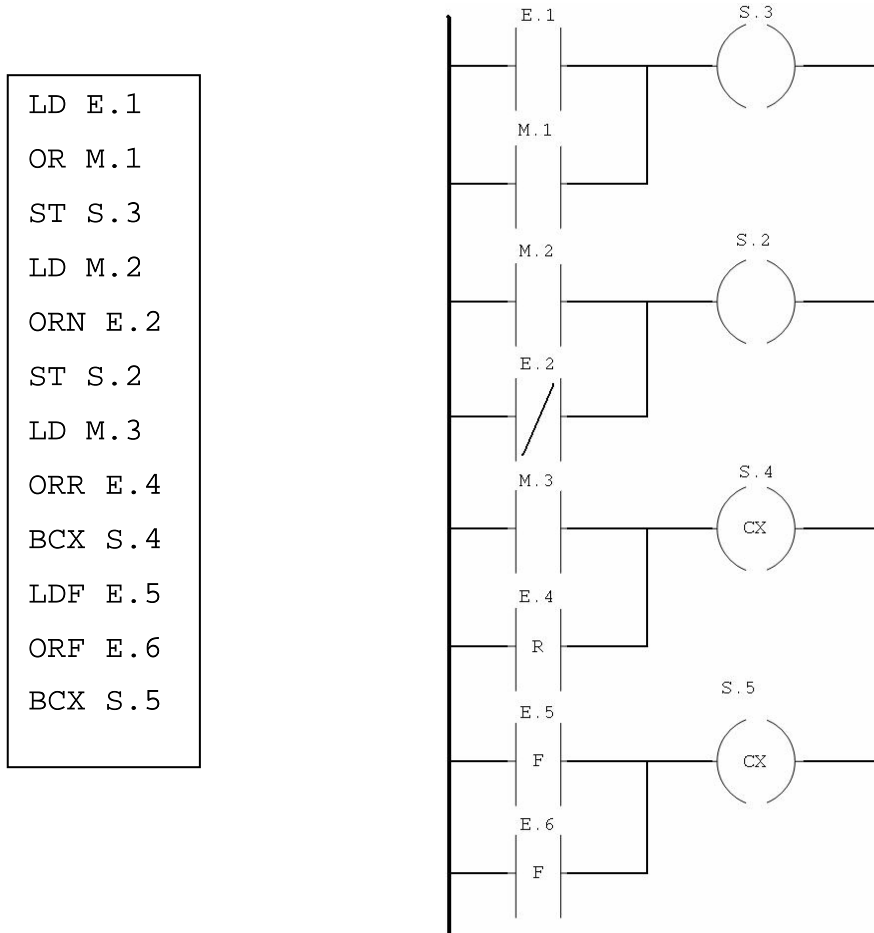
Código	Operando
AND	E,S,M
ANDN	E,S,M
ANDR	E
ANDF	E

```
LD E.1
AND M.1
ST S.3
LD M.2
ANDN
E.2
ST S.2
LD E.3
ANDR
E.4
BCX S.4
LD M.3
```



Instrucciones O. Estas instrucciones realizan un O entre el operando (o su inverso, o frente ascendente, o frente descendente) y el resultado booleano de la instrucción anterior.

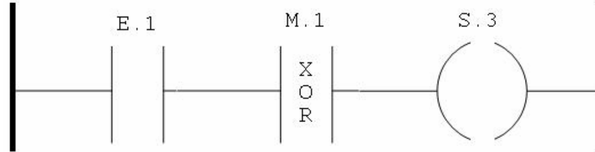
Código	Operando
OR	E,S,M
ORN	E,S,M
ORR	E
ORF	E



Instrucciones O Exclusiva. Estas instrucciones realizan un O exclusivo entre el operando (o su inverso, o frente ascendente, o frente descendente) y el resultado booleano de la instrucción anterior.

Código	Operando
XOR	E,S,M
XORN	E,S,M
XORR	E
XORF	E

```
LD E.1
XOR
M.1
```



Instrucción Negación: N. Esta instrucción realiza la negación del resultado de la instrucción anterior.

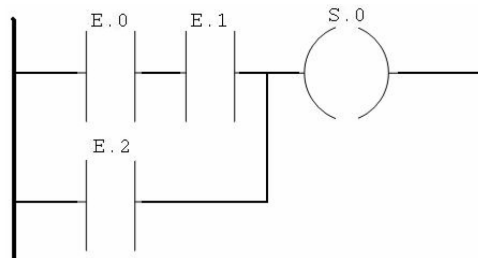
Código	Operando
N	-

```
LD E.1
OR M.2
N
AND M.3
ST S.3
```

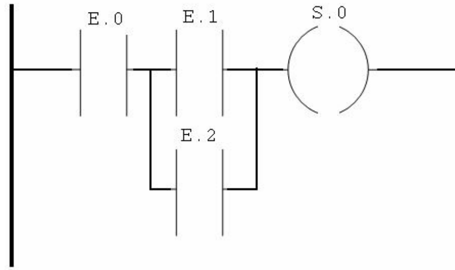
1.2.7 Utilización de paréntesis

Las instrucciones AND y OR pueden utilizar paréntesis ( ). Estos paréntesis permiten realizar esquemas de contactos de forma sencilla. El signo de abertura de paréntesis se asocia a la instrucción AND u OR. El paréntesis de cierre es una instrucción que es obligatoria para cada paréntesis abierto. Nota: A los paréntesis no se les puede asociar la instrucción negación.

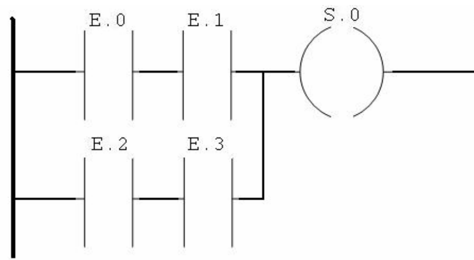
```
LD E.0
AND E.1
OR E.2
ST S.0
```



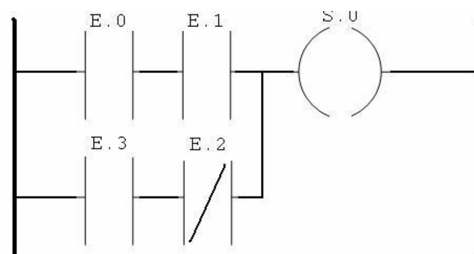
```
LD E.0
AND( E.1
OR E.2
)
ST S.0
```



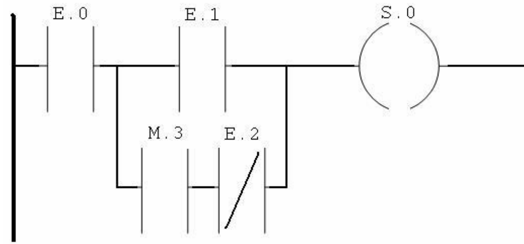

```
LD E.0
AND E.1
OR( E.2
AND E.3
)
ST S.0
```



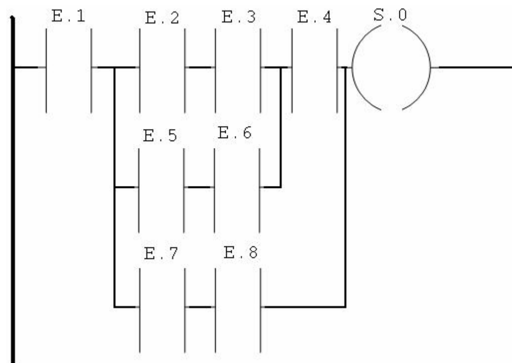

```
LD E.0
AND E.1
OR( E.3
ANDN
E.2
)
```



```
LD E.0
AND(
E.1
OR( M.3
ANDN
E.2
)
```




```
LD E.1
AND(
E.2
AND E.3
OR( E.5
AND E.6
)
AND E.4
OR( E.7
AND E.8
))
```



1.2.8 Funciones de bloque

BLK. Señala el inicio del bloque y define el comienzo del circuito y el inicio de la porción de entrada en el bloque.

SLD_BLK. Señala el inicio de la porción de salida del bloque.

FIN_BLK. Señala el fin del bloque y del circuito

Se definen 6 tipos de temporizadores:

• TRCX: este tipo de temporizador permite gestionar retardos en la conexión.

• TRDX: este tipo de temporizador permite gestionar los retardos de desconexión.

• TMP: este tipo de temporizador permite elaborar un pulso de duración precisa.

• FPTV: Flip-Flop con Retardo en la Entrada

• FPTU: Flip-Flop con Retardo en la Salida

• FPTW: Flip-flop con retardo en la Entrada Memorizado

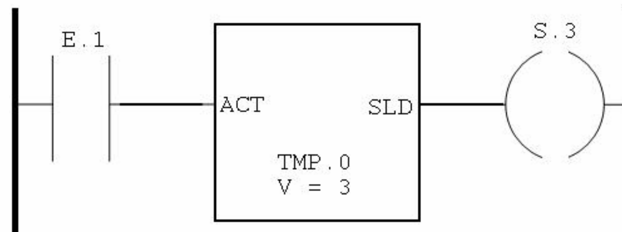
Bloque de función Temporizador TMP

Características

Número de temporizadores	**TMP.i**	De 0 a 2
Valor del retardo	**V**	La duración del retardo elaborado es igual a V segundos.
Instrucción Activación	**ACT**	En flanco ascendente arranca el temporizador.
Salida Temporizador	**SLD**	Simultáneamente a la activación del temporizador (ACT), la salida SLD es activada durante V segundos. En caso de presentarse otro flanco ascendente mientras la salida SLD esté activada, será ignorado.

Este programa indica que el bloque que se usa es el TMP.0, el valor de retardo es de 3 segundos, la entrada es por E.1, y la salida por S.3.

```
BLK TMP.0
V = 3
LD E.1
ACT
SLD_BLK
LD SLD
ST S.3
FIN_BLK
```



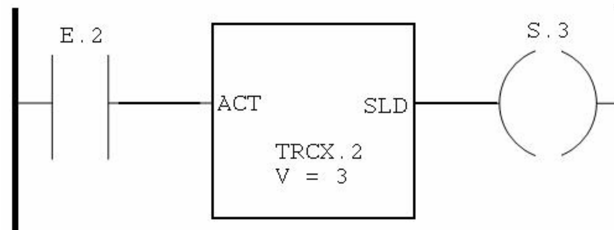
Bloque de función Temporizador con retardo en la conexión TRCX

Características

Número de temporizadores	**TRCX.i**	De 0 a 2
Valor del retardo	**V**	La duración del retardo elaborado es igual a V segundos.
Instrucción Activación	**ACT**	En flanco ascendente arranca el temporizador.
Salida Temporizador	**SLD**	Después de la activación del temporizador (ACT) se presenta un retardo de V segundos, tras el cual, es activada la salida SLD. SLD se desconecta al mismo tiempo que ACT. En caso de que la entrada ACT se desconecte antes de que sea alcanzado el valor de retardo V, SLD no es activado.

Este programa indica que el bloque que se usa es el TRCX.2, el valor de retardo es de 3 segundos, la entrada es por E.2, y la salida por S.3.

```
BLK TRCX.2
V = 3
LD E.2
ACT
SLD_BLK
LD SLD
ST S.3
FIN_BLK
```



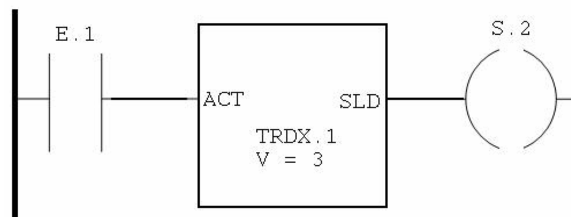
Bloque de función Temporizador con Retardo en la desconexión TRDX

Características

Número de temporizadores	**TRDX.i**	De 0 a 2
Valor de preselección	**V**	La duración del retardo elaborado es igual a V segundos.
Instrucción Activación	**ACT**	En flanco ascendente arranca el temporizador.
Salida Temporizador	**SLD**	Simultáneamente a la activación del temporizador (ACT), la salida SLD es activada. Cuando la entrada ACT es desconectada, se presenta un retardo de V segundos, tras el cual, la salida SLD también es desactivada. En caso de presentarse un flanco ascendente mientras V está alcanzándose, será ignorado y la salida SLD continuará en 1 hasta que se cumpla con el retardo especificado.

Este programa indica que el bloque que se usa es el TRDX.1, el valor de retardo es de 3 segundos, la entrada es por E.1, y la salida por S.2.

```
BLK TRDX.1
V = 3
LD E.1
ACT
SLD_BLK
LD SLD
ST S.2
FIN_BLK
```



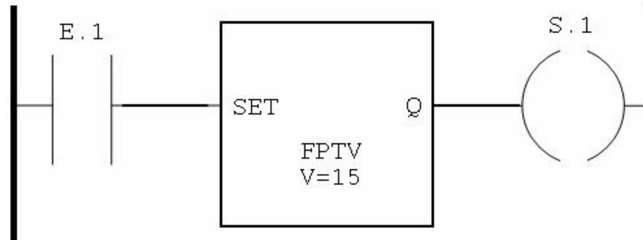
Bloque de función Flip-Flop con Retardo en la Entrada FPTV

Características

Número de temporizador	**FPTV**	Único
Valor del retardo	**V**	La duración del retardo elaborado es igual a V segundos.
Instrucción Ajuste	**SET**	Un flanco ascendente ajusta el contador a 0 e inicia el temporizador.
Salida Temporizador	**Q**	Una vez que el temporizador alcanza el valor V, la salida se activa, siempre y cuando la instrucción SET continúe habilitada.

Este programa indica que el bloque que se usa es el FPTV, el valor de retardo es de 15 segundos, la instrucción ajuste es sensada por E.1, y la salida se entrega por S.1.

```
BLK FPTV
V = 15
LD E.1
SET
SLD_BLK
LD Q
ST S.1
FIN_BLK
```



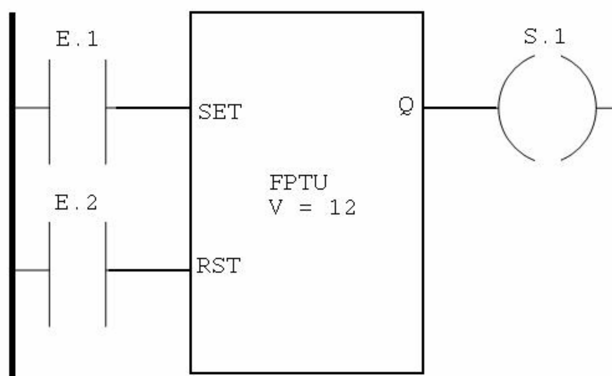
Bloque de función Flip-Flop con Retardo en la Salida FPTU

Características

Número de temporizador	**FPTU**	Único
Valor del retardo	**V**	La duración del retardo elaborado es igual a V segundos.
Instrucción Ajuste	**SET**	Ajusta el contador a 0. Después de haber transcurrido el retardo indicado, su encendido es simultáneo con la salida.
Instrucción Reiniciar	**RST**	Con un flanco descendente (apagado de la entrada) arranca el temporizador.
Salida Temporizador	**Q**	Una vez que el temporizador alcanza el valor V, la salida se activa de acuerdo a lo que indica la entrada.

Este programa indica que el bloque que se usa es el FPTU, el valor de retardo es de 12 segundos, la instrucción ajuste es sensada por E.1, la instrucción reiniciar es sensada por E.2 y la salida se entrega por S.1.

```
BLK
FPTU
V = 12
LD E.1
SET
LD E.2
RST
SLD_BLK
LD Q
ST S.1
```



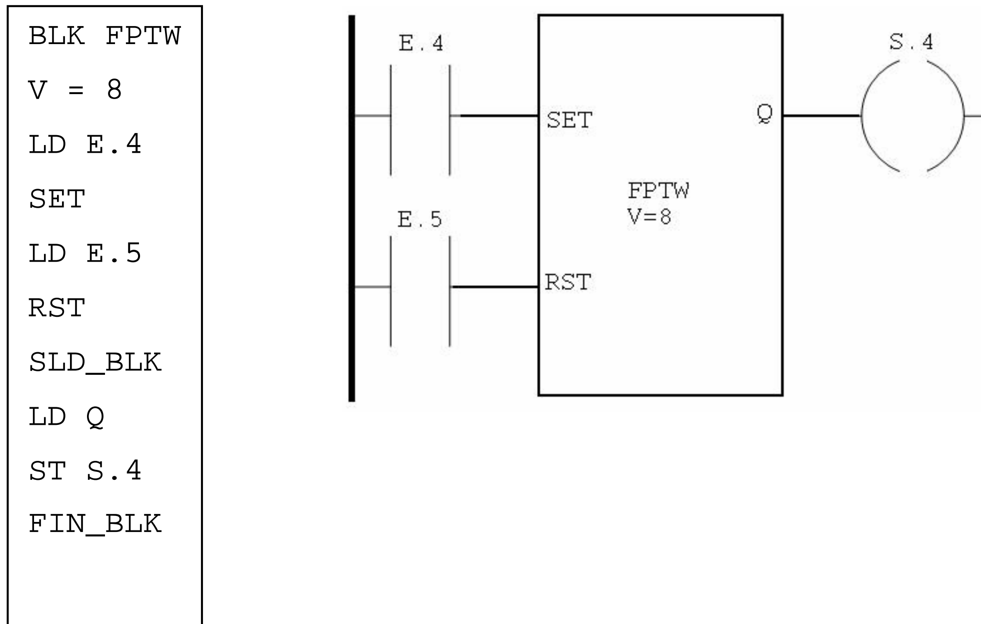
Bloque de función Flip-flop con retardo en la Entrada Memorizado FPTW

Características

Número de temporizador	**FPTW**	Único
Valor del retardo para la conexión	**V**	La duración del retardo elaborado es igual a V segundos.
Instrucción Ajuste	**SET**	Inicia la cuenta: con un flanco ascendente (encendido de la entrada) arranca el temporizador.
Instrucción Reiniciar	**RST**	Ajusta el contador a 0. Después de haber transcurrido el retardo indicado, su encendido es simultáneo con la salida.
Salida Temporizador	**Q**	Una vez que el temporizador alcanza el valor V, la salida Q se activa de acuerdo al RST.

Funcionamiento: Primero presionamos el reset para ajustar el contador a cero, el siguiente encendido del SET iniciará la cuenta. Una vez transcurrida la cuenta, la salida se comportará de acuerdo a lo que indique el RESET. Si mientras el contador trata de alcanzar el valor V, es presionado el botón RST la cuenta se reinicializa a cero y se detiene esperando un nuevo pulso en SET.

Este programa indica que el bloque que se usa es el FPTW, el valor de retardo es de 8 segundos, la instrucción ajuste es sensada por E.4, la instrucción reiniciar es sensada por E.5 y la salida se entrega por S.4.



Bloque de función Contador C

El bloque de función de contador/descontador realiza el conteo o desconteo de eventos, estas dos operaciones pueden ser simultáneas.

Número de contador	**C.i**	De 0 a 2
Valor de preselección	**V**	0<V<9999. Numero de eventos para activar la salida SPA.
Instrucción Reinicialización a 0	**R**	En estado 1 la cuenta de eventos es ajustada a 0.
Instrucción Salida Forzada	**SF**	En estado 1 la cuenta de eventos es ajustada a V y en consecuencia la salida SPA es activada.
Instrucción Conteo	**CC**	Aumenta V en flanco ascendente
Instrucción Desconteo	**DD**	Disminuye V en flanco ascendente
Salida Desbordamiento	**SDV**	SDV = 1, cuando el desconteo pasa de 0 a V (puesta a 1 cuando V es alcanzado en reversa, y de nuevo a 0 si el

(VACIO)		contador sigue descontando)
Salida Preselección alcanzada	**SPA**	El bit asociado SPA = 1, cuando se alcanza el valor V
Salida Desbordamiento (LLENO)	**SDL**	SDL =1, cuando el conteo pasa de V a 0 (puesta a 1 cuando 0 es alcanzado después pasar por V, y de nuevo a 0 si el contador continúa contando)

Funcionamiento

• Conteo: con la aparición de un flanco ascendente en la entrada de conteo CC (o activación de la instrucción CC), el valor actual aumenta en una unidad. Cuando este valor es igual al valor de preselección V, el bit de salida SPA "preselección alcanzada" asociado a la salida SPA pasa al estado 1. El bit de salida SDL (desbordamiento del conteo) pasa al estado 1 cuando se pasa de V a 0; vuelve a cero si el contador sigue contando.

• Desconteo: con la aparición de un flanco ascendente en la entrada de desconteo" DD (o activación de la instrucción DD), el valor actual disminuye en una unidad. El bit de salida SDV (desbordamiento del conteo de decrementos) pasa al estado 1 cuando se pasa de 0 a V; vuelve a 0 si el contador sigue descontando.

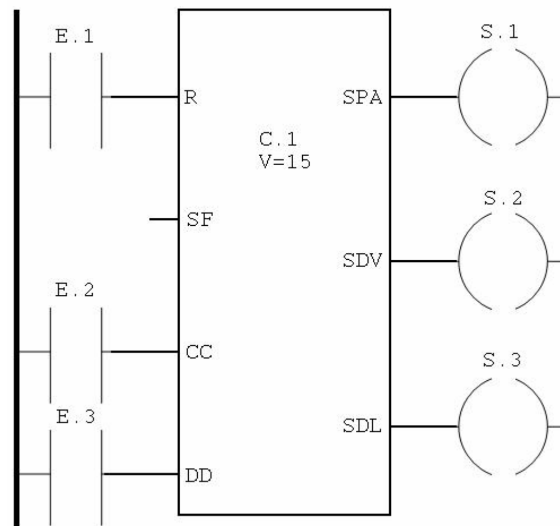
• Conteo/desconteo: Para utilizar de forma simultánea las funciones de conteo de incrementos y decrementos (o activar las instrucciones CC y DD), es necesario controlar las dos entradas correspondientes CC y DD; estas dos entradas se exploran sucesivamente. Si las dos entradas están a 1 simultáneamente, el valor actual no cambia (o si las 2 instrucciones se activan de forma simultánea).

• Puesta a cero: cuando se pone a 1 la entrada R (o se activa la instrucción), el valor V se fuerza a 0, las salidas SDV, SPA y SDL están a 0. La entrada "puesta a cero" es prioritaria.

• Salida Forzada: si la entrada SF se encuentra en el estado 1 y la entrada R "puesta a cero" en el estado 0 (o la instrucción R no activa), el valor actual de conteo se iguala a V activando la salida SPA.

Este programa indica que el bloque que se usa es el TRDX.1, el valor de retardo es de 3 segundos, la instrucción reincialización a 0, R, es introducida por E.1, el conteo se introduce por E.2 y el desconteo por E.3. En este programa no se hace uso de la instrucción SF por lo que simplemente es omitida en las instrucciones. En caso de no activarse la instrucción DD tampoco es necesario activar la instrucción SDV.

```
BLK C.1
V = 15
LD E.1
R
LD E.2
CC
LD E.3
DD
SLD_BLK
LD SPA
ST S.1
LD SDV
ST S.2
LD SDL
ST S.3
FIN_BLK
```



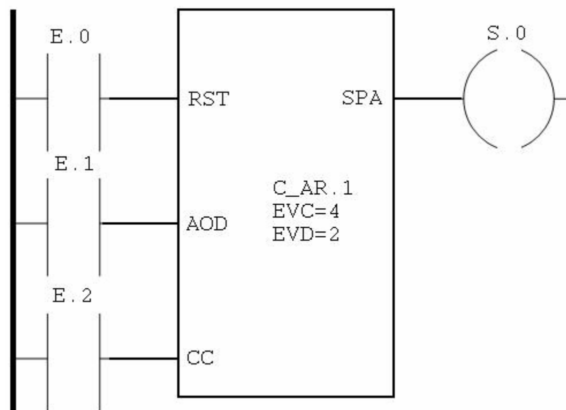
Bloque de función Contador Avance-Retroceso C_AR

El bloque de función de contador/descontador realiza el conteo o desconteo de eventos, estas dos operaciones pueden ser simultáneas.

Número de contador	**C_AR.i**	Dónde i = 0, 1 o 2
Eventos para la conexión	**EVC**	Número de eventos necesarios para habilitar SPA.
Eventos para la desconexión	**EVD**	Número de eventos necesarios para deshabilitar SPA.
Entrada RESET	**RST**	El contador es restaurado a cero
Dirección ascendente/ descendente	**AOD**	Determina si la dirección del contador es ascendente o descendente. Su valor predeterminado es AOD = 0, ascendente.
Instrucción conteo	**CC**	En flanco ascendente aumenta o disminuye el contador en una unidad de acuerdo a la dirección indicada en AOD.
Salida Preselección Alcanzada	**SPA**	Activada a partir de que el contador alcanza el valor EVC y hasta que el contador alcanza el valor EVC+EVD.

Nota: La instrucción de carga para AOD es opcional.

```
BLK C_AR.1
EVC = 4
EVD = 2
LD E.0
RST
LD E.1
AOD
LD E.2
CC
SLD_BLK
LD SPA
ST S.0
FIN_BLK
```



1.3 Elementos que constituirán el sistema

A continuación se presentan los requerimientos del equipo que se debe utilizar para trabajar sin ningún problema con el programa

1.3.1 Software

Sistema Operativo Windows 98 o superior

Java(TM) 2 Runtime Environment, Standard Edition version "1.5.0_06" (descargable de manera libre en java.sun.com).

1.3.2 Hardware

Procesador Pentium II de 450 MHz o equivalente

Disco duro 20 MB de espacio libre

Memoria RAM 128 MB

Monitor Super VGA (800 X 600) con una resolución de 256 colores

Mouse

1.3.3 Descripción de Java

La plataforma elegida para el desarrollo del software asociado al PLC fue Java debido a dos características importantes. La primera es su facilidad para trabajar con dispositivos y hacerlo además vía web. Una vez que se encuentre implementado el compilador es posible extender el programa y junto con el resto de las clases ya escritas, lograr la programación de manera remota, lo cual podría ser realizado en un proyecto

futuro. La segunda característica es su facilidad para el procesamiento de las cadenas. Java proporciona una gran cantidad de funciones que simplifican el trabajo, sobre todo en el momento de realizar la implementación de autómatas, lo que en otros lenguajes significaría escribir nuevamente las funciones necesarias.

A continuación se da una breve descripción del lenguaje de programación Java:

Java se creó como parte de un proyecto de investigación para el desarrollo de software avanzado para una amplia variedad de dispositivos de red y sistemas embebidos. La meta era diseñar una plataforma operativa sencilla, fiable, portable, distribuida y de tiempo real. Cuando se inició el proyecto, C++ era el lenguaje del momento. Pero a lo largo del tiempo, las dificultades encontradas con C++, tales como la dependencia de los apuntadores, crecieron hasta el punto en que se pensó que los problemas podrían resolverse mejor creando una plataforma de lenguaje completamente nueva. Se extrajeron decisiones de diseño y arquitectura de una amplia variedad de lenguajes. El resultado es un lenguaje que se ha mostrado ideal para desarrollar aplicaciones de usuario final seguras, distribuidas y basadas en red en un amplio rango de entornos desde los dispositivos de red embebidos hasta los sistemas de sobremesa e Internet.

Java fue diseñado para ser:

Sencillo, orientado a objetos y familiar: Sencillo, para que no requiera grandes esfuerzos de entrenamiento para los desarrolladores. Orientado a objetos, porque la tecnología de objetos se considera madura y es el enfoque más adecuado para las necesidades de los sistemas distribuidos y/o cliente/servidor. Familiar, porque aunque se rechazó C++, se mantuvo Java lo más parecido posible a C++, eliminando sus complejidades innecesarias, para facilitar la migración al nuevo lenguaje.

Robusto y seguro: Robusto, simplificando la gestión de memoria y eliminando las complejidades de la gestión explicita de punteros y aritmética de punteros del C. Seguro para que pueda operar en un entorno de red.

Independiente de la arquitectura y portable: Java está diseñado para soportar aplicaciones que serán instaladas en un entorno de red heterogéneo, con hardware y sistemas operativos diversos. Para hacer esto posible el compilador Java genera 'bytecodes', un formato de código independiente de la plataforma diseñado para transportar código eficientemente a través de múltiples plataformas de hardware y software. Es además portable en el sentido de que es rigurosamente el mismo lenguaje en todas las plataformas. El 'bytecode' es traducido a código máquina y ejecutado por la Java Virtual Machine, que es la implementación Java para cada plataforma hardware-software concreta.

Alto rendimiento: A pesar de ser interpretado, Java tiene en cuenta el rendimiento, y particularmente en las últimas versiones dispone de diversas herramientas para su optimización. Cuando se necesitan capacidades de proceso intensivas, pueden usarse llamadas a código nativo.

Interpretado, multi-hilo y dinámico: El intérprete Java puede ejecutar bytecodes en cualquier máquina que disponga de una Máquina Virtual Java (JVM). Además Java incorpora capacidades avanzadas de ejecución multi-hilo (ejecución simultánea de más de un flujo de programa) y proporciona mecanismos de carga dinámica de clases en tiempo de ejecución.

Características de Java:

- Lenguaje de propósito general.
- Lenguaje Orientado a Objetos.
- Sintaxis inspirada en la de C/C++.
- Lenguaje multiplataforma: Los programas Java se ejecutan sin variación (sin recompilar) en cualquier plataforma soportada (Windows, UNIX, Mac, etc.)
- Lenguaje interpretado: El intérprete a código máquina (dependiente de la plataforma) se llama Java Virtual Machine (JVM). El compilador produce un código intermedio independiente del sistema denominado *bytecode*.

- Lenguaje libre: Creado por SUN Microsystems, que distribuye libremente el producto base, denominado JDK (Java Development Toolkit) o actualmente J2SE (Java 2 Standard Edition).
- API distribuida con el J2SE muy amplia. Código fuente de la API disponible.

¿Qué incluye el J2SE (Java 2 Standard Edition) ?

- Herramientas para generar programas Java. Compilador, depurador, herramienta para documentación, etc.
- La JVM, necesaria para ejecutar programas Java.
- La API de Java (jerarquía de clases).
- Código fuente de la API (Opcional).
- Documentación.

¿Qué es el JRE (Java Runtime Environment) ? JRE es el entorno mínimo para ejecutar programas Java 2. Incluye la JVM y la API. Está incluida en el J2SE aunque puede descargarse e instalarse separadamente. En aquellos sistemas donde se vayan a ejecutar programas Java, pero no compilarlos, el JRE es suficiente. El JRE incluye el Java Plug-in, que es el 'añadido' que necesitan lo navegadores (Explorer o Netscape) para poder ejecutar programas Java 2. Es decir que instalando el JRE se tiene soporte completo Java 2, tanto para aplicaciones normales (denominadas 'standalone') como para Applets (programas Java que se ejecutan en una página Web, cuando son accedidos desde un navegador).

CAPÍTULO 2. DISEÑO DEL SISTEMA

2.1 LENGUAJE Y GRAMÁTICA

Los lenguajes no necesariamente consisten de palabras y sentencias en el sentido que estamos acostumbrados. En un lenguaje, una frase puede ser una secuencia de los carácteres contenidos en el alfabeto del lenguaje. Además, los carácteres deben ser ordenados de acuerdo a las reglas definidas por la gramática que define al lenguaje. Una cadena de carácteres compuesta de esta manera se llama una **sentencia** del lenguaje.

Por ejemplo, las siguientes cadenas
abc
aabbcc
aaabbbccc
aaaabbbbcccc
aaaaabbbbbccccc
son parte del lenguaje definido de la siguiente manera: *el conjunto de cadenas consistente de los caracteres a, b, y c tales que el número de **a**s, **b**s y **c**s es igual, el número de **a**s, **b**s y **c**s va de 1 a 5, y las **a**s aparecen primero, las **b**s después y las **c**s al final.*

Consideremos otro lenguaje cuyas cadenas también consistan de las letras a, b y c. Estas letras forman el *alfabeto* del lenguaje. Las cadenas en cualquier lenguaje son escritas a partir del alfabeto. El alfabeto del lenguaje se denota por el símbolo especial Σ. Σ es el símbolo usado para representar el conjunto de símbolos desde el cuál se producen las cadenas de un lenguaje. Para el lenguaje abc, el alfabeto o Σ consiste de las letras **a**, **b** y **c**.

a, **b** y **c** son las sentencias más simples que pueden producirse por este alfabeto. Otras sentencias más complejas pueden crearse a partir de combinaciones con la operación de *concatenación* de cadenas. La concatenación crea nuevas sentencias juntando el final de una sentencia con el inicio de otra. Las cadenas **a** y **b** pueden concatenarse para crear una

sentencia nueva **ab**. Esta sentencia puede concatenarse con ella misma para crear otra sentencia, **abab**.

Con la concatenación tenemos una nueva manera de producir sentencias nuevas. De hecho la concatenación puede usarse para crear un número infinito de sentencias a partir del alfabeto. Algunos ejemplos son: ab, aa, bb, cc, abc, abca, aabbcc, aaabbc.

El conjunto de todas las posibles cadenas que pueden formarse a partir del alfabeto se denota por el símbolo $\Sigma+$. Una cadena especial llamada *cadena vacía* se representa por el símbolo ε. Esta cadena representa la inexistencia de cadenas. Se comporta como el 0 en la suma y como el 1 en la multiplicación. La cadena vacía es el elemento identidad para las cadenas. Concatenando la cadena especial ε con cualquier otra cadena se produce la cadena original. $a\varepsilon = a$.

Agregando ε a $\Sigma+$ se produce un nuevo conjunto de cadenas denotado por el símbolo Σ^*. Σ^* es el conjunto de todas las cadenas que pueden producirse por algún alfabeto q y la cadena vacía ε. Un lenguaje es un subconjunto del conjunto de cadenas en Σ^*.

Decir que un lenguaje es un subconjunto de Σ^* solamente nos dice de dónde vienen las sentencias del lenguaje. Esta definición es demasiado general para tener un uso práctico. Una manera posible de hacer la definición más específica es describir cadenas de Σ^* que pertenezcan al lenguaje usando alguna regla de definición del lenguaje. El problema con este enfoque es que la descripción de las cadenas puede llegar a ser muy complicada.

Una manera más eficiente de describir el subconjunto de Σ^* que forma el lenguaje es usando una *gramática*. Una gramática está hecha de reglas que especifican como se forman las sentencias en un lenguaje.

La notación para describir lenguajes en las gramáticas es un metalenguaje llamado Forma Extendida de Backus-Naur (FEBN). Dos elementos de la FEBN son los *terminales* y

los *no terminales*. Estos elementos representan clases de los símbolos usados en la gramática FEBN. El conjunto de terminales es solamente otro nombre para el conjunto Σ. Los símbolos terminales son los únicos que pueden aparecer en las sentencias del lenguaje.

Los símbolos no terminales se distinguen de los terminales en que nunca aparecerán en una sentencia del lenguaje y que son usados para representar clases de elementos en el lenguaje. Algunos de los elementos en el Español, por ejemplo, son los sustantivos y los verbos, a los cuales podríamos denotar como **S** y **V** respectivamente.

En la notación FEBN, los nombres de los no terminales siempre inician con una letra mayúscula. Los terminales siempre inician con una letra minúscula. Cuando es necesario especificar un símbolo como no terminal, se encierra en comillas simples (‘).

Las reglas para una gramática FEBN consisten de tres partes: un lado izquierdo, un operador de reescritura y un lado derecho. La flecha es el operador de reescritura. Una regla de producción en FEBN se escribe de la siguiente manera:

```
Lado izquierdo → Lado derecho
```

Este tipo de regla se conoce como regla de producción o de reescritura. Se lee usualmente como “El Lado izquierdo puede reescribirse como el Lado derecho”. Tanto el lado izquierdo como el lado derecho pueden ser secuencias de terminales y no terminales, aunque para simplicidad, el lado izquierdo sólo contendrá un nombre no terminal. Esta restricción hace la gramática más fácil de procesar. Esto también restringe los tipos de sentencias en un lenguaje. Sin embargo, para los lenguajes de programación, la restricción no impone ninguna limitación en los tipos de sentencias que se puedan definir. Con esta aclaración, la forma general de las reglas de las gramáticas es:

```
No terminal → secuencia de terminales y no terminales
```

Existen otros elementos de la FEBN que simplifican la escritura de gramáticas para los lenguajes. Supongamos que se tienen varias producciones en cuyo lado izquierdo se tiene lo mismo

```
A → S
A → T
A → Q
```

El mismo conjunto de reglas de producción puede expresarse usando el símbolo |

```
A → S | T | Q
```

lo cual se lee como "A puede reescribirse como S o T o Q".

Para especificar la elección de uno o mas elementos en una gramática FEBN, el elemento se encierra en paréntesis. La regla de producción

```
A → S(T | Q)
```

es equivalente a las reglas de producción

```
A → ST
A → SQ
```

También es típico que las sentencias de un lenguaje se repitan. Por ejemplo, una lista de variables tiene una secuencia de nombres de variables repetidos. Una gramática para una lista de este tipo sería:

```
Lista de variables → Nombre de variable
Lista de variables → Nombre de variable ',' Lista de variables
```

Un conjunto equivalente de reglas de producción usando la notación de corchetes en FEBN es:

```
Lista de variables → Nombre de variable{‘,’ Lista de variables}
```

Esta regla de producción se lee como “Una lista de variables puede reescribirse como un nombre de variable seguido de cero o más ocurrencias de una coma seguida de un nombre de variable.

A continuación se usa la FEBN para definir la gramática para un subconjunto muy pequeño del Español.

```
(1)   SNT → PS PV
(2)   PS → Art S
(3)   PS → S
(4)   PV → V PS
(5)   Art → el
(6)   Art → la
(7)   S → niño
(8)   S → niña
(9)   S → gato
(10)  S → pelota
(11)  S → comida
(12)  V → juega
(13)  V → come
```

Es posible separar esta gramática en varias partes. El conjunto de los no terminales es:

```
N = { SNT, PS, PV, Art, S, V }
```

El conjunto de los terminales es:

```
T = { el, la, niño, niña, gato, pelota, comida, juega, come}
```

Algunas de las sentencias que pueden producirse usando esta gramática son:

```
(S1) el niño come galletas
(S2) la niña come galletas
(S3) el niño juega la pelota
(S4) la niña juega la pelota
```

Pueden producirse muchas sentencias con esta gramática. Algunas que incluso no tienen significado. Para saber si una sentencia es parte de un lenguaje es necesario *derivar* la sentencia usando la gramática que define el lenguaje. Derivar sentencias en un lenguaje es un procedimiento en el que se usan las reglas de producción de la gramática para crear cada paso de la derivación. La derivación de la sentencia S1 se muestra a continuación:

```
SNT   → PS PV                ( regla 1 )
      → Art S PV             ( regla 2 )
      → el S PV              ( regla 5 )
      → el niño PV           ( regla 7 )
      → el niño V PS         ( regla 4 )
      → el niño come PS      ( regla 13 )
      → el niño come galletas ( regla 11 )
```

El símbolo → se lee “deriva en”. En cada paso de la derivación un símbolo no terminal se reemplaza. En el primer paso de la derivación, el símbolo SNT es reemplazado por los símbolos PS y PV. El símbolo SNT es un símbolo no terminal especial de la gramática llamado *símbolo de inicio*. Solo las reglas de producción que tienen al símbolo de inicio en su lado izquierdo pueden iniciar una derivación. El símbolo de inicio es identificado como parte de la definición de la gramática.

En el paso dos de la derivación el símbolo PS se reescribe como los símbolos Art y S usando la regla de producción 2. El proceso de reemplazar un símbolo en cada paso de la derivación continúa hasta que no haya mas terminales para reemplazar. El último paso de la derivación contiene únicamente terminales.

Una derivación puede escribirse de otra manera, conocida como árbol de parseo o árbol creador de frases. El árbol para la derivación de la sentencia “el niño come galletas” se muestra en la figura 2.1.

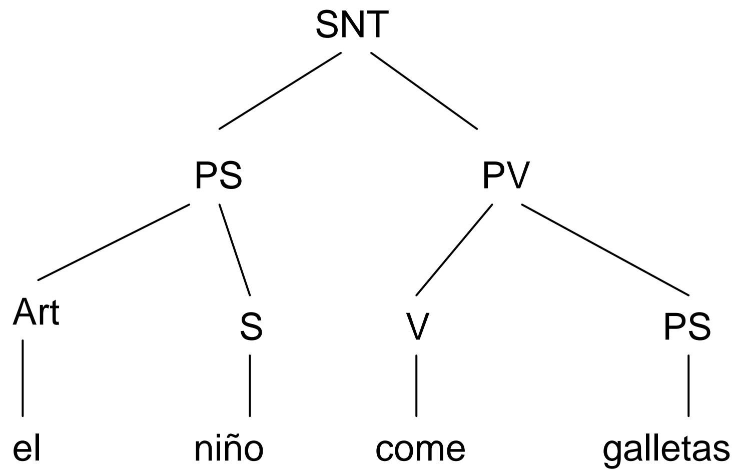


Fig. 2.1.
Árbol de análisis para la sentencia “el niño come galletas”.

La raíz del árbol siempre es el símbolo de inicio de la gramática. Los hijos o descendientes de un nodo en el árbol son los símbolos del lado derecho de la producción usada en un paso del proceso de derivación. El árbol de derivación completo se construye extendiéndolo con los hijos del lado derecho de las reglas de producción usadas en cada derivación. El proceso termina cuando todas las hojas del árbol son símbolos terminales. Estos símbolos no tienen descendientes, ya que no hay reglas de producción en las cuales un símbolo terminal sea el lado izquierdo de una regla de producción.

Es importante saber si una sentencia pertenece a un lenguaje definido por una gramática. La sentencia pertenece cuando es posible construir un árbol de derivación y cuando todas sus hojas son símbolos terminales. Si, por otra parte, cuando menos una hoja del árbol es un símbolo no terminal, la derivación no está completa y la sentencia no es parte del lenguaje definido por la gramática.

Las gramáticas son escritas observando regularidades y patrones en la sentencias del lenguaje. Las clases de componentes son nombradas y estos nombres se vuelven el conjunto de los no terminales. De esta manera y considerando las operaciones explicadas en el conjunto de instrucciones del capítulo anterior podemos definir la gramática de nuestro lenguaje de la siguiente manera:

```
PROGRAMA_PLC ->   SENTENCIA_PLC {SENTENCIA_PLC}

SENTENCIA_PLC ->  INSTRUCCIONES | BLOQUE

INSTRUCCIONES -> INST_DE_TRABAJO INST_DE_SALIDA

BLOQUE ->   BLOQUE_TRCX | BLOQUE_TRDX | BLOQUE_TMP | BLOQUE_FPTV |
            BLOQUE_FPTU | BLOQUE_FPTW | BLOQUE_CONT | BLOQUE_CAR

INST_DE_TRABAJO -> CARGA BIT PROCESO

PROCESO -> {OPERADOR OPERACIÓN}*

OPERACIÓN -> ( BIT OPERADOR OPERACIÓN )

OPERACIÓN -> BIT

INST_SALIDA -> {ASIGNA BIT}+

CARGA -> 'LD' | 'LDN' | 'LDR' | 'LDF'

ASIGNA -> 'ST' | 'STN' | 'BCX' | 'BDX'

OPERADOR -> SENTENCIA_AND | SENTENCIA_OR | SENTENCIA_XOR

SENTENCIA_AND -> 'AND' | 'ANDN' | 'ANDR' | 'ANDF'

SENTENCIA_OR -> 'OR' | 'ORN' | 'ORR' | 'ORF'

SENTENCIA_XOR -> 'XOR' | 'XORN' | 'XORR' | 'XORF'
```

```
BLOQUE_TRCX ->          'BLK' TRCX.NUM
                        'V' '=' NUM
                        INST_DE_TRABAJO
                        'IA'
                        'SLD_BLK'
                        INST_DE_TRABAJO
                        INST_DE_SALIDA
                        'FIN_BLK'

BLOQUE_TRDX ->          'BLK' TRDX.NUM
                        'V' '=' NUM
                        INST_DE_TRABAJO
                        'IA'
                        'SLD_BLK'
                        INST_DE_TRABAJO
                        INST_DE_SALIDA
                        'FIN_BLK'

BLOQUE_TMP  ->          'BLK' TMP.NUM
                        'V' '=' NUM
                        INST_DE_TRABAJO
                        'IA'
                        SLD_BLK
                        INST_DE_TRABAJO
                        INST_DE_SALIDA
                        'FIN_BLK'

BLOQUE_FPTV ->          'BLK' 'FPTV'
                        'V' '=' 'NUM'
                        INST_DE_TRABAJO
                        'SET'
                        'SLD_BLK'
                        INST_DE_TRABAJO
                        INST_DE_SALIDA
                        'FIN_BLK'

BLOQUE_FPTU ->          'BLK' 'FPTU'
                        'V' '=' NUM
```

```
                         INST_DE_TRABAJO
                         'SET'
                         INST_DE_TRABAJO
                         'RST'
                         'SLD_BLK'
                         INST_DE_TRABAJO
                         INST_DE_SALIDA
                         'FIN_BLK'

BLOQUE_FPTW ->           'BLK' 'FPTW'
                         'V' '=' NUM
                         INST_DE_TRABAJO
                         'SET'
                         INST_DE_TRABAJO
                         'RST'
                         'SLD_BLK'
                         INST_DE_TRABAJO
                         INST_DE_SALIDA
                         'FIN_BLK'

BLOQUE_CONT ->           'BLK' C.NUM
                         'V' '=' NUM
                         INST_DE_TRABAJO
                         'R'
                         INST_DE_TRABAJO
                         'CC'
                         INST_DE_TRABAJO
                         'DD'
                         'SLD_BLK'
                         INST_DE_TRABAJO
                         INST_DE_SALIDA
                         INST_DE_TRABAJO
                         INST_DE_SALIDA
                         INST_DE_TRABAJO
                         INST_DE_SALIDA
                         'FIN_BLK'

BLOQUE_CAR  ->           'BLK' C_AR.NUM
```

```
'EVC' '=' NUM
'EVD' '=' NUM
INST_DE_TRABAJO
'RST'
INST_DE_TRABAJO
'AOD'
INST_DE_TRABAJO
'CC'
'SLD_BLK'
INST_DE_TRABAJO
INST_DE_SALIDA
'FIN_BLK'
```

2.1.1 Jerarquía de lenguajes

Uno de los grandes trabajos en lingüística fue escrito por el lingüista Noam Chomsky. Chomsky proyectó un sistema de clasificación de lenguajes, el cual es importante porque cada clase de lenguaje en la jerarquía tiene un procedimiento de proceso asociado a éste. En esta jerarquía hay cuatro tipos de lenguajes: tipo 0, tipo 1, tipo 2 y tipo 3. Los lenguajes están organizados de acuerdo a su *expresividad*. La expresividad es un término técnico que denota las clases de sentencias que pueden escribirse en el lenguaje. Los lenguajes tipo 0 son los menos restrictivos y los más expresivos; los lenguajes tipo 3, por otra parte, son los más restrictivos y los menos expresivos.

Como se explicó antes, existen muchos tipos de lenguajes además de los llamados lenguajes naturales. Los lenguajes naturales incluyen el Español, el Inglés, el Francés y otros que la gente usa para comunicarse. Los lenguajes naturales son muy complejos. Sin embargo, también existen lenguajes muy simples, que consisten de alfabetos pequeños (**a**s y **b**s) y un número pequeño de reglas de producción. Mientras más complicado sea el lenguaje, más complicadas son las reglas de producción. Los lenguajes están relacionados con su tipo, y el tipo está relacionado con la complejidad del lenguaje.

Sin restricciones	Tipo 0
Sensibles al contexto	Tipo 1
Libres del contexto	Tipo 2
Regulares	Tipo 3

Dos de estos tipos de lenguajes son importantes para el desarrollo de los procesadores de lenguajes. La mayoría de los lenguajes son de tipo 2, lenguajes libres de contexto. Debido a que los lenguajes tipo 2 incluyen todos los aspectos de los lenguajes tipo 3 centramos nuestra atención en estos dos tipos de lenguajes. Ambos son fundamentales en el desarrollo de procesadores de lenguaje. La característica que clasifica una gramática (y consecuentemente el lenguaje definido por la gramática) es la forma de sus reglas de producción. Una gramática tipo 3 tiene reglas de producción de la siguiente manera:

```
A → xB
A → x

O

A → Bx
A → x
```

En estas reglas, **A** y **B** son símbolos no terminales y **x** es un símbolo terminal. Los lenguajes libres de contexto por otra parte tienen reglas de producción como la siguiente:

```
A → s
```

A es cualquier símbolo no terminal *solo*, y **s** es una cadena de símbolos terminales y no terminales. Estas dos diferentes clases de reglas nos conducen a lenguajes muy distintos. Otra manera de decir esto, es que estas dos distintas clases de reglas producen tipos de sentencias muy distintas. Por ejemplo, un lenguaje que puede definirse por una gramática libre del contexto (tipo 2) y no por una gramática regular (tipo 3) es el lenguaje de los paréntesis.

El lenguaje de los paréntesis consiste de un número igual de paréntesis propiamente anidados. La tabla muestra ejemplos de sentencias correctas e incorrectas en el lenguaje de los paréntesis.

Correctas	Incorrectas
()	(
(())	())
(()()(()))	(()(()

El lenguaje de los paréntesis puede definirse fácilmente por la siguiente gramática libre de contexto

```
S → ( )
S → ( S )
S → SS
```

Para mostrar como funciona esta gramática se muestran las derivaciones del tercero de los ejemplos previos.

```
S    →    (S)
     →    (SS)
     →    (SSS)
     →    (()SS)
     →    (()()S)
     →    (()()(S))
     →    (()()(()))
```

¿Porqué no es posible definir una gramática regular para el mismo lenguaje? Podemos escribir reglas que definieran un lenguaje con algunas de las sentencias en el lenguaje de los paréntesis, pero no *todas* las sentencias. Además, no podríamos garantizar, en un lenguaje regular, que todas las sentencias producidas fueran parte del lenguaje de los paréntesis.

2.2 AUTÓMATAS

Una vez que la gramática ha sido establecida para el lenguaje, puede ser usada para determinar si una sentencia es parte del lenguaje que se ha definido. Esta sección es el puente entre la teoría que soporta el procesamiento de lenguajes y el actual procesamiento de lenguajes. La siguiente figura muestra la estructura de un procesador de lenguajes genérico.

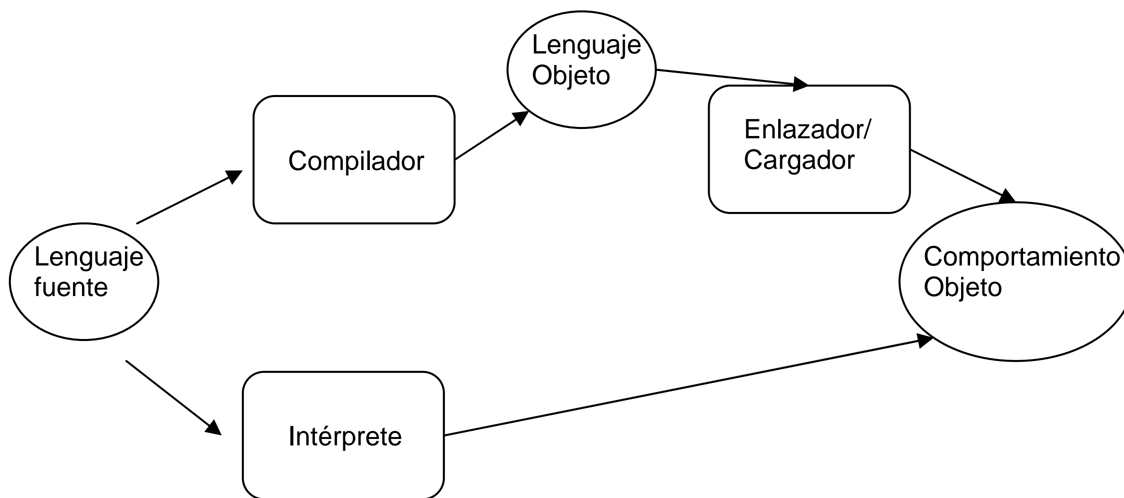


Figura 2.2.
Procesador de lenguajes genérico.

El compilador y el intérprete son programas que traducen el lenguaje fuente en un lenguaje de máquina o en un comportamiento de la máquina; los dos usan la gramática que define el lenguaje que procesan. El analizador léxico y el analizador sintáctico son programas internos del compilador y el intérprete. El analizador léxico proporciona información al analizador sintáctico acerca del programa en lenguaje fuente como se muestra en la figura.

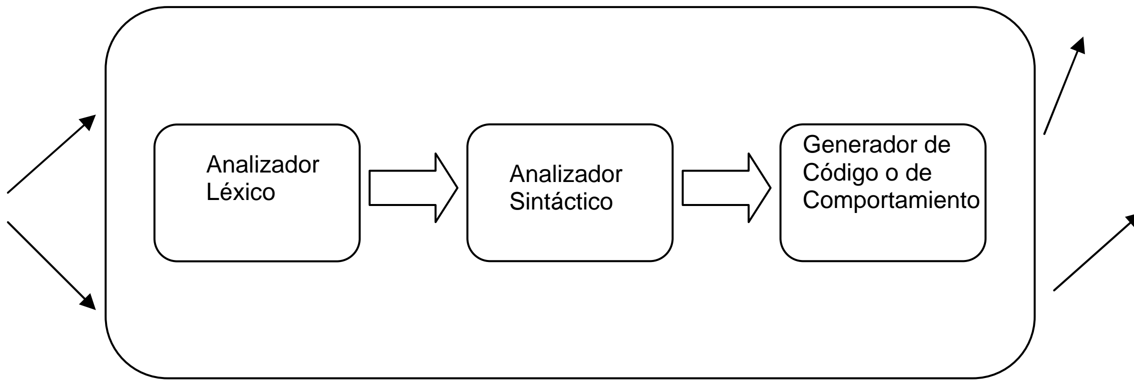


Figura 2.3.

Analizadores léxico y sintáctico dentro del procesador.

El analizador léxico y el sintáctico son dos clases de procesadores de lenguajes, pero con procesamientos de distintos lenguajes: de manera típica, el analizador léxico procesa los tokens de un lenguaje, mientras que el analizador sintáctico procesa las sentencias del lenguaje. Normalmente, el análisis léxico es distinto del análisis sintáctico.

2.2.1 De gramáticas a máquinas computacionales

Clasificando un lenguaje como tipo 2 o tipo 3 especificamos la clase de máquina computacional que se necesitará para procesar el lenguaje. Una *máquina computacional* es un programa o máquina que realiza un proceso de cómputo. Esto incluye a las computadoras y a las calculadoras, así como una clase de máquinas abstractas llamadas *autómatas*.

Un autómata es una máquina abstracta que consiste en un dispositivo que puede estar en cualquiera de un número finito de estados, uno de los cuales es el estado inicial y por lo menos uno es un estado de aceptación. A este dispositivo está unido un flujo de entrada por medio del cual llegan en secuencia los símbolos de un alfabeto determinado. La máquina tiene la capacidad para detectar los símbolos conforme llegan y basándose en el estado actual y el símbolo recibido, ejecutar una transición (de estado) que consiste en un cambio a otro estado o la permanencia en el estado actual. La determinación de cuál será precisamente la transición que ocurra al recibir un símbolo depende de un mecanismo de

control de la máquina, programado para conocer cuál debe ser el nuevo estado dependiendo de la combinación del estado actual y el símbolo de entrada. Para efectos de este trabajo, entenderemos autómata como la representación formal de una máquina computacional, la cual podemos representar en forma de grafo. En cuanto a procesamiento de lenguajes, un autómata es capaz de procesar las sentencias o expresiones de éste. Como ejemplo se muestra el autómata que se usa para procesar los bits de memoria mayores a 9.

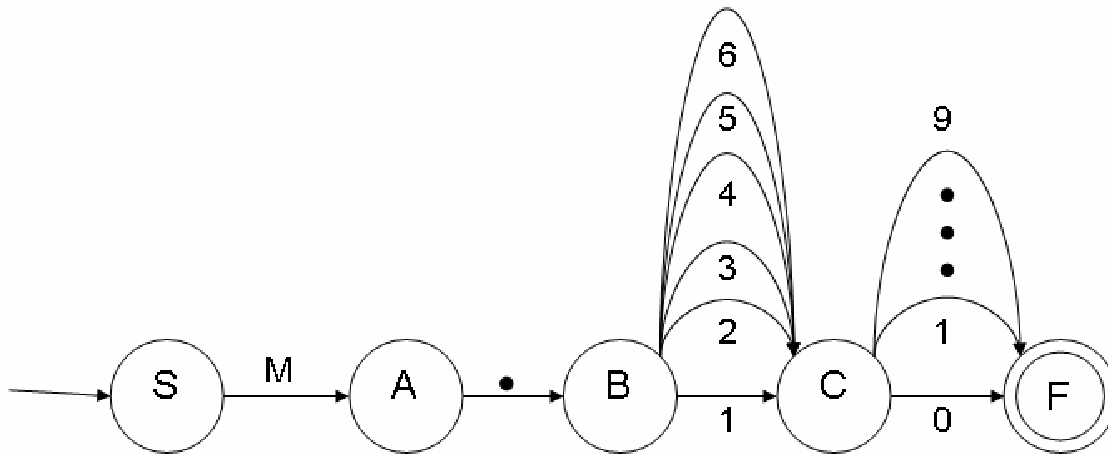


Figura 2.4.

Autómata que se utiliza para la detección de bits de memoria mayores a 9.

Un grafo es una combinación de nodos y flechas. Contiene dos clases de nodos representados por circunferencias sencillas o circunferencias de doble valla. Un nodo es usado para representar un estado de procesamiento en un autómata.

Un programa es la unión de una secuencia de instrucciones que una computadora puede interpretar y ejecutar. Cuando la computadora ejecuta el programa, está en una instrucción o paso en particular. En cada paso del programa, las variables usadas tienen valores específicos. Si tomáramos una fotografía del programa en ese punto, podríamos observar el *estado actual* del programa de manera análoga al grafo de un autómata.

Un estado actual del autómata está representado por un nodo. El nodo especifica que ha sido procesado. Conforme los nodos son recorridos en el grafo del autómata, mayor procesamiento se ha realizado. Como ejemplo procesamos el bit de memoria M.15.

S es un estado especial del autómata llamado estado inicial. Todos los cómputos del autómata comienzan desde este estado. Del estado **S** se puede hacer una transición al estado **A** leyendo el carácter **M**. En el caso del bit **M.15** el primer carácter es una **M**, así que podemos transitar al estado **A**. La cadena que resta para ser procesada es **.15**. Posteriormente encontramos un carácter de '**.**' por lo que podemos pasar al estado **B**, y nos resta por procesar la cadena **15**. La transición del estado **B** al estado **C** será posible si se detecta un dígito del 1 al 6. Esto se debe a que el conjunto de instrucciones nos permite usar bits de memoria del 0 al 63. El carácter 0 en este caso no es válido porque el procesamiento de bits con valor menor a 10 es realizado por un autómata distinto. Debido a que el siguiente carácter detectado es un **1**, la transición es posible. Para alcanzar el último estado es necesario que el carácter que se detecte sea cualquiera de los dígitos. El carácter restante es el **5**, con lo que se alcanza el estado **F** el cual es el estado final y lo reconocemos por la doble circunferencia. Cuando este estado se alcanza, y no hay más entradas para procesar, el autómata ha procesado con éxito la sentencia del lenguaje. Si la sentencia es procesada por un autómata y el último estado alcanzado no es el estado final, entonces el autómata no acepta la sentencia, y la sentencia no es parte de los lenguajes aceptados por el autómata. En este caso, la palabra M.15 es parte del lenguaje aceptado por este autómata, lo cual significa que M.15 es un bit de memoria válido.

En el lenguaje diseñado para nuestro programa, muchos de los tokens son procesados por autómatas. Las palabras reservadas, los bits de entrada, bits de memoria, bits de salida y los valores numéricos son algunos ejemplos de esto.

El autómata descrito procesará únicamente lenguajes de tipo 3, los cuales consisten en un número de estados de tamaño finito y un conjunto de reglas para hacer transiciones entre estados. Este tipo de autómata es llamado autómata finito determinístico. El autómata para procesar un lenguaje de tipo 2 o libre de contexto se llama autómata de pila. La

estructura de la pila permite que este tipo de autómata procese sentencias más complejas que pueden resultar de un lenguaje libre de contexto. Una de las diferencias fundamentales entre los lenguajes de tipo 3 y los de tipo 2 es la necesidad de recordar si un símbolo fue procesado para generar una respuesta. Este es el caso del lenguaje de los paréntesis en el cual es necesario recordar el número de paréntesis izquierdos para reconocer el número de paréntesis derecho. El autómata de pila, mantiene una estructura que es usada como la memoria para este propósito.

En los autómatas finitos deterministas las transiciones entre estados están basadas en el estado actual y en el carácter actual de la cadena de entrada. En los autómatas de pila, las transiciones están en función del estado actual, la cadena de entrada, y el carácter que se encuentra en la punta de la pila del autómata.

En el lenguaje de los paréntesis es necesario regresar a revisar cada paréntesis izquierdo encontrado, para cada paréntesis derecho que se detecta posteriormente en la sentencia. Esto puede lograrse colocando una marca en la pila del autómata cada vez que un paréntesis izquierdo se encuentra. Cuando se encuentra un paréntesis derecho en la sentencia, se realiza una transición entre estados si la pila tiene una marca correspondiente al paréntesis izquierdo; en caso de que no, la sentencia no es parte del lenguaje. Y, si el último estado del autómata no es un estado final, la sentencia tampoco es parte del lenguaje.

El autómata consiste de cuatro estados: el estado inicial (S), el estado para procesar paréntesis izquierdos (L), el estado para procesar paréntesis derechos (R), y el estado final. Las sentencias de este lenguaje deben terminar con el signo #. Este carácter, cuando se encuentre, permitirá una transición del estado R al estado F. El autómata para procesar el lenguaje de los paréntesis se muestra en la siguiente figura.

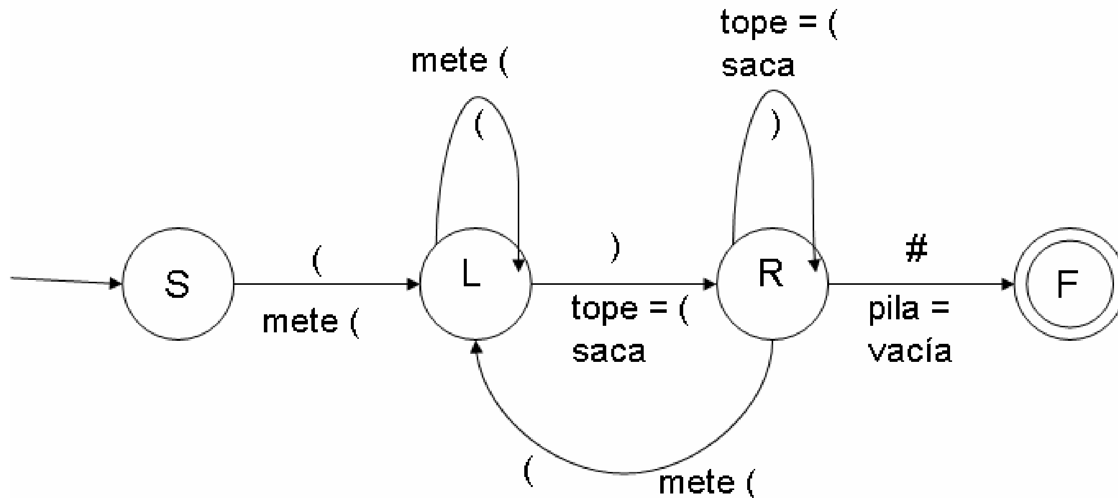


Figura 2.5.
Autómata de pila para el lenguaje de los paréntesis

Un aspecto importante del lenguaje de los paréntesis es que los paréntesis izquierdos y derechos están emparejados unos con otros. El autómata de pila tiene la memoria necesaria para revisar los paréntesis izquierdos mientras son procesados en una sentencia del lenguaje.

Los arcos del autómata que se muestra en la figura tienen dos tipos adicionales de notación: operaciones de pila (mete y saca) u operaciones de revisión como pila vacía y tope = carácter. Las revisiones ocurren antes de las operaciones. La notación **)**, **tope = (**, y **saca**, significa que el actual carácter a procesar es un), que la punta de la pila debe ser un **(**, que si ambas condiciones se cumplen, debe sacarse el elemento que está en la punta de la pila. Después de esto, la transición al siguiente estado se lleva a cabo.

Un ejemplo de procesamiento de sentencias de este tipo se muestra en la tabla para la cadena (()())#.

CADENA DE ENTRADA	ESTADO ACTUAL	CARÁCTER ACTUAL	PILA	NUEVO ESTADO
(()())#	S	(	vacía	L

()())#	L	(	(	L
)())#	L	)	((	R
())#	R	(	(	L
))#	L	)	((	R
)#	R	)	(	R
#	R	#	vacía	F

Figura 2.6.

Procesamiento del autómata de pila de la sentencia (()())#.

Para la cadena (()())#, el autómata de pila alcanza el estado final. En cambio, si se analiza la sentencia ())# vemos que el autómata no la acepta, por lo tanto, no es parte del lenguaje.

CADENA DE ENTRADA	ESTADO ACTUAL	CARÁCTER ACTUAL	PILA	NUEVO ESTADO
())#	S	(	vacía	L
))#	L	)	(	R
)#	R	)	vacía	no hay transición, la sentencia no cumple

Figura 2.7.

Procesamiento del autómata de pila de la sentencia ())#.

2.2.2 De máquinas computacionales a Programas

La construcción del procesador de lenguaje es como realizar un rompecabezas de muchas piezas. Las máquinas abstractas no son parte del rompecabezas. Debemos dar un paso más para volverlas piezas. El paso involucra volver estas máquinas abstractas, ya sean autómatas finitos deterministas o autómatas de pila, programas que puedan procesar

cadenas para determinar si pertenecen al lenguaje. Después de que las máquinas abstractas son transformadas en programas, pueden ser incorporadas al proceso de análisis.

El puente entre lo teórico y lo práctico en este caso es directo. De la gramática, puede formularse una máquina abstracta, y de la máquina puede escribirse un programa. Esto es muy importante porque es la base de cómo puede armarse un procesador de lenguajes.

2.3 LA ARQUITECTURA DEL COMPILADOR

La estructura básica del procesador de lenguajes se muestra en la figura.

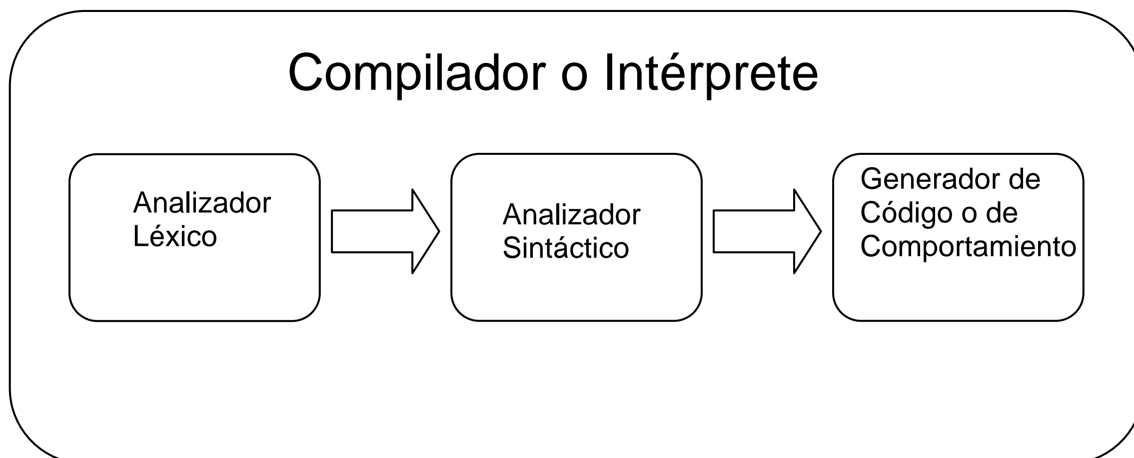


Figura 2.8

Organización básica de un procesador de lenguajes

El procesador de lenguaje consiste de un analizador léxico, un analizador sintáctico, y un generador de código. El *analizador léxico* acepta cadenas escritas en el lenguaje e identifica subcadenas que son elementos del lenguaje. Las subcadenas, llamadas tokens, son pasadas hacia el analizador sintáctico cuya responsabilidad es construir las estructuras que representan las sentencias del lenguaje. La salida del analizador sintáctico es pasada entonces al generador de código.

El generador de código o comportamiento representa dos componentes distintos. Si el procesador de lenguajes produce código en un lenguaje fuente, como podría ser lenguaje máquina, entonces se llama *generador de código*. Si, por otra parte, no se produce lenguaje máquina, y el procesador de lenguaje ejecuta el programa, entonces este componente es un *intérprete*. En la figura se le llama generador de comportamiento porque el intérprete desarrolla el comportamiento especificado en el programa.

Un último componente de muchos procesadores de lenguaje es un *optimizador*. Un optimizador procesa la salida del generador de código para hacer el código más eficiente, o se coloca entre el analizador sintáctico y el intérprete para que la salida interpretada del programa sea más eficiente. Una descripción más completa de un procesador de lenguajes se muestra a continuación.

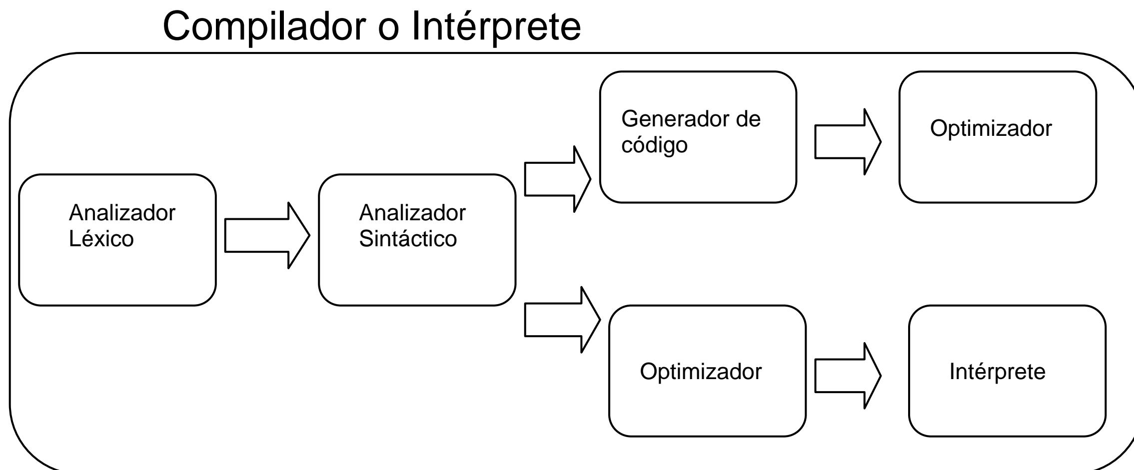


Figura 2.9

Organización de un compilador o un intérprete

De manera completa, el procesador funciona como sigue: el analizador léxico lee un archivo de programa e identifica todos los tokens en él. La salida del analizador léxico puede ser otro archivo o una estructura de memoria que será usada por el analizador sintáctico. Ya que el analizador léxico ha identificado todos los tokens del programa, los elementos identificados son pasados al analizador sintáctico.

Otra posibilidad es que el analizador sintáctico controle al analizador léxico. Cuando se opera de esta manera, el analizador sintáctico le pide al analizador léxico determinado tipo de token. El analizador léxico revisa el siguiente token del programa, determina su tipo y regresa el token al analizador sintáctico. En el primer caso descrito el analizador léxico pasa una vez a través del código y entonces pasa todos los tokens al analizador sintáctico, el cual volverá a revisar los tokens. En el segundo caso, el analizador léxico y el sintáctico pasan solamente una vez a través del código.

El analizador sintáctico produce estructuras que representan sentencias. Estas estructuras pueden ser árboles, una representación intermedia del código de salida, o alguna otra estructura. La salida del analizador sintáctico puede ser un archivo o una estructura en memoria.

Una vez que el analizador sintáctico ha creado una representación del programa, y asumiendo que no ha habido errores, el generador de código puede producir código en el lenguaje objeto. Si existe un optimizador en el procesador, entonces, en el caso del compilador, la salida del generador de código es analizada para intentar cambios que mejorarán la eficiencia del código. Si el procesador de lenguajes fuera un intérprete, el optimizador analizaría la salida ejecutable producida por el analizador sintáctico para mejoras en eficiencia.

2.4 ELEMENTOS DEL COMPILADOR

Una gramática puede imaginarse como una especie de mapa de carreteras. Este mapa puede seguirse para determinar si una sentencia es parte del lenguaje. Y, así como existen uniones en el camino, existen uniones en las reglas de producción, las cuales pueden ser identificadas en reglas cuyos lados no-terminales son los mismos. Durante el proceso de derivación, una sola regla de derivación puede ser escogida para continuar una derivación. Algunas veces la derivación será incorrecta y será necesario escoger otra regla de producción. Esto es lo mismo que escoger la dirección incorrecta en una desviación y regresar para tomar un camino diferente. La manera en que una regla de producción es

escogida para cada paso de la derivación es un problema fundamental en la automatización del proceso de análisis.

Existen dos partes en el proceso de análisis. La primera es la identificación de los elementos del lenguaje como pueden ser variables, constantes y palabras reservadas. Esto es la fase de análisis léxico. La segunda parte es la construcción de la derivación para la sentencia. Una derivación puede llevarse a cabo en dos maneras:

1. Decimos que una derivación es arriba-abajo si avanza desde el símbolo de inicio de la gramática a la sentencia.
2. En el enfoque abajo-arriba, la derivación inicia con la sentencia. Los elementos de la sentencia son identificados en la gramática y el proceso de derivación avanza hacia atrás. El último símbolo derivado en un análisis abajo-arriba es el símbolo de inicio de la gramática.

La gramática guía el proceso de derivación. Cada regla controla como avanzará la derivación. En un análisis arriba-abajo, el analizador buscará elementos en el lado derecho de una regla. En esta relación el analizador llama al analizador léxico de manera cooperativa. El analizador léxico también puede ser usado para procesar un programa completamente y producir una versión analizada léxicamente del programa. Los elementos de la sentencia son llamados *tokens*.

2.4.1 Identificación de elementos: análisis léxico

Los autómatas pueden ser usados para crear programas, los cuales a su vez pueden ser usados para reconocer elementos de lenguaje. Un analizador léxico consiste de una serie de estas máquinas (o algún tipo de mecanismo alternativo) para reconocer el siguiente token en una cadena de entrada. Un analizador léxico procesará cualquier elemento de lenguaje que sea una sentencia en un lenguaje de tipo 3.

De acuerdo al tipo de operación que realizan y basándonos en el conjunto de instrucciones definido anteriormente, los tokens que reconocerá el analizador léxico se dividen en 10 tipos, lo cual, además, facilita su reconocimiento durante la implementación: carga, asignación, operador, negación, instrucciones de pila, instrucciones de bloque, identificador de bloque, bits, símbolos y números.

TIPO	TOKEN
CARGA	LD, LDN, LDR, LDF
SÍMBOLO	(,), =
ASIGNACIÓN	ST, STN, BCX, BDX
OPERADOR	OR, ORN, ORR, ORF, AND, ANDN, ANDR, ANDF, XOR, XORN, XORR, XORF
NEGACIÓN	N
MEMORIA_PILA	MPL, LPL, SPL
PROG_BLOQUE	INI_BLOQUE, SLD_BLOQUE, FIN_BLOQUE
ENTRADA_BLOQUE	ACT, SET, RST, R, SF, CC, DD, AOD
SALIDA_BLOQUE	SLD,Q, SPA, SDV, SDL
VALOR_TOKEN	V, EVC, EVD
TIPO_BLOQUE	FPTU, FPTV, FPTW, C.0 A C.2, TMP.0 A TMP.2, TRCX.0 A TRCX.2, TRDX.0 A TRDX.2, C_AR.0 A C_AR.2
BIT	E.0 A E.15, S.0 A S.15, M.0 A M.63
NÚMERO	ENTEROS (0 - 99)

Figura 2.10

Tipos de tokens reconocidos por el analizador léxico del programa cplc

2.4.2 Estructura del lenguaje: análisis sintáctico.

Si una gramática es un mapa del camino, el analizador sintáctico es quien sigue el camino. Los destinos del mapa son las sentencias del lenguaje. El análisis sintáctico es el proceso de recorrer las rutas del mapa hacia una dirección apropiada. La gramática debe construirse para que se pueda, sin mucha complejidad, decidir que regla de producción usar

en cada paso del proceso de análisis. Seleccionar una regla de producción es análogo a encontrar el siguiente camino a elegir.

Consideremos la siguiente sentencia en lista de instrucciones

```
LD E.0
AND E.1
OR E.2
ST S.0
```

Esta sentencia está definida por las reglas de producción mostradas a continuación

```
(R1)  SENTENCIA_PLC -> INSTRUCCIONES | BLOQUE
(R2)  INSTRUCCIONES -> INST_DE_TRABAJO INST_DE_SALIDA
(R3)  INST_DE_TRABAJO -> CARGA BIT PROCESO
(R4)  PROCESO -> {OPERADOR OPERACIÓN}*
(R5)  OPERACIÓN -> BIT
(R6)  OPERACIÓN -> ( BIT OPERADOR OPERACIÓN )
(R7)  INST_SALIDA -> {ASIGNA BIT}+
(R8)  CARGA -> LD | LDN | LDR | LDF
(R9)  ASIGNA -> ST | STN | BCX | BDX
(R10) OPERADOR -> SENTENCIA_AND | SENTENCIA_OR | SENTENCIA_XOR
(R11) SENTENCIA_AND -> AND | ANDN | ANDR | ANDF
(R12) SENTENCIA_OR -> OR | ORN | ORR | ORF
```

El no terminal de esta gramática es el símbolo SENTENCIA_PLC. El análisis léxico de esta sentencia nos genera la siguiente lista:

```
LD    CARGA
E.0   BIT_ENTRADA
AND   OPERADOR
E.1   BIT_ENTRADA
OR    OPERADOR
E.2   BIT_ENTRADA
```

```
ST    ASIGNACION
S.0   BIT_SALIDA
```

Para iniciar un análisis arriba-abajo de esta sentencia, el analizador sintáctico usa las reglas de producción que contienen el símbolo no terminal. La regla de producción que contiene este símbolo es R1. El primer símbolo del lado derecho de esta regla es el símbolo no terminal INSTRUCCIONES. El primer token no es INSTRUCCIONES. De hecho, INSTRUCCIONES ni siquiera es reconocido por el analizador léxico. El analizador busca reglas que definan el símbolo INSTRUCCIONES. La regla R2 define este símbolo no terminal. INSTRUCCIONES está definido por una INST_DE_TRABAJO seguida de una INST_DE_SALIDA. Ahora se busca una regla que defina el símbolo INST_DE_TRABAJO. R3 la define como CARGA BIT PROCESO, así que ahora busca la definición de CARGA y encuentra el símbolo terminal LD que coincide con el primer elemento de la lista. El segundo elemento del lado derecho de R3 es un bit, por lo que lo reconoce y continúa el análisis buscando una definición para PROCESO. R4 define proceso como cero o más ocurrencias de OPERADOR seguido de OPERACIÓN. El siguiente token es un OPERADOR, lo reconoce y busca una regla que defina OPERACIÓN. En este caso encuentra dos, R5 y R6. En este caso hace un recorrido por ambas, si falla la primera, regresa a este punto y recorre la segunda. La R5 representa una secuencia sencilla y nos llevará a un error ya que la sentencia trabaja con varios bits. La R6 es una regla recursiva, es decir, recurre a sí misma para lograr el reconocimiento de las sentencias. Está definida por BIT, OPERADOR y OPERACIÓN. El siguiente token es el bit de entrada 1, lo reconoce y busca un operador, encuentra el operador OR y busca una OPERACIÓN. De acuerdo a R5, OPERACIÓN también es un solo bit, así que reconoce a E.2 como OPERACIÓN y con esto completa la secuencia OPERACIÓN inicial. Es importante subrayar que la secuencia OPERACIÓN contiene otra secuencia OPERACIÓN dentro de ella. Con esto se logra también el reconocimiento de R4, PROCESO, que había quedado pendiente. A su vez, el reconocimiento de PROCESO completa el reconocimiento de INST_DE_TRABAJO. Ahora el analizador busca reglas que definan INST_DE_SALIDA. R7 la define como una o más ocurrencias de ASIGNA seguido de BIT. El siguiente token es ST el cual es un ASIGNA y el siguiente a su vez es un BIT, con lo que se completa el

reconocimiento de INST_DE_SALIDA. Esto completa el reconocimiento de INSTRUCCIONES, la cual es una SENTENCIA_PLC y el reconocimiento queda completo.

Empezando con el símbolo de inicio de la gramática, todas las reglas de producción son seleccionadas teniendo este símbolo como su símbolo de lado izquierdo. En el caso de la gramática había una sola regla. Vemos entonces el primer símbolo del lado derecho. ¿Este símbolo coincide con el primer token? Si coincide, avanzamos al siguiente símbolo de la cadena de entrada. Si no, revisamos si el siguiente símbolo de la regla de producción es el lado izquierdo de alguna regla de producción. Si lo es, colectamos todas estas reglas y revisamos cada una para ver si el primer símbolo en el lado de la mano-derecha coincide con el token actual. Este proceso continúa hasta que todos los tokens de la sentencia son procesados o hasta que no puedan hallarse reglas para aplicar al token actual.

Con esto, mostramos la relación entre el análisis léxico y el análisis sintáctico. Durante el proceso de construcción de los autómatas, las flechas entre los estados son creadas examinando las reglas de producción y determinando que tipo de token o carácter tiene que aparecer para moverse al siguiente estado. Siempre –y esto es muy importante - el símbolo tiene que ser distinto. En otras palabras, cada flecha que sale de un estado, tiene un símbolo único que la marca; no existen dos flechas con la misma etiqueta. Dos flechas pueden tener el mismo símbolo, pero el procesamiento de los estados se vuelve muy enredado. El autómata resultante se llama *no determinista*, porque si un estado tiene al menos dos flechas con la misma etiqueta, los dos caminos deben recorrerse antes de que pueda seleccionarse la ruta correcta.

Por otra parte, si ningún estado tiene dos o más flechas etiquetadas con el mismo símbolo, el autómata es llamado *determinista*, y quiere decir que la ruta correcta siempre puede ser determinada. El diseño del programa cplc fue pensando en un autómata determinista. Esto se puede lograr haciendo que dos reglas de producción con el mismo símbolo en el lado izquierdo no tengan el mismo primer símbolo en el lado derecho.

2.4.3 Del análisis al código

La última tarea del programa es la producción del código. Hasta el momento toda la maquinaría que ha sido descrita está diseñada para el reconocimiento de lenguajes. Con algunas modificaciones, esta maquinaria también puede producir código. Determinadas operaciones son agregadas al analizador sintáctico para guardar la información recolectada en las sentencias mientras se realiza el proceso. Esta información es usada para generar código.

Con cada sentencia están asociadas una o más operaciones de los autómatas. Las sentencias en el lenguaje cplc están agrupadas en general en tres categorías: INSTRUCCIONES booleanas, sentencias de BLOQUE, y sentencias de PILA. Para cada tipo distinto de sentencia, el programa cplc, toma una lista de argumentos y mientras los procesa construye una estructura con información para el método encargado de la traducción.

Lo anterior se muestra con un ejemplo. Consideremos el siguiente programa en lista de instrucciones.

```
LD E.0
AND( E.1
OR( M.3
ANDN E.2
))
ST S.0
```

Mientras se realiza el análisis sintáctico de la sentencia de trabajo, se crea también en una estructura de lista interna la siguiente expresión postfija:

```
0  1  3  2  ANDN  OR  AND
```

La cual, puede ser representada por el siguiente árbol de recorrido en postorden.

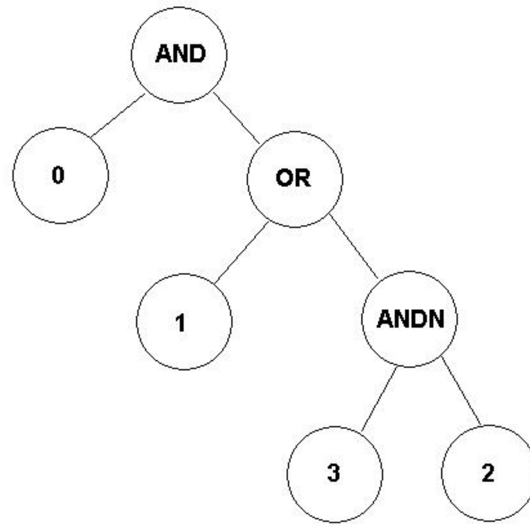


Figura 2.11.
Estructura para la sentencia de ejemplo.

Al generar el código, el programa cplc se encarga de llamar a las estructuras correctas para leer la información recabada al realizar el análisis sintáctico.

CAPÍTULO 3. DESARROLLO DEL SISTEMA

Como se explicó en el capítulo anterior, los componentes del procesador de lenguajes son el analizador léxico, analizador sintáctico y generador de código o comportamiento. Sin embargo, es importante profundizar en la lista de componentes y dar algunos detalles más de cómo estos elementos se relacionan entre ellos. En este capítulo presentaremos el diseño completo, en términos de diseño del procesador de lenguajes. Es importante entender como los diferentes componentes de un procesador de lenguajes se relacionan unos con otros. Por ejemplo, ¿cómo pasa información el analizador léxico al analizador sintáctico?

También se tratan dos temas adicionales. Primero, las características de las estructuras de datos, las cuales son necesarias para tener una idea general de lo que realiza el programa. El segundo tema es la simplicidad. Debido a que es necesario implementar el procesador de lenguajes en determinado tiempo, las cosas que faciliten la tarea son completamente útiles. La complejidad del procesador de lenguajes está directamente relacionada con la complejidad de la gramática del lenguaje que procesa, las simplificaciones que se explican se realizaron con la finalidad de hacer mas sencilla la implementación del programa.

A continuación se presenta la estructura del algoritmo del analizador léxico, analizador sintáctico e intérprete.

3.1 Estructura del analizador léxico

El proceso desarrollado por el analizador léxico está representado por el siguiente algoritmo:

1. Para todos los caracteres en un archivo de programa:
2. Identificar el tipo de token de los siguientes carácteres en el archivo de programa.

3. Si ningún tipo puede ser identificado, envía un mensaje de error, y continuar con el paso 2 para tratar de encontrar algún token identificable (recuperación).
4. Si algún tipo de token es identificado, obtener el token completo del archivo de programa.
5. Escribir el tipo de token y el valor de token en el archivo de salida del analizador léxico.

La identificación de cada tipo de token requiere diferentes autómatas. Para cada token, existe un método de clase para reconocer palabras reservadas, enteros, bits de entrada, bits de memoria, bits de salida, operadores, etc. Los métodos se llaman: esCarga, esAsignacion, esOperador, esNegacion, esMemoriaPila, esBloque, esBit, esSimbolo, esNumero, esTipoBloque, etc. Cada método consta de uno o varios *autómatas*.

Usando este agrupamiento la parte del analizador léxico que identifica tokens puede organizarse como se muestra:

```
private boolean identificarCadena( ){
            if ( esCarga() || esAsignacion() || esOperador() ||
                  esNegacion()  || esMemoriaPila() ||
                  esBloque() || esBit() || esSimbolo() ||
                  esNumero() || esTipoBloque() )
                  return true;
            else
                  return false;
} // fin de identificarCadena
```

Donde cada método se encarga de identificar y asignar un tipo de token a cada palabra reconocida.

¿Qué pasa cuando se llama a una función autómata y esta no puede identificar el token? El paso 3 del algoritmo del analizador léxico toma las precauciones necesarias para esta situación. Esto probablemente significa que el token no es parte del lenguaje y por lo tanto representa alguna clase de error en la sentencia que está siendo procesada.

Lo primero que se hace es enviar un mensaje de error indicando que se ha detectado un token no válido, posteriormente se realiza un intento de recuperación para que el analizador léxico pueda avanzar lo más posible en el programa. Esto es conseguido saltando los tokens que sean necesarios para localizar un tóken válido. Cuando se localiza ese token, el análisis léxico continúa.

Es importante hacer notar que este tipo de recuperación no incluye el algoritmo de sustitución del token válido, por lo que en caso de presentarse errores en el análisis léxico, no se concluye el proceso de compilación.

3.1.1 Simplificaciones para el analizador léxico

La construcción de autómatas para el reconocimiento de los elementos del lenguaje es el enfoque más general para realizar el análisis léxico. Dependiendo de las características del lenguaje, es posible tomar un enfoque más simple.

Por ejemplo, si el lenguaje usa el carácter de espacio como delimitador, se puede tomar ventaja de esta característica para simplificar el proceso de separación de los tokens. Bajo estas circunstancias, el proceso de separación se convierte en una búsqueda en la entrada hasta que se encuentre un espacio. Cuando se encuentra, sabemos que estamos al final de un token y al inicio de otro. Esto permite que la sentencia de entrada se descomponga en tokens.

Por ejemplo, la sentencia:

```
BLK TMP.0
V = 3
LD E.1
ACT
SLD_BLK
LD SLD
```

```
ST S.3
FIN_BLK
```

Puede ser descompuesta de la siguiente manera:

```
BLK         INI_BLOQUE
0           TMP
V           VALOR_BLOQUE
=           SIMBOLO
3           NUMERO
LD          CARGA
1           BIT_ENTRADA
ACT         ENTRADA_BLOQUE
SLD_BLK     SLD_BLOQUE
LD          CARGA
SLD         SALIDA_BLOQUE
ST          ASIGNACION
3           BIT_SALIDA
FIN_BLK     FIN_BLOQUE
```

Dado que Java, al igual que C, contiene funciones para identificar cadenas y separar tokens, los autómatas que usamos no necesariamente realizan la inspección carácter por carácter, ya que pueden detectar palabra por palabra.

3.2 Estructura del analizador sintáctico

El analizador sintáctico crea estructuras que eventualmente serán traducidas a código ejecutable. Empezando con el símbolo de inicio de la gramática, el analizador sintáctico utiliza la estructura fragmentada en tokens proporcionada por el analizador léxico para completar los requerimientos de la gramática.

Consideremos nuevamente la sentencia del ejemplo anterior y sus correspondientes reglas de producción

```
INST_DE_TRABAJO   -> CARGA BIT PROCESO
PROCESO           -> {OPERADOR OPERACIÓN}*
OPERACIÓN         -> ( BIT OPERADOR OPERACIÓN )
OPERACIÓN         -> BIT
INST_SALIDA       -> {ASIGNA BIT}+
CARGA             -> 'LD' | 'LDN' | 'LDR' | 'LDF'
ASIGNA            -> 'ST' | 'STN' | 'BCX' | 'BDX'
BLOQUE_TMP        -> 'BLK' TMP.NUM 'V' '=' NUM INST_DE_TRABAJO 'IA'
                     'SLD_BLK' INST_DE_TRABAJO INST_DE_SALIDA 'FIN_BLK'
```

Usando la gramática, el analizador produce la estructura de árbol que se muestra en la figura 3.1. Una vez que la sentencia ha sido procesada a esta estructura los elementos relevantes para la generación de código pueden ser extraídos del árbol.

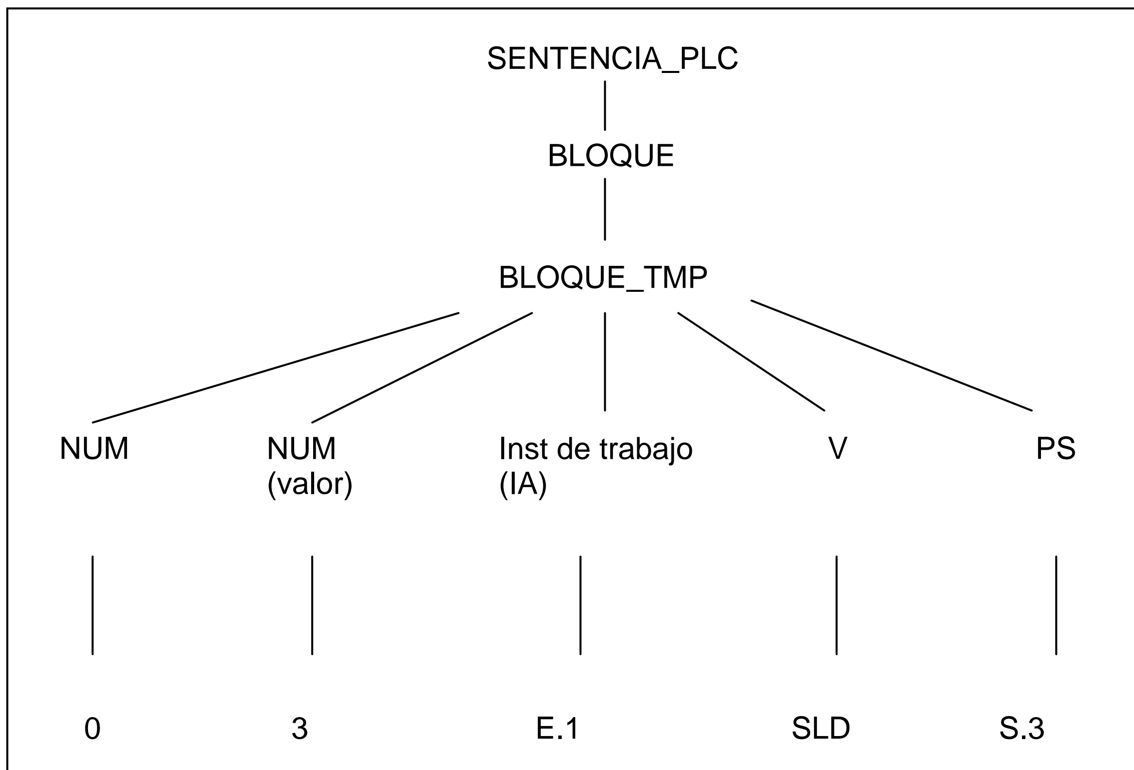


Figura 3.1.

Árbol de análisis para una sentencia BLOQUE_TMP.

Para esta sentencia, los elementos significativos son: el número de bloque, el valor del temporizador, dos sentencias INST_DE_TRABAJO y una sentencia INST_DE_SALIDA. La estructura de almacenamiento se muestra en la figura 3.2.

Tipo de operación	Número de bloque	Valor del temporizador	Sentencia Inst_de_trabajo	Sentencia Inst_de_trabajo	Sentencia Inst_de_salida

Figura 3.2.

Estructura de datos para la sentencia BLOQUE_TMP.

Los datos pueden ser extraídos del árbol y usados para la producción de la sentencia BLOQUE_TMP, creándose con esto una relación entre el árbol de análisis y la estructura de datos.

Esta relación puede imaginarse como una serie de operaciones en el árbol de análisis. Las operaciones crean una estructura de datos para cada tipo de sentencia. Existe al menos una secuencia de operaciones para cada sentencia que genera código. Por ejemplo, las siguientes operaciones serán usadas para crear la estructura de datos correspondiente a la sentencia de bloque TMP.

1. Crear una nueva estructura para la sentencia BLOQUE.
2. Crear una nueva estructura para la sentencia BLOQUE_TMP.
3. Localizar el token correspondiente al número de bloque TMP que se usa.
4. Localizar el token con el valor del temporizador.
5. Crear una sentencia INST_DE_TRABAJO para asignar su resultado lógico a la entrada IA y localizar los tokens correspondientes.
6. Crear una sentencia INST_DE_TRABAJO para trabajar con la salida SLD como si fuera un operador y localizar los tokens correspondientes.
7. Crear una sentencia INST_DE_SALIDA y localizar los tokens correspondientes.

Esto concluye la construcción de la estructura para la sentencia BLOQUE_TMP.

Es importante mencionar que no basta con que se encuentren presentes los valores para la construcción de la sentencia, también es necesario que la sentencia que se examina cumpla con las reglas de producción de la gramática. De lo contrario y de manera similar al análisis léxico, es enviado un mensaje de error sintáctico y el proceso es detenido. Esto sucede porque quien construye las estructuras son los autómatas reconocedores de las gramáticas asociadas a las sentencias y no están programados para realizar las funciones de corrección o avance a prueba de errores.

Para cualquier sentencia en el lenguaje, existen operaciones similares para construir el resto de las estructuras. A la secuencia de operaciones para construir una estructura de sentencia se le llama *función constructora de estructura.*

3.2.1 Simplificaciones para el analizador sintáctico

De la misma manera que el analizador léxico, el analizador sintáctico también puede ser simplificado. En el caso del analizador sintáctico, la manera de hacerlo es seleccionando la producción correcta que se aplicará a determinado tipo de sentencia o elemento del lenguaje. Si la regla a aplicar puede predecirse exactamente del token actual, entonces el programa del analizador sintáctico puede consistir en una serie de llamadas a funciones que procesen elementos del lenguaje.

La gramática está diseñada para que cada sentencia se distinga de las demás por su palabra inicial. Esto puede hacerse ya que nuestro lenguaje no es sintácticamente complejo como los lenguajes de programación de propósito general, de otra manera no sería posible seguir este camino. Esta preferencia por la simplicidad es muy apropiada y simplifica mucho la tarea de implementación.

El analizador sintáctico produce un árbol de análisis para cada sentencia y convierte la estructura en otra lista para la traducción. Cada sentencia tiene una función de análisis y una función constructora de estructura asociada con ella. En ocasiones, cuando el elemento

de una sentencia es complejo, lo que significa que el elemento requiere su propia estructura, el elemento también tiene una función de análisis y una función constructora de estructura asociado a él.

La entrada al analizador sintáctico es la estructura de tokens creada por el analizador léxico. Determinados grupos de estos tokens corresponden a sentencias. El primer token de cada grupo identifica la sentencia. Este token es usado para escoger la función de análisis y la función constructora correctas. La función de análisis lee los tokens apropiados de la salida del analizador léxico y posteriormente los pasa a la función constructora. El algoritmo para el analizador sintáctico se especifica a continuación.

```
1. Abrir la estructura de salida del analizador léxico
2. Leer un token de la estructura
3. While (true) {
      a. Si no hay mas tokens salir del ciclo
      b. Si el token no es válido reportar error
      c. Si el token es LD
               Procesar los tokens para una sentencia_INST_DE_TRABAJO
               Construir una estructura para la sentencia_INST_DE_TRABAJO
      d. Si el token es ST
               Procesar los tokens para una sentencia_INST_SALIDA
               Construir una estructura para la sentencia_INST_SALIDA
      e. Si el token es INI_BLOQUE
         e1. Si token.tipo es TMP
               Procesar los tokens para un BLOQUE_TMP
               Construir una estructura para el BLOQUE_TMP
         e2. Si token.tipo es TRCX
               Procesar los tokens para un BLOQUE_TRCX
               Construir una estructura para el BLOQUE_TRCX
         e3. Si token.tipo es TRDX
               Procesar los tokens para un BLOQUE_TRDX
               Construir una estructura para el BLOQUE_TRDX
         e4. Si token.tipo es FPTV
               Procesar los tokens para un BLOQUE_FPTV
               Construir una estructura para el BLOQUE_FPTV
         e5. Si token.tipo es FPTW
               Procesar los tokens para un BLOQUE_FPTW
               Construir una estructura para el BLOQUE_FPTW
```

```
        e6. Si token.tipo es FPTU
                Procesar los tokens para un BLOQUE_FPTU
                Construir una estructura para el BLOQUE_FPTU
        e7. Si token.tipo es C_AR
                Procesar los tokens para un BLOQUE_C_AR
                Construir una estructura para el BLOQUE_C_AR
        e8. Si token.tipo es C
                Procesar los tokens para un BLOQUE_C
                Construir una estructura para el BLOQUE_C
    f. Si el token no es reconocido reportar error
    g. Anexar la estructura a la lista ligada de estructuras para el
       intérprete
    h. Leer un token de la estructura
4. Cerrar la estructura de salida del analizador léxico
```

3.3 La estructura del intérprete

La sentencia de bloque TMP del ejemplo es una de muchas en el lenguaje. La estructura mostrada en la figura 3.2 representando la sentencia es también una de las muchas creadas por el analizador sintáctico. Estas estructuras son solamente una parte de otra estructura más grande (una lista lineal ligada) que representa la totalidad de las sentencias en un programa determinado. El campo Tipo de operación en las estructuras de las sentencias define como se llevará a cabo el procesamiento de la traducción. El resto de los campos de las estructura son parámetros para la función que procesa la sentencia en el intérprete. El intérprete recorre esta lista ligada de estructuras y selecciona la función a ejecutar de la tabla de operaciones.

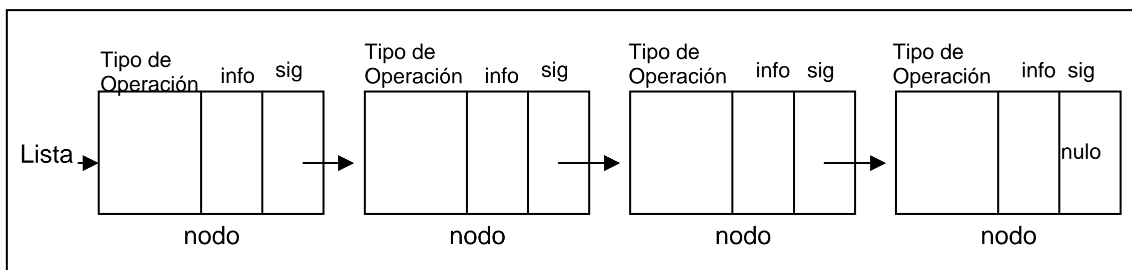


Figura 3.3

Lista lineal ligada con estructuras de sentencias como nodos.

Por ejemplo, hay una función trad_BLOQUE_TMP que contiene el código para manejar el comportamiento definido por la sentencia BLOQUE_TMP del ejemplo anterior. De la misma manera, hay una función de implementación para cada sentencia en el lenguaje: trad_INST_DE_TRABAJO, trad_INST_SALIDA, trad_bloque_TMP, trad_bloque_TRCX, trad_bloque_TRDX, trad_bloque_FPTV, trad_bloque_FPTU, trad_bloque_FPTW, trad_bloque_CAR y trad_bloque_C.

El algoritmo del intérprete se muestra a continuación:

```
1. Ajustar la estructura actual al primer nodo de la lista ligada de
   las estructuras de sentencias del programa.
2. Mientras no se alcance el fin de la lista de estructuras{
   a. Ajustar la función de acuerdo al campo Tipo de operación de la
      estructura
   b. Llamar a la función de traducción con los argumentos devueltos
      por el árbol
   c. Llamar a la siguiente estructura en la lista ligada
   }
```

3.4 Estructuras de datos para el procesador de lenguajes

El analizador desarma las sentencias del lenguaje y produce, para cada sentencia una serie de tokens. Estos tokens definen la función que se ejecutará y los parámetros de la función. Una vez que la sentencia ha sido analizada sintácticamente, sabemos qué es lo que hay que hacer con esta sentencia. Esta información (función y parámetros) es guardada hasta que el código ejecutable se genera. ¿En dónde se guarda la información?

La información recabada de las sentencias del lenguaje se guarda en estructuras de datos especialmente diseñadas. Debido a que las sentencias están ordenadas linealmente, es lógico que puedan ser almacenadas en una estructura de datos más o menos lineal. También debido a que es imposible predecir el número de sentencias en un programa, es razonable que la estructura sea dinámica. Finalmente, debido a que cada sentencia tiene su propio conjunto de parámetros asociado, la estructura debe permitir a los cambiantes elementos

que sean anexados como se requiera. La estructura más conveniente para estos requerimientos es la lista ligada. Las estructuras de datos para este compilador están basadas en las listas ligadas.

La estructura de datos básica

El elemento básico de la lista ligada que contiene el programa interpretado se llama *NodoSentencia*. Un programa interpretado consiste de una lista lineal ligada de este tipo de Nodos. Esta estructura se llama *ListaSentencias*. Ambas estructuras están representadas en la figura 3.3.

El siguiente programa define el NodoSentencia en Java.

```
class NodoSentencia {
      String tipoSentencia;
      Object sentencia;
      NodoSentencia siguienteNodo;
      NodoSentencia( String tipo, Object sentence )
      {
            this( tipo, sentence, null );
      }
      NodoSentencia(String tipo, Object sentence, NodoSentencia nodo)
      {
            tipoSentencia = tipo;
            sentencia = sentence;
            siguienteNodo = nodo;
      }
      NodoSentencia obtenerSiguiente()
      {
      return siguienteNodo;
      }
}
```

El siguiente programa define la lista ligada ListaSentencias.

```
public class ListaSentencias {
      private NodoSentencia primerNodo;
      private NodoSentencia ultimoNodo;
      private String nombre;
      public ListaSentencias()
      {
            this( "Lista de Sentencias" );
      }
      public ListaSentencias( String nombreLista )
      {
            nombre = nombreLista;
            primerNodo = ultimoNodo = null;
      }
      public synchronized void insertarAlFrente( String tipo, Object
elementoInsertar )
      {
      if ( estaVacia() )
            primerNodo = ultimoNodo = new NodoSentencia( tipo,
      elementoInsertar );
      else
            primerNodo = new NodoSentencia( tipo, elementoInsertar,
      primerNodo );
      }
      public synchronized void insertarAlFinal( String tipo, Object
elementoInsertar )
      {
      if ( estaVacia() )
            primerNodo = ultimoNodo = new NodoSentencia( tipo,
      elementoInsertar );
      else
            ultimoNodo = ultimoNodo.siguienteNodo = new NodoSentencia(
      tipo, elementoInsertar );
      }
      public synchronized NodoSentencia eliminarDelFrente() throws
ExcepcionListaVacia
      {
      if ( estaVacia() )
            throw new ExcepcionListaVacia( nombre );
```

```
	NodoSentencia elementoEliminado = primerNodo;
	if ( primerNodo == ultimoNodo )
		primerNodo = ultimoNodo = null;
	else
		primerNodo = primerNodo.siguienteNodo;
	return elementoEliminado;
	}
	public synchronized NodoSentencia eliminarDelFinal() throws
ExcepcionListaVacia
	{
	if ( estaVacia() )
		throw new ExcepcionListaVacia( nombre );
	NodoSentencia elementoEliminado = ultimoNodo;
	if ( primerNodo == ultimoNodo )
		primerNodo = ultimoNodo = null;
	else {
		NodoSentencia actual = primerNodo;

	while ( actual.siguienteNodo != ultimoNodo )
		actual = actual.siguienteNodo;

	ultimoNodo = actual;
	actual.siguienteNodo = null;
	}
	return elementoEliminado;
	}
	public synchronized boolean estaVacia()
	{
		return primerNodo == null;
	}
}
```

La tabla de símbolos

Otra estructura de datos importante es la tabla de símbolos. Normalmente contiene los nombres de los identificadores usados en un programa. Se usa para guardar el valor actual del símbolo y regresarlo cuando sea necesario. La manera en que una tabla de

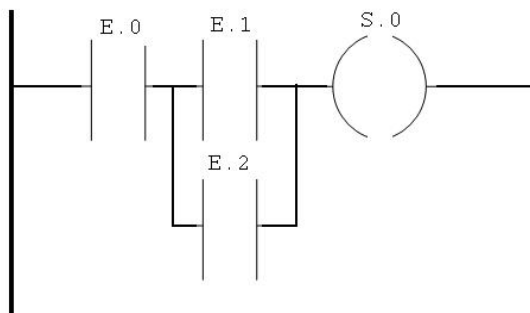
símbolos está organizada afecta la eficiencia. En nuestro caso una organización lineal es suficiente, ya que son pocos identificadores, en caso de llegar a ser más de 100 sería necesaria una estructura de datos que optimizara la búsqueda tal como un árbol binario.

Cada entrada en la tabla de símbolos guarda información acerca de los identificadores (en nuestro caso los bits de entrada, salida, memoria) usados en el programa. Debido a que la tabla crecerá y se contraerá, un tipo dinámico de lista lineal como una lista ligada es apropiado para este propósito.

Mientras el programa está traduciéndose por el compilador, las entradas son creadas en la tabla de símbolos. Antes de que una variable sea creada y agregada a la tabla de símbolos, la tabla de símbolos es revisada para ver si ya ha sido definida en el programa. Si se ha definido, no es creada una nueva entrada en la tabla de símbolos. La tabla de símbolos se accede siempre que una variable es usada en un enunciado.

sentencia_INST_DE_TRABAJO

Esta estructura es muy importante ya que se encarga de realizar el procesamiento de todas las sentencias lógicas. Por este motivo, lo único que guarda además de su Tipo de operación, es otra lista ligada que contiene bits y operadores en notación postfija para poder realizar el cálculo de manera efectiva. Por ejemplo, el siguiente diagrama de escalera



se representa en el lenguaje diseñado para este compilador como:

```
LD E.0
```

```
AND( E.1
OR E.2
)
ST S.0
```

instrucciones que la sentencia_INST_DE_TRABAJO guardará como la siguiente cadena postfija:

```
La Cadena Postfija es:
0           BIT_ENTRADA
1           BIT_ENTRADA
2           BIT_ENTRADA
OR          OPERADOR
AND         OPERADOR
```

A continuación se muestra de manera general el algoritmo usado para el procesamiento de esta sentencia.

```
PILA = vacía
simbl = siguiente token
while (no sea fin de cadena de entrada)
{
      if (simbl es operando)
            agregar simbl a cadena postfija
      else
      { // no es operando, puede ser operador o paréntesis
            while ( !PILA.vacía() && prcdnc (PILA.tope, simbl) )
            {
                  punta = PILA.pop();
                  agregar punta a cadena postfija
            }
            if ( PILA.vacía() || simbl != ')' )
                  PILA.push(simbl)
            else
                  punta = PILA.pop();
      }
```

```
      simbl = siguiente token
}
// extracción de los operadores restantes
while ( !PILA.vacía() )
{
      punta = PILA.pop();
      agregar punta a cadena postfija
}
```

y donde la función *prcdnc* es la encargada de determinar si un operando tiene mayor precedencia sobre el otro.

sentencia_INST_SALIDA

Esta estructura también hace uso de una lista ligada como uno de sus elementos. Sin embargo, esta lista no siempre es necesaria y sólo se usa cuando a partir de una sentencia INST_DE_TRABAJO se requieren encender dos o más bobinas de salida. Los datos que guarda la estructura son:

```
NodoListaTokens BIT;
ListaSalidas SECUENCIA_SALIDAS;
```

El objeto *ListaSalidas* también es una lista ligada lineal y está formado por nodos *Salida* que a su vez contienen los siguientes elementos:

```
NodoListaTokens ASIGNADOR;
NodoListaTokens BIT;
```

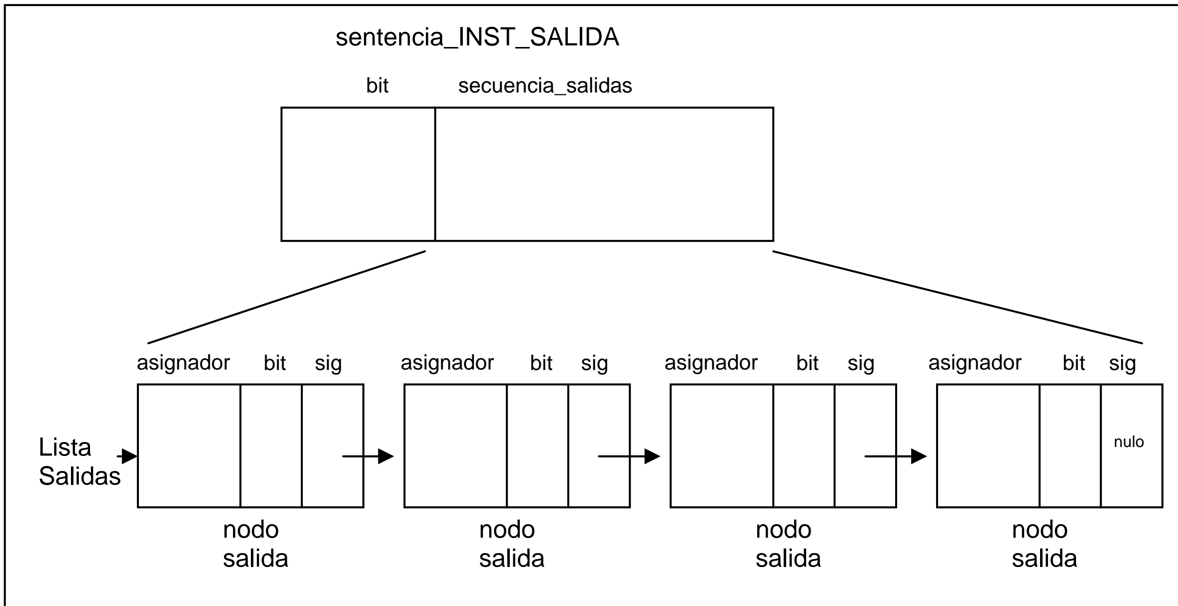


Figura 4.4

sentencia_INST_SALIDA

sentencia_BLOQUE

Por razones de implementación esta sentencia contiene a todas las sentencias bloque contempladas en el lenguaje. Los datos que contiene la estructura son:

```
String tipo;
bloque_TMP bTMP;
bloque_TRCX bTRCX;
bloque_TRDX bTRDX;
bloque_FPTV bFPTV;
bloque_FPTU bFPTU;
bloque_FPTW bFPTW;
bloque_CAR bCAR;
bloque_C bC;
```

Al realizar el procesamiento de esta sentencia, lo primero que se hace es buscar el tipo de bloque. Una vez localizado, se llama a la función que procesará el bloque correcto y se crea la estructura. El resto de las sentencias permanece sin alteración y no afectan la estructura del programa ya que el tipo se encarga de direccionar la estructura indicada.

Cada estructura almacena datos distintos, de acuerdo a las entradas y los procesos que desarrolla el PLC. A continuación se muestran los datos relevantes para la construcción de las estructuras de bloque.

sentencia bloque_TMP

```
String num;
String valor;
sentencia_INST_DE_TRABAJO instEnt, instSld;
sentencia_INST_SALIDA instAsg;
// temporizador que permite elaborar un pulso de duración precisa.
```

sentencia bloque_TRCX bTRCX

```
String num;
String valor;
sentencia_INST_DE_TRABAJO instEnt, instSld;
sentencia_INST_SALIDA instAsg;
// temporizador que permite gestionar retardos en la conexión
```

sentencia bloque_TRDX bTRDX

```
String num;
String valor;
sentencia_INST_DE_TRABAJO instEnt, instSld;
sentencia_INST_SALIDA instAsg;
// temporizador que permite gestionar retardos en la desconexión
```

sentencia bloque_FPTV bFPTV

```
String num;
String valor;
sentencia_INST_DE_TRABAJO instEnt, instSld;
sentencia_INST_SALIDA instAsg;
// Flip-flop con retardo en la entrada.
```

sentencia bloque_FPTU bFPTU

```
String num;
String valor;
```

```
sentencia_INST_DE_TRABAJO instESET, instERST, instSQ;
sentencia_INST_SALIDA instAsg;
// Flip-flop con retardo en la salida
```

sentencia bloque_FPTW bFPTW

```
String num;
String valor;
sentencia_INST_DE_TRABAJO instESET, instERST, instSQ;
sentencia_INST_SALIDA instAsg;
// Flip-flop con retardo en la entrada "memorizado"
```

sentencia bloque_CAR bCAR

```
String num;
String valorEVC, valorEVD;
sentencia_INST_DE_TRABAJO instERST, instEAOD, instECC, instSSPA;
sentencia_INST_SALIDA instAsgSPA;
// Realiza el conteo o desconteo de eventos
```

sentencia bloque_C bC

```
String num;
String valor;
sentencia_INST_DE_TRABAJO instER, instECC, instEDD, instSSPA,
      instSSDV, instSSDL;
sentencia_INST_SALIDA instAsgSPA, instAsgSDV, instAsgSDL;
// Realiza el conteo o desconteo de eventos, incluye salidas
// para indicar desbordamientos
```

3.5 Proceso de compilación

Como se explicó en los requerimientos, para correr este programa compilador se requiere que esté instalada la máquina virtual de Java, específicamente el ambiente de ejecución.

Pasos para abrir el ambiente del compilador:

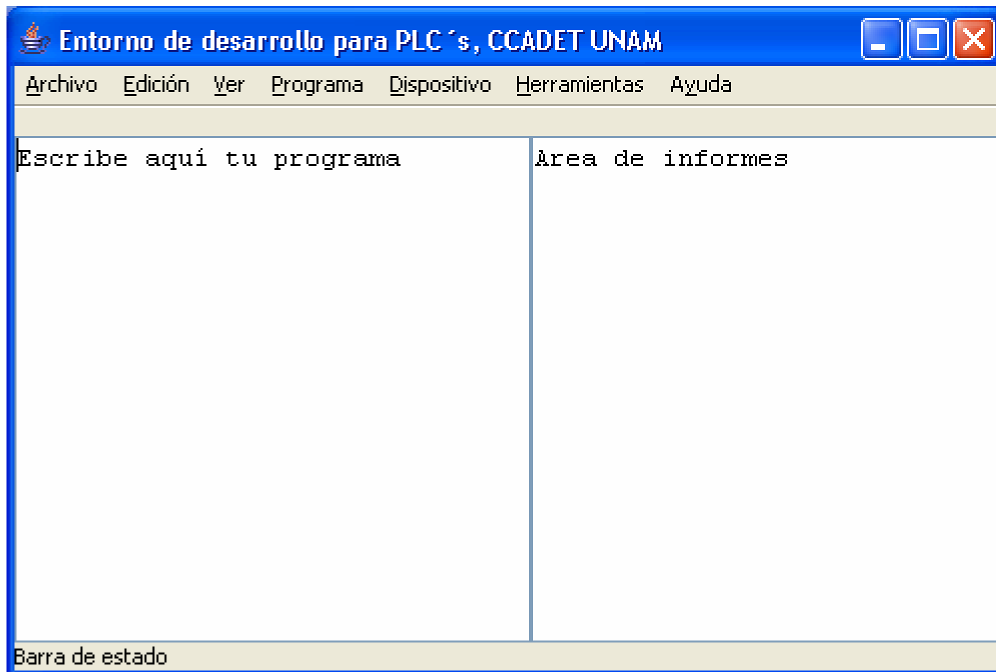
1. Abrir el ambiente de comandos de Windows (símbolo del sistema).

2. Colocarse dentro de la carpeta donde se encuentra el archivo cplc.class.
3. Llamar a la máquina virtual de java con la instrucción: java cplc.

Ejemplo:

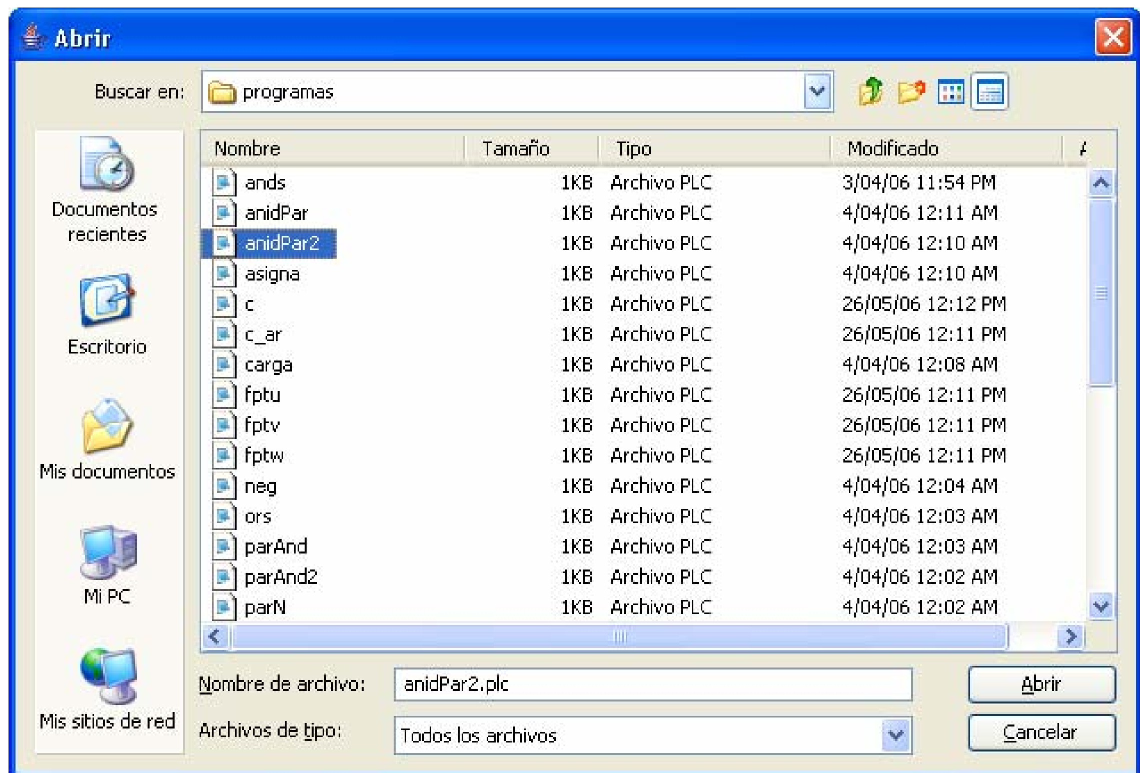
```
Símbolo del sistema - java cplc
Microsoft Windows XP [Versión 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\max>cd escritorio

C:\Documents and Settings\max\Escritorio>cd rt

C:\Documents and Settings\max\Escritorio\rt>java cplc
```

Esto nos desplegará la siguiente pantalla:

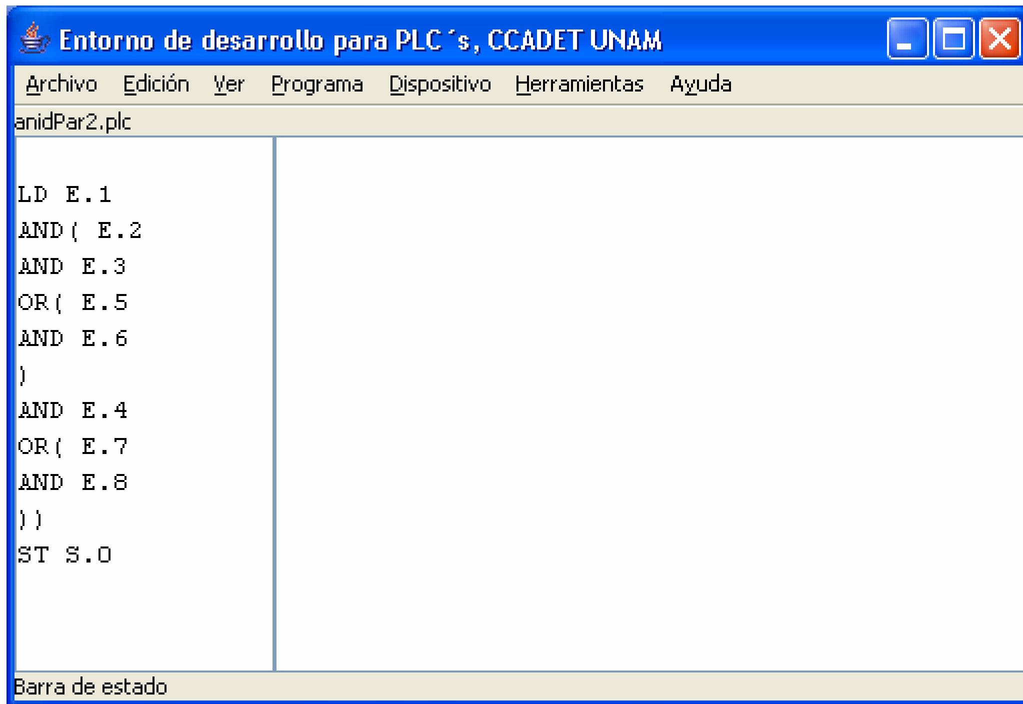


A partir de este momento podemos escribir nuestro programa en lenguaje de lista de instrucciones como se describe en la sección 1.2 Conjunto de instrucciones o abrir algún programa escrito previamente.

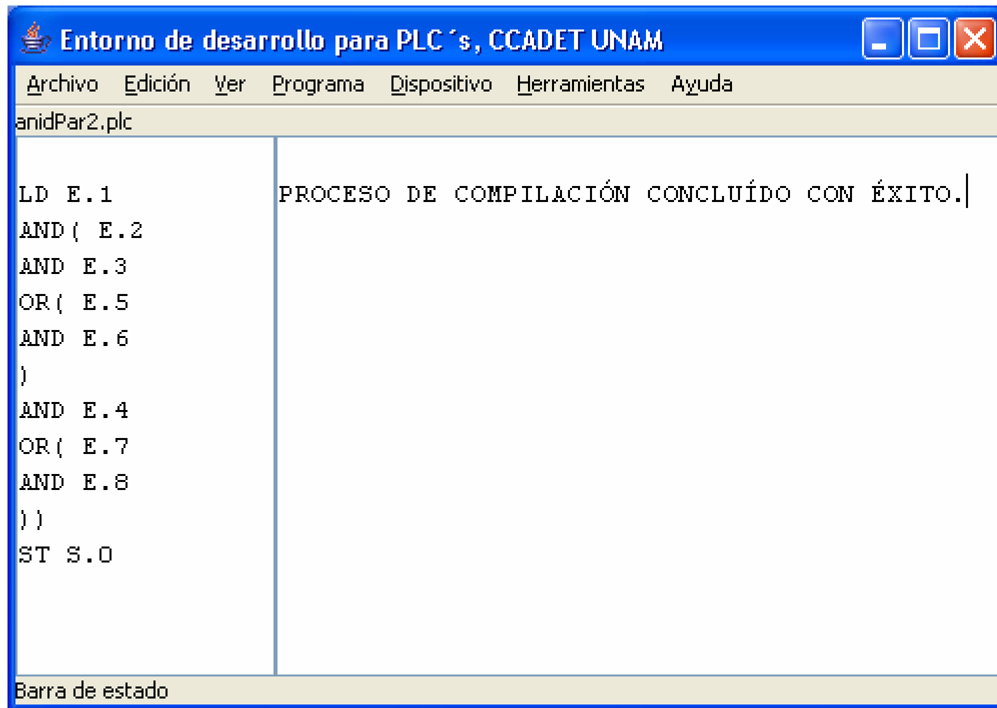
En el caso de que se decida trabajar con un programa ya escrito, es necesario usar el botón **Abrir** en el menú **Archivo**. Esto nos desplegará una ventana para elegir nuestro programa:



Una vez seleccionado, el programa se muestra en la zona de escritura:



Para compilar, es necesario usar la opción **Compilar** del menú **Programa**. Esta acción realizará el proceso completo de compilación: análisis léxico, sintáctico y traducción a código de máquina. De resultar correcto el código desplegará el siguiente mensaje en el área de informes.



Cabe señalar que el proceso de análisis puede ser observado en la pantalla del símbolo del sistema, en donde se muestran paso a paso los tokens detectados por el analizador léxico y el resto de las estructuras creadas. Esta pantalla sin embargo se usa simplemente con fines de depuración ya que el proceso debe ser transparente para el usuario.

Símbolo del sistema - java cplc

```
C:\Documents and Settings\max\Escritorio\rt>java cplc
La Lista de Tokens es:
LD        CARGA
1         BIT_ENTRADA
AND       OPERADOR
(         SIMBOLO
2         BIT_ENTRADA
AND       OPERADOR
3         BIT_ENTRADA
OR        OPERADOR
(         SIMBOLO
5         BIT_ENTRADA
AND       OPERADOR
6         BIT_ENTRADA
)         SIMBOLO
AND       OPERADOR
4         BIT_ENTRADA
OR        OPERADOR
(         SIMBOLO
7         BIT_ENTRADA
AND       OPERADOR
8         BIT_ENTRADA
)         SIMBOLO
)         SIMBOLO
ST        ASIGNACION
0         BIT_SALIDA
          FIN_DE_TABLA


LD                CARGA
La Cadena Postfija es:
1         BIT_ENTRADA
2         BIT_ENTRADA
3         BIT_ENTRADA
AND       OPERADOR
5         BIT_ENTRADA
6         BIT_ENTRADA
AND       OPERADOR
4         BIT_ENTRADA
AND       OPERADOR
OR        OPERADOR
7         BIT_ENTRADA
8         BIT_ENTRADA
AND       OPERADOR
OR        OPERADOR
AND       OPERADOR


                  sentencia_INST_DE_TRABAJO RECONOCIDA Y AGREGADA
ST                ASIGNACION
 ejecutandoAnalisis.sentencia_INST_SALIDA
                  sentencia_INST_SALIDA RECONOCIDA Y AGREGADA
```

CAPÍTULO 4. PRUEBAS DEL SISTEMA Y MANTENIMIENTO

Las pruebas son parte importante en el desarrollo de sistemas. Se realizan con el propósito de encontrar fallas y se establecen para mejorar la calidad del sistema. En las pruebas se requiere que se descarten las ideas acerca de lo que se desarrolló en el sistema, ya que al descubrir los errores, se logra superar cualquier conflicto en el sistema.

La prueba ideal de un sistema sería exponerlo en todas las situaciones posibles, así encontraríamos hasta la última falla, de tal manera que es posible garantizar su respuesta ante cualquier caso que se le presente en la ejecución real. Esto es imposible desde todos los puntos de vista: humano, económico y matemático.

Dado que todo es finito en programación (el número de líneas de código, el número de variables, etc.) es posible pensar que el número de pruebas posibles es finito. Esto deja de ser cierto cuando entran en juego los ciclos, en los que es fácil introducir condiciones para un funcionamiento sin fin. Aún en el caso de que el número de posibilidades fuera finito, el número de combinaciones posibles es tan grande que se hace imposible su identificación y ejecución a todos los efectos prácticos.

Los objetivos de las pruebas son:

- Asegurar la obtención y formalización de los requerimientos del usuario y verificar que son adquiridos de una manera completa y correcta.
- Mostar errores no descubiertos en los requerimientos establecidos.
- Obtener un conjunto de pruebas que tengan la mayor probabilidad de descubrir los defectos del sistema.
- Ejecutar el programa con la intención de descubrir un error.

Las pruebas se centran en los procesos lógicos internos del programa, asegurando que todas las sentencias se han comprobado, realizándose para poder detectar errores, cuando se produzcan resultados reales de acuerdo con los resultados requeridos.

Las pruebas pueden dividirse en distintas categorías, estas pueden ser: caja negra, caja blanca, unitarias, de integración y de regresión.

4.1 Pruebas de caja negra

También denominada prueba de comportamiento, las pruebas de caja negra se emplean conociendo la función específica para la que fue diseñado el producto. Se pueden llevar a cabo pruebas que demuestren que cada función es completamente operativa y al mismo tiempo buscando errores en cada función.

Estas pruebas son diseñadas para validar los requisitos funcionales sin fijarse en el funcionamiento interno del programa.

Las pruebas de caja negra están especialmente indicadas en aquellos módulos que van a ser interfaz con el usuario (en sentido general: teclado, pantalla, archivo, canales de comunicación, etc.).

Las pruebas de caja negra intentan encontrar errores de las siguientes categorías:

1. Funciones incorrectas o ausentes.
2. Errores de interfaz.
3. Errores de estructuras de datos o en acceso a bases de datos externas.
4. Errores de rendimiento.
5. Errores de inicialización.

El probador se limita a suministrarle datos como entrada y estudiar la salida.

4.1.1 Limitaciones de las pruebas de caja negra

Lograr una buena cobertura con pruebas de caja negra es un objetivo deseable pero no suficiente a todos los defectos. Un programa puede pasar con una holgura millones de pruebas y sin embargo tener defectos internos que surgen en el momento más inoportuno.

4.2 Pruebas de caja blanca

Se centra en la estructura de control del programa, basado en un minucioso examen de detalles de procedimientos donde se puede verificar el estado del programa en varios puntos para determinar si el real coincide con el esperado. Con las pruebas de caja blanca se puede comprobar que:

1. Todos y cada uno de los caminos se llevan a cabo de manera correcta.
2. Se lleva a cabo cada una de las decisiones lógicas.
3. Se ejecuten todos los bucles en sus límites operacionales.
4. Se ejerciten las estructuras internas de datos para asegurar su validez.

4.2.1 Limitaciones de las pruebas de caja blanca

Lograr una buena cobertura con pruebas de caja blanca también es un objetivo deseable pero no suficiente a todos los defectos. Un programa puede estar perfecto en todos sus términos, y sin embargo no servir a la función que se pretende ejecutar.

Las pruebas de caja blanca nos convencen de que un programa hace bien lo que hace pero no de que haga lo que el usuario necesita.

4.3 Pruebas unitarias

Son las pruebas que centran el proceso de verificación sobre un programa o módulo con la finalidad de encontrar problemas en la lógica y problemas técnicos en el código.

4.4 Pruebas de integración

La prueba de integración, es una técnica sistemática para que la estructura del programa esté de acuerdo con lo que dictó el diseño.

4.5 Pruebas de regresión

Ayudan a asegurar que los cambios (debidos a las pruebas o por otros motivos) no introducen un comportamiento no deseado de errores adicionales.

Con este tipo de pruebas se permite detectar fallas que se hayan introducido durante las modificaciones a un sistema, permitiendo verificar que dichas modificaciones no impacten y que sigan cumpliendo con los requerimientos planteados.

4.6 Consideraciones para la ejecución de las pruebas

Es importante tomar en cuenta algunas consideraciones al realizar las pruebas, para así obtener mejores resultados en la detección de errores cometidos al desarrollar el sistema.

Riesgos. Los riesgos son aquellos elementos que pueden afectar negativamente la ejecución de las pruebas.

Suposiciones. Son las premisas que puedan afectar positiva o negativamente la ejecución de las pruebas complicando o facilitando las actividades de las pruebas.

Condiciones. En esta consideración se llevan a cabo la definición de datos, funciones y comportamiento al implementar el sistema, así como la información sobre el rendimiento que delimita el sistema.

Restricciones. Las restricciones están relacionadas con el desarrollo del proyecto en sí: la tecnología de pruebas, el estado del ambiente de pruebas.

4.7 Pruebas aplicadas

Pruebas de caja negra

Para esta prueba se probaron específicamente las partes del programa en las que se tiene interacción con el usuario. Específicamente son dos, la carga del programa y el proceso de compilación activado por la opción **Compilar** del menú Programa. La carga del programa puede ser desde un archivo o escribiéndolo en el área dedicada a ese fin. En ningún caso se detectaron problemas o errores. El proceso de verificación también trabaja de acuerdo a lo planeado aún cuando se presentan entradas equivocadas.

Pruebas de caja blanca

Estas pruebas fueron aplicadas durante el desarrollo del software, en cada una de las clases mientras iban siendo programadas.

Pruebas unitarias

Se aplicaron a cada clase del programa para verificar su buen funcionamiento. Se revisaron cada una de las sentencias que puede aceptar la gramática del compilador así como las estructuras creadas por las sentencias. Esta fase fue una repetición de las pruebas de caja blanca.

Pruebas de integración

Debido a que se trabajó con un enfoque orientado a objetos, la integración de todas las clases es muy importante. En la comunicación de las clases no se encontró ningún error y operaron correctamente en su conjunto.

Pruebas de regresión

El sistema no requirió cambios que afectaran el comportamiento del compilador. Por lo que esta prueba consistió únicamente en realizar nuevamente las pruebas de caja negra.

4.8 Mantenimiento

El mantenimiento es la última fase del proceso de ingeniería de software. Puede ser la parte más costosa ya que a esta parte se le puede dedicar hasta un 70% de los esfuerzos.

Se llevan a cabo tres tipos de mantenimiento:

1. Mantenimiento correctivo. Existen algunos errores en el compilador que no fueron detectados durante la fase de pruebas pero han sido encontrados con el tiempo y uso del sistema.
2. Mantenimiento adaptativo. En esta actividad se cambian, mejoran o modifican los elementos del sistema debido a nuevas versiones de sistemas operativos de manera que el sistema pueda interactuar de manera correcta con el entorno cambiante.
3. Mantenimiento perfectivo. Se presenta cuando el usuario utiliza el sistema y se da cuenta de que puede mejorar, agregar nuevas funciones, etc. También se da cuando se cambia el software para se pueda adecuar a mejoras futuras.

El mantenimiento se asocia a algunos problemas: muchas veces la falta de disciplina durante el desarrollo casi siempre trae problemas en la etapa de mantenimiento. Algunos de los problemas que trae consigo son:

- A veces resulta difícil seguir una evolución de software a través de varias versiones ya que algunos cambios no son documentados.
- No existe documentación adecuada.
- Algunas veces el software no se diseña pensando en un futuro cambio.

El mantenimiento puede ser algo riesgoso, se debe tener cuidado con las sentencias que se cambian, ya que el cambio de alguna de estas puede cambiar completamente el concepto del sistema.

Alguna de las desventajas que se presentan en la etapa tanto de pruebas como de mantenimiento además de los costos son:

- Insatisfacción del cliente cuando una petición de reparacion o de modificación aparentemente legítima no se puede atender en un tiempo razonable.
- Disminución de la calidad del software debido a los errores latentes que introducen los cambios en el software.
- Transtornos en otros esfuerzos de desarrollo al tener que poner a trabar a la plantilla de mantenimiento.

A pesar de que el mantenimiento es una fase complicada y costosa es necesario aplicar ese paso ya que la mayoría de los sistemas necesitan modificaciones y actualizaciones.

CONCLUSIONES

El compilador objeto de este ejercicio facilita el proceso de programación del PLC diseñado por el Laboratorio de Electrónica del CCADET.

Este programa trabaja a partir de listas de instrucciones, mejorando la entrada por lenguaje de máquina con que se trabaja actualmente. Esta primera versión del programa, aún no contiene la herramienta de programación visual que se encuentra en los programadores comerciales, sin embargo, la forma de la gramática permitirá cambiar de modo texto a modo gráfico con facilidad en las siguientes mejoras.

Las clases del compilador pueden ser adaptadas a otra interfaz o a otro programa Java para lograr la programación de manera remota vía web. Esto, debido al alcance de la máquina virtual de Java.

Al saber que las instrucciones que se almacenarán en el PLC ya han sido revisadas por el compilador, se tiene la certeza de no dañar el PLC por código incorrecto.

El enfoque orientado a objetos permitió que el diseño no fuera tan complicado, ya que todos los elementos del programa pueden comunicarse entre sí. Hacer lo mismo bajo un enfoque de programación estructurada implicaría más esfuerzo. Cabe mencionar que la bibliografía sobre programación de compiladores trabajando en un lenguaje orientado a objetos es muy escasa y cuando se localiza, el enfoque es demasiado abstracto.

GLOSARIO

API. (del inglés **Application Programming Interface) Interfaz de Programación de Aplicaciones**. Es un conjunto de especificaciones de comunicación entre componentes de software. Representa un método para conseguir abstracción en la programación, generalmente entre los niveles o capas inferiores y los superiores del software. Uno de los principales propósitos de una API consiste en proporcionar un conjunto de funciones de uso general, por ejemplo, para dibujar ventanas o iconos en la pantalla. De esta forma, los programadores se benefician de las ventajas de la API haciendo uso de su funcionalidad, evitándose el trabajo de programar todo desde el principio. Las APIs asimismo son abstractas: el software que proporciona una cierta API generalmente es llamado la implementación de esa API.

Autómata. Del griego *automatos* que significa *espontáneo* o *con movimiento propio*. En computación, un **autómata finito** o **máquina de estado finito** es un modelo matemático de un sistema que recibe una cadena constituida por símbolos de un alfabeto y determina si esa cadea pertenece al lenguaje que el autómata reconoce. Formalmente, un autómata finito (AF) puede ser descrito como una 5-tupla (S,Σ,T,s,A) donde: Σ es un alfabeto; S un conjunto de estados; T es la función de transición; $s \in S$ es el estado inicial; $A \in S$ es un conjunto de estados de aceptación o finales.

Botón pulsador. Botón que controla directamente un contacto de entrada al PLC.

Bytecode. Representación binaria de un programa ejecutable diseñado para ser ejecutado por una máquina virtual más que por hardware. Debido a qué es ejecutado por software, es usualmente más abstracto que el código de máquina.

Compilador. Es un programa que lee un programa escrito en un lenguaje de alto nivel, y lo traduce a un programa equivalente a otro lenguaje de bajo nivel. Como parte importante de

este proceso de traducción, el compilador informa al usuario de la presencia de errores en el programa fuente.

Contador. Circuito secuencial construido a partir de flip-flops y compuertas lógicas capaz de realizar el cómputo de los impulsos que recibe en la entrada destinada a tal efecto, almacenar datos o actuar como divisor de frecuencia.

Gramática. Conjunto de reglas y normas que se proponen para el uso correcto de un lenguaje determinado.

Flip-flop. Dispositivos biestables síncronos. En este caso, el término síncrono significa que la salida varía de estado únicamente en un instante específico de una entrada de disparo denominada reloj.

Hyperterminal. Programa de comunicaciones incluido en varias versiones del sistema operativo Windows. Esta diseñada para emular varios tipos de terminales de texto. Puede ser configurada para hacer comunicaciones a través de un modem o directamente sobre un puerto serial.

Intérprete. En lugar de producir un programa objeto como resultado de una traducción, un intérprete realiza las operaciones que implica el programa fuente. Muchas veces los intérpretes se usan para ejecutar lenguajes declarativos, pues cada operador que se ejecuta en esos lenguajes suele ser una invocación de una rutina compleja o definir un tamaño de memoria requerido que no se puede deducir en el momento de la compilación.

Java. Lenguaje de programación desarrollado por James Gosling y sus compañeros de Sun Microsystems al inicio de la década de 1990. A diferencia de los lenguajes de programación convencionales, que generalmente están diseñados para ser compilados a código nativo, Java es compilado en un bytecode que es ejecutado por una máquina virtual Java. El lenguaje en sí mismo toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos mucho más simple y elimina herramientas de bajo nivel como punteros.

JVM (del inglés **Java Virtual Machine**). Desarrollada por Sun Microsystems, es una máquina virtual que ejecuta el código resultante de la compilación de un programa escrito en Java, conocido como bytecode de Java.

Lenguaje. En términos generales, el lenguaje designa todas las comunicaciones animales basadas en la interpretación, incluyendo el lenguaje humano. En matemáticas y computación, los lenguajes artificiales son llamados lenguajes formales.

Máquina Virtual. Software que crea un entorno virtual entre la plataforma de la computadora y el usuario final, permitiendo que este ejecute un software determinado.

PLC. (Controlador Lógico Programable, por sus siglas en inglés) Computadora electrónica usada para automatización de procesos industriales, como el control de la maquinaria en una línea de ensamblado en una fábrica. A diferencia de las computadoras de propósito general, un PLC está diseñado para soportar elevados niveles de temperatura, condiciones de polvo o suciedad, inmunidad al ruido electrónico, y es mecánicamente más resistente a las vibraciones y a los impactos.

Relevador. Dispositivo electromecánico que funciona como un interruptor controlado por un circuito eléctrico en el que, por medio de un electroimán, se acciona un juego de uno o varios contactos que permiten abrir o cerrar otros circuitos eléctricos independientes. Debido a que es capaz de controlar un circuito de salida de mayor potencia que el de entrada, puede considerarse, una forma de amplificador eléctrico.

Sensor. Dispositivo que detecta manifestaciones de cualidades o fenómenos físicos, como la energía, velocidad, aceleración, tamaño, cantidad, etc. Un sensor es un tipo de transductor que transforma la magnitud que se quiere medir, en otra, que facilita su medida.

Temporizador. Dispositivo capaz de controlar el inicio, la duración y el término de un evento.

Token. Palabra o elemento atómico dentro de una secuencia de caracteres o cadenas.

Transductor. Dispositivo capaz de transformar o convertir un determinado tipo de energía de entrada, en otra diferente de salida. Ejemplo. Un micrófono es un transductor electroacústico que convierte la energía acústica (vibraciones sonoras: oscilaciones en la presión del aire) en energía eléctrica (variaciones de voltaje).

BIBLIOGRAFÍA

Deitel, Harvey M; Deitel, Paul J. Java, How to program. 6th edition. New Jersey, Prentice Hall, 2002.

Booch, Grady; Rumbaugh, James; Jacobson, Ivar. El lenguaje unificado de modelado. Addison Wesley. Madrid, 1999

Brookshear, J. Glenn; Teoría de la computación. Lenguajes formales, autómatas y complejidad. Addison-Wesley Iberoamericana. 1993.

Kaplan, Randy M. Constructing Language Processors for Little Languages. USA. John Wiley & Sons, Inc. 1994. 1st edition.

Pressman, Roger S. Ingeniería de Software, un enfoque práctico. Carlos Cervigón Ruchauer, Luis Hernández Yánez. España, Mc Graw Hill, 1993, 3ª Edición.

Sommerville, Ian. Software engineering. USA. Pearson Education, 2001, Sixth Edition.

Tenenbaum, Aaron M.; Langsam, Yedidyah; Augenstein, Moshe A. Esructuras de datos en C. Javier Pulido Cejudo, Raymundo H. Rangel Gutiérrez. México, Prentice-Hall, 1993, 1ª Edición.

Webb, John W. Programmable logic controllers: principles and applications, USA. Macmillan Publishing Company. 1992. 2nd edition.