UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES
ARAGÓN

COMPILADOR EN ESPAÑOL

BASADO EN

LENGUAJE JAVA EN ESPAÑOL

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

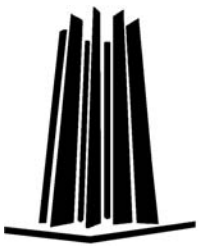
INGENIERO EN COMPUTACIÓN

P R E S E N T A :

M I G U E L A N G E L L Ó P E Z T E C I L L O

ASESOR:

M. EN C. MARCELO PÉREZ MEDEL



MÉXICO, 2008.

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE ESTUDIOS SUPERIORES
ARAGÓN

Compilador en español

Basado en

Lenguaje Java en Español

Tesis que para obtener el título de Ingeniero en Computación presenta:

MIGUEL ANGEL LÓPEZ TECILLO

Asesor:

M. en C. Marcelo Pérez Medel

Sínodos:

M. en I. Arcelia Bernal Díaz

M. en C. Jesús Hernández Cabrera

Ing. Ares Missael Morales Juárez

Ing. Gabriel Ortíz Cordero

México, Octubre de 2008

Jesús y Yolanda, Mis padres, me dieron La vida y me ofrecieron sin condición todo lo que tenían a su alcance: Amor, techo, abrigo, alimento y educación.

Claudia, Jorge y Alberto, Mis hermanos y sin duda alguna mis más grandes compañeros de juego..

Letty, Mi amada esposa, Mi Psicóloga particular. Amor, apoyo, y paciencia. Sin ti, quizá me hubiera faltado coraje para afrontar y superar algunos momentos dificiles de mi vida.

Mi hijo, Un Ángel que, desde el cielo en todo momento nos cuida y acompaña.

Familiares, Amigos y Profesores que de forma positiva han influido en el curso que ahora sigue mi vida.

Finalizo la carrera de Ingeniería en Computación y al alcanzar la meta que me propuse hace ya algunos años, hago un alto para mirar el camino recorrido y me doy cuenta de que apenas he avanzado un escalón más en mi vida. Uno muy importante, del cual quiero agradecer y hacer partícipes a todos ustedes; pues a pesar de mis desaciertos, siempre estuvieron ahí a mi lado apoyándome, guiándome y alentándome desde sus diferentes perspectivas.

A todos ustedes muchas Gracias.

Siempre estarán en mi corazón.

Índice

INTRODUCCIÓN **6**

PRIMERA PARTE **12**

1 TEORÍA DE LENGUAJES FORMALES Y AUTÓMATAS **13**

1.1 CONCEPTOS 14
1.1.1 Símbolo *14*
1.1.2 Palabra *14*
1.1.3 Longitud de una cadena *15*
1.1.4 Prefijo y sufijo de una cadena *15*
1.1.5 Concatenación de cadenas *15*
1.1.6 Alfabeto *16*
1.1.7 Lenguaje *16*
1.2 GRAMÁTICAS DE LIBRE CONTEXTO 16
1.3 EXPRESIONES REGULARES 19
1.4 AUTÓMATAS 20
1.4.1 Diagramas de transiciones *21*
1.4.1.1 Autómatas finitos 22

2 COMPILADORES **23**

2.1 DEFINICIÓN 23
2.2 CÓDIGOS OBJETO 26
2.3 ANÁLISIS Y SÍNTESIS EN LA COMPILACIÓN 27
2.4 FASES DE UN COMPILADOR 28
2.4.1 Análisis léxico *29*
2.4.1.1 Especificación de los componentes léxicos 30
2.4.2 Análisis sintáctico *32*
2.4.2.1 Árboles de análisis sintáctico 34
2.4.2.2 Análisis sintáctico descendente 37
2.4.2.2.1 Análisis sintáctico por descenso recursivo 37
2.4.2.2.2 Analizador sintáctico predictivo 39
2.4.2.3 Análisis sintáctico ascendente 39
2.4.3 Análisis semántico *40*
2.4.4 Generación de código intermedio *42*
2.4.4.1 Árboles binarios 44
2.4.4.1.1 Recorrido de un árbol 45
2.4.5 Optimización de código *47*
2.4.6 Generación de código *48*
2.4.7 Detección y manejo de errores *49*
2.4.8 Administrador de la tabla de símbolos *49*
2.5 APLICACIONES DE LOS COMPILADORES 50

SEGUNDA PARTE **51**

3 ORIENTACIÓN A OBJETOS **52**

3.1 PARADIGMAS DE LA PROGRAMACIÓN 52
3.2 PROGRAMACIÓN ORIENTADA A OBJETOS 54
3.2.1 Objetivos de la programación orientada a objetos *55*
3.2.2 Conceptos básicos dentro de la POO *56*
3.2.2.1 Objeto 56
3.2.2.2 Clase 57
3.2.2.2.1 Instancia 57
3.2.2.2.2 Atributos 58
3.2.2.2.3 Comportamiento 58

3.2.2.3 Encapsulamiento ... 58
3.2.2.3.1 Abstracción ... 59
3.2.2.3.2 Ocultamiento de la implementación ... 60
3.2.2.3.3 División de la responsabilidad ... 60
3.2.2.4 Herencia ... 61
3.2.2.5 Polimorfismo ... 63

4 JAVA ... 65

4.1 EL LENGUAJE JAVA ... 66
4.1.1 Universo de palabras reservadas del Lenguaje Java ... 67
4.1.1.1 Modificadores de acceso ... 67
4.1.1.2 Modificadores de clase, método y variable ... 68
4.1.1.3 Control de flujo ... 68
4.1.1.4 Control de paquete ... 69
4.1.1.5 Datos primitivos ... 69
4.1.1.6 Tipo de retorno void ... 70
4.1.2 Estructura de una clase en Java ... 70
4.1.3 Estructura de una interfaz en Java ... 70
4.1.3.1 Declaración de paquete ... 71
4.1.3.2 Declaración de clases a importar ... 71
4.1.3.3 Declaración de clase ... 72
4.1.3.4 Cuerpo de la clase ... 73
4.1.3.4.1 Atributo ... 74
4.1.3.4.2 Método ... 74
4.1.3.4.3 Constructor ... 76
4.2 COMPILACIÓN E INTERPRETACIÓN DE PROGRAMAS CON JAVA ... 77
4.2.1 Compilador Java ... 78
4.2.2 Consideraciones del Compilador Java ... 80
4.2.3 Java Virtual Machine ... 81

TERCERA PARTE ... 83

5 LENGUAJE JAVA EN ESPAÑOL ... 84

5.1 ALFABETO DEL LENGUAJE JAVA EN ESPAÑOL ... 85
5.1.1 Palabras reservadas ... 85
5.2 DEFINICIÓN DEL LENGUAJE JAVA EN ESPAÑOL ... 86
5.2.1 Clase ... 86
5.2.1.1 Comentarios ... 87
5.2.1.2 Declaración de paquete ... 88
5.2.1.3 Declaración importar ... 89
5.2.1.4 Declaración de interfaz ... 90
5.2.1.5 Declaración de clase ... 91
5.2.1.6 Cuerpo de una interfaz ... 93
5.2.1.7 Cuerpo de una clase ... 94
5.2.1.8 Cuerpo de un método ... 97
5.2.1.8.1 Sentencias de asignación ... 97
5.2.1.8.2 Sentencia de impresión ... 102
5.2.1.8.3 Sentencia de ejecución de método ... 102
5.2.1.8.4 Estructuras de Control ... 103
5.2.1.9 Tipos de datos ... 106
5.2.1.10 Identificador ... 106
5.2.2 Código fuente empleando las reglas sintácticas ... 107
5.3 REGLAS SEMÁNTICAS ... 108
5.3.1 Código fuente empleando las reglas semánticas ... 112

6 COMPILADOR EN ESPAÑOL ... 113

6.1 PLAN DE TRABAJO ... 113
6.1.1 Integración Lenguaje Java Español – Compilador en Español ... 114
6.2 DISEÑO DE COMPILADOR EN ESPAÑOL ... 117
6.3 PRIMERA PASADA ... 118

6.3.1 Control y manejo de errores ... *119*
6.3.1.1 Diseño del controlador de errores ... 119
6.3.2 Análisis léxico ... *120*
6.3.2.1 Diseño de un analizador léxico ... 122
6.3.3 Análisis sintáctico ... *124*
6.3.3.1 Diseño del analizador sintáctico (Primera pasada) ... 126
6.3.3.1.1 GLC para la declaración de paquete ... 129
6.3.3.1.2 GLC para la declaración importar ... 130
6.3.3.1.3 GLC para la declaración de clase ... 130
6.3.3.1.4 GLC para declaración de atributos y métodos ... 131
6.4 SEGUNDA PASADA ... 131
6.4.1 Análisis sintáctico ... *132*
6.4.1.1 Diseño del analizador sintáctico (Segunda pasada) ... 132
6.4.1.1.1 GLC para verificar expresiones ... 134
6.4.1.1.2 GLC para verificar asignaciones ... 135
6.4.1.1.3 GLC para verificar sentencias de ejecución de método ... 137
6.4.1.1.4 GLC para verificar sentencias de control de flujo ... 137
6.4.1.1.5 GLC para verificar sentencias repetitivas ... 138
6.5 TERCERA PASADA ... 138
6.5.1 Analizador semántico ... *139*
6.5.1.1 Diseño de analizador semántico ... 140
6.5.1.1.1 GLC de apoyo en el análisis semántico ... 142
6.5.1.1.2 GLC para verificar tipos de datos boleanos ... 143
6.5.1.1.3 GLC para verificar tipos de datos carácter ... 143
6.5.1.1.4 GLC para verificar tipos de datos texto ... 144
6.5.1.1.5 GLC para verificar tipos de datos real y entero ... 144
6.6 CUARTA PASADA ... 145
6.6.1 Generación de código intermedio ... *145*
6.6.2 Optimización de código ... *146*
6.6.3 Generación de código ... *147*
6.6.3.1 Equivalencias en el proceso de generación de código ... 147
6.7 IMPLEMENTACIÓN ... 150
6.7.1 Requerimientos del sistema ... *151*
6.8 COMPILADOR EN ESPAÑOL EN ACCIÓN ... 151
6.8.1 Configuración del Compilador en Español ... *151*
6.8.2 Uso del Compilador en Español ... *152*
6.8.3 Ejemplos acerca del uso del Compilador en Español ... *154*
6.8.3.1 Ejemplo de una compilación exitosa ... 154
6.8.3.2 Ejemplo de una compilación con errores sintácticos ... 155
6.8.3.3 Ejemplo de una compilación con errores semánticos ... 157
6.8.3.4 Resumen de la primera pasada ... 159
6.8.3.5 Resumen de la segunda pasada ... 160
6.8.3.6 Resumen de la tercera pasada ... 161
6.8.3.7 Resumen de la cuarta pasada ... 162

7 CONCLUSIONES ... 163

8 BIBLIOGRAFÍA ... 166

Introducción

Existe una gran cantidad de tecnología de Software en el mercado mundial. Una buena parte de esa tecnología se puede obtener de forma gratuita, es de gran calidad y su variedad es orientada a satisfacer diferentes necesidades. Sin embargo, sería interesante conocer *¿Cuánta de esa tecnología de software es planeada, diseñada y/o desarrollada en México?*. Se sabe que, en México, hay poca inversión económica en desarrollo de tecnología propia. Al haber poca inversión, es de esperar que sea bajo ó prácticamente nulo el desarrollo de nuevas tecnologías de software y hardware.

En la actualidad, en México, gran parte de las actividades de desarrollo de sistemas e informática están dedicadas al soporte de las actividades empresariales que son, en su gran mayoría, labores administrativas y de control (nóminas, finanzas, contabilidad, inventarios, control y registro de personal, administración de clientes, etcétera). Esto hace del mercado de desarrollo de aplicaciones administrativas algo muy lucrativo que da la oportunidad de negocio para muchas empresas grandes y pequeñas. Sin embargo, se puede observar claramente que el desarrollo de los sistemas, hechos por un gran número de informáticos de nuestro país, se reduce a aplicar los conocimientos adquiridos de una o muchas herramientas de software elaboradas en otros países.

Planteada la primera motivación que da origen al tema del presente trabajo, se agrega la nacida por una vivencia previa. En el periodo de tiempo que estuve como estudiante universitario de la carrera de ingeniería en computación; compartí con varios de mis compañeros de escuela, quizá sin fundamento, algunos temores respecto a la complejidad que presentaba el estudio y comprensión de algunas áreas como la de programación, teoría de lenguajes y también compiladores.

El impulso al desarrollo tecnológico como primer tema de investigación resulta muy ambicioso y a la vez ambiguo. Me cuestioné muchas veces acerca de ¿Qué podría

desarrollar? que, en la medida de lo posible, pudiera incluir las motivaciones antes citadas. Al principio resultaba nebuloso encontrar un punto de convergencia. Sin embargo, sobre la marcha fui encontrando y acumulando material referente a la:

- Teoría de lenguajes.
- Teoría de compiladores.
- Paradigmas de programación.

Establecida la información que sería objeto de estudio, el siguiente paso consistió en determinar ¿el cómo? se aterrizaría en la práctica dicho conocimiento. La respuesta fue desarrollo de un compilador. El desarrollo de un lenguaje de programación involucra el conocimiento de la teoría de lenguajes, un compilador trabaja teniendo como entrada un texto escrito en un lenguaje de programación, y definitivamente para desarrollar un compilador habrá que programar.

El universo de aplicaciones que se pueden dar a los compiladores es amplio, existen en el mercado una gran cantidad de compiladores enfocados a obtener resultados igualmente variados.

Muchos especialistas del área informática pueden no necesariamente enfrentarse al reto de desarrollar un intérprete o un compilador; sin embargo en algún momento de su carrera, quizá tendrán que elegir alguna herramienta de desarrollo o lenguaje de programación, así que conocer las bases de su funcionamiento les sería de utilidad para tomar la mejor decisión. Así, en principio se podría suponer que el conocimiento, diseño y desarrollo de un compilador no debería ser tomado únicamente como una actividad de fines académicos.

Así pues, de forma quizá ambiciosa; pero, sin pretender tener conocimientos siquiera cercanos a los que poseen los grandes gurús de las áreas señaladas, los objetivos finales del presente material son:

- Definir y delimitar un nuevo lenguaje de programación con la restricción de que el empleo de palabras reservadas sea en español.

- Diseñar y construir un compilador para el lenguaje de programación definido previamente.

- Motivar a profesionistas y futuros profesionistas del área informática a desarrollar nueva tecnología.

Se puede prever que al abordar las temáticas ya señaladas nos introducimos en un extenso mar de información; así pues, se consideró conveniente y necesario colocar fronteras al material presentado, procurando mantener las herramientas necesarias para cubrir los objetivos finales. Después de varios intentos por agrupar y estructurar de la mejor forma el material incluido en el presente texto, se desarrollaron los siguientes capítulos: 1. Teoría de Lenguajes Formales y Autómatas, 2. Compiladores, 3. Orientación a Objetos, 4. Java, 5. Lenguaje Java en Español, y por último 6. Compilador en Español. Es claro que aunque prácticamente todos los capítulos, en forma directa o indirecta, hacen referencia a un compilador, hay algunos más relacionados con otros; razón por la que se determinó agrupar, para su estudio, esos capítulos en tres bloques o partes.

La primera parte del presente material se titula “Teoría de Lenguajes y Compiladores” y contiene dos capítulos: 1. Teoría de Lenguajes Formales y Autómatas, 2. Compiladores. A lo largo de la historia de los compiladores, se han descubierto técnicas sistemáticas para manejar muchas de las tareas importantes que hay en la compilación. De la mano de dichas técnicas, se han desarrollado robustos lenguajes de programación, entornos de desarrollo y herramientas de apoyo en la construcción de compiladores. Así pues, en el primer capítulo presentan tanto los conceptos involucrados en la definición de

un lenguaje como las técnicas sistemáticas que manejan algunas de las muchas tareas importantes que hay en el proceso de compilación. El segundo capítulo se centra en presentar la teoría referente a los "compiladores tradicionales", entendiendo por compilador tradicional el tipo de compilador comúnmente desarrollado y estudiado por los diferentes libros de texto del área. Sin embargo no está de más recordar que, dado que la teoría de lenguajes y de compiladores es muy extensa, se buscará capturar sólo el material necesario para colocar los cimientos del caso práctico, del que se hablará más adelante.

La primera parte, según se comentó, presenta los conceptos básicos referentes a la teoría de lenguajes formales y autómatas, las técnicas empleadas en el manejo de algunas tareas importantes de un compilador, se describe el objetivo de un compilador y la forma en que funciona. Cubierto éste material, quizá se podría iniciar el desarrollo de un nuevo lenguaje de programación. Sin embargo, en los objetivos no se describe la estructura ni las reglas que deberá seguir el nuevo lenguaje de programación (excepto que las palabras reservadas empleadas deberán estar en español), ni tampoco se describe cual será el producto del proceso de compilación realizado por el compilador a construir. Es necesario, entonces, decidir y definir la estructura que tendrá el código fuente de entrada y el de salida.

Un punto de inicio para determinar la estructura y/o funcionamiento que se quiere dar al código fuente que saldrá del nuevo lenguaje de programación es estudiando las diferentes formas de concebir la programación. En la actualidad, en el área de desarrollo de software (particularmente el empresarial) es común referirse a Lenguajes de Programación Orientados a Objetos, y ya que se tiene libertad de elegir un paradigma de programación se optó por la Orientación a Objetos. Así pues, la segunda parte, titulada "Orientación a Objetos y Java", incluye dos capítulos: 3. Orientación a Objetos, 4. Java. El capítulo 3 presenta la historia de las diferentes formas de concebir la programación[1], centrándose en presentar las bases de la Orientación a Objetos y por consiguiente de la Programación Orientada a Objetos. Aunque con la presentación del capítulo 3 ya se podría tener una idea

[1] También conocidos como paradigmas de programación

de la estructura que deberá tener el nuevo lenguaje y también sus respectivos límites, no se ha determinado cual sería la forma y características del producto de la compilación.

Cubierta la primera parte, se sabe que el código de salida del proceso de compilación es determinada por el diseñador del compilador. Al no haber una restricción en el producto obtenido del proceso de compilación del compilador a desarrollar, se tomó la decisión de realizar como salida un archivo con extensión "*java*" que a su vez pueda ser compilado por Java. Así que, el capítulo 4 se centra en el conocimiento del lenguaje de programación Java[2], el proceso de compilación del código fuente hecho en dicho lenguaje y la forma en que éste es ejecutado.

Tomando como base el material presentado en las dos primeras partes, es momento de centrar esfuerzos en un caso práctico. La tercera parte denominada "Caso práctico" se divide en dos capítulos. El capítulo 5 se dedica a la definición y delimitación del nuevo lenguaje de programación, mientras que el capítulo 6 se encarga de presentar el diseño que deberá seguir el compilador, que como ya se adelantó recibirá código fuente basado en español y a su salida proporcionará un archivo con extensión ".java" que podrá ser compilado por el compilador de Java. Con la finalidad de referirnos a los productos de ésta tercera parte, se denomina al nuevo lenguaje de programación: "*Lenguaje Java en Español*", mientras que el software recibirá por nombre: "*Compilador en Español*".

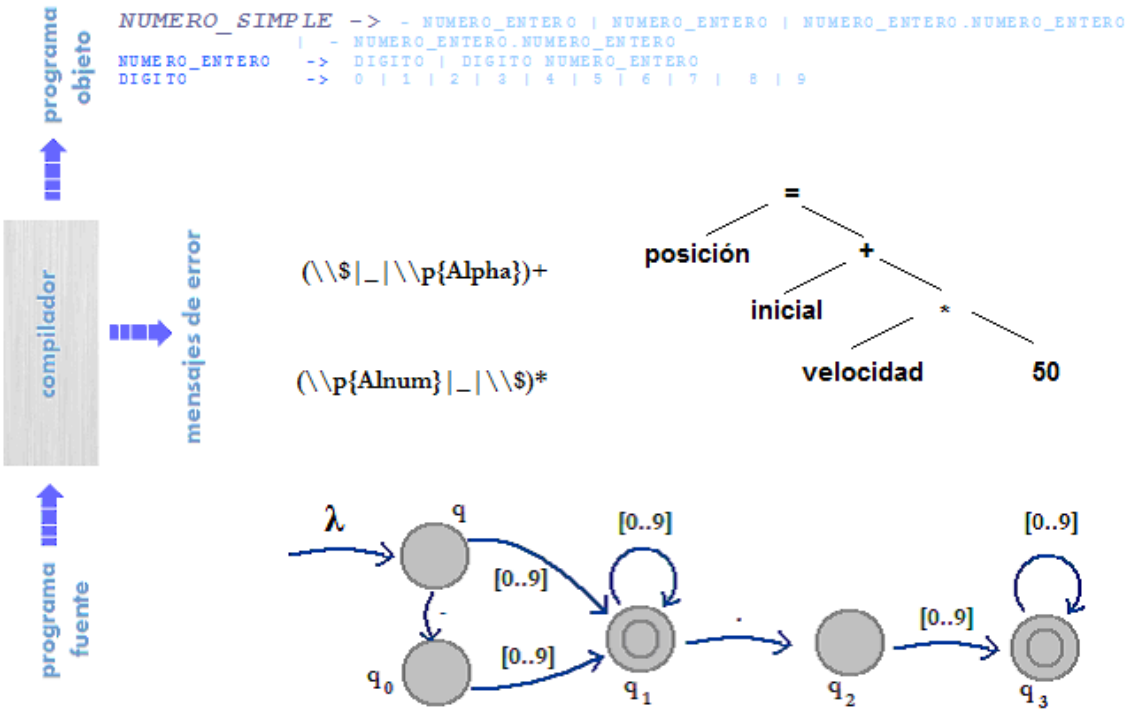
Como se comentó, una de las razones de ser, del presente documento, es intentar mostrar que en México también es posible desarrollar nuevas tecnologías de software, a partir de la teoría existente. Lejos de cuestionar el uso de tecnologías actuales y el seguimiento de los estándares mundiales actuales, se pretende animar a los lectores a presentar ideas y buscar nuevas soluciones tecnológicas no sólo en beneficio propio sino de la población de nuestro país en general y, por que no, del mundo.

---

[2] Se elige el lenguaje de programación Java entre otras cosas, por que es un lenguaje orientado a objetos, simple, robusto y con una gran aceptación en el mundo.

Por último, encontrará en un CD adjunto, el sistema desarrollado usando el *Lenguaje Java en Español* y el diseño del *Compilador en Español* vistos a lo largo de la tercera parte del presente texto.

Primera parte



Teoría de Lenguajes y Compiladores

1 Teoría de lenguajes formales y autómatas

Desde la aparición de los primeros compiladores a la fecha, se han descubierto técnicas sistemáticas para manejar muchas de las tareas importantes que hay en la compilación. De la mano de dichas técnicas, se han desarrollado robustos lenguajes de programación, entornos de desarrollo y herramientas de apoyo en la construcción de compiladores.

El presente capítulo, presentará de forma breve, la teoría de lenguajes formales y autómatas empleada en el diseño y construcción de un compilador.

Una línea de investigación que ha adquirido mucha importancia, durante los últimos años, es la aplicación de las ideas de la teoría de lenguajes, por ejemplo:

- Las expresiones regulares
- Los autómatas
- Las gramáticas libres de contexto

Son de interés en la presente sección las aplicaciones en el diseño de software como compiladores y procesadores de texto. En la actualidad es enorme el universo de compiladores, de herramientas que realizan algún tipo de análisis de texto bases de datos, de interfaces de desarrollo de software, etcétera.

Podrá observar, más adelante, que un compilador entre otras cosas se encarga de verificar que un programa[3] cumpla las reglas impuestas por el lenguaje en el cual se basó. A manera de preámbulo, antes de mencionar la teoría de compiladores serán abordados algunos puntos importantes dentro de la teoría de lenguajes formales.

[3] Un programa es un conjunto detallado explícito de instrucciones de computadora para realizar un trabajo. FUNDAMENTOS DE PROGRAMACIÓN. Peñaloza R. Ernesto, edita UNAM, tercera edición, 2001, pág. 454.

1.1 Conceptos

Es necesario tener conocimiento de los conceptos necesarios en la teoría de lenguajes formales. Los siguientes puntos describen algunos de ellos.

1.1.1 Símbolo

Se entenderá por símbolo, el trazo de una forma geométrica. Con esta idea, los siguientes son ejemplos de símbolos:

- Las letras
- Los números
- Los signos puntuación

1.1.2 Palabra

Una palabra o cadena es una secuencia finita de símbolos yuxtapuestos.

Así por ejemplo: Si ***a***, ***b*** y ***c*** son símbolos, cada una de las siguientes secuencias de símbolos pertenece al universo infinito de cadenas que se pueden formar con los símbolos citados.

aa	abc	bbbba	ababaabab
bbb	cba	aaccbb	ccccccccc
ccc	bcabc	cccccaa	cacacacac

Tabla 1-1 Cadenas muestra tomadas del universo posible, dados los símbolos a, b y c

Es necesario tener presente que las cadenas mostradas sólo representa un pequeño grupo del universo de posibilidades que se pueden obtener.

1.1.3 Longitud de una cadena

La longitud de una cadena ***s***, que se denota como | ***s*** |, es el número de símbolos que componen una cadena. La cadena vacía, frecuentemente denotada por el ***λ***, es la cadena que consiste en cero símbolos.

Así, por ejemplo: Dada la cadena ***efg***, su longitud será 3

1.1.4 Prefijo y sufijo de una cadena

Los prefijos de una cadena están formados por los primeros símbolos de ésta, mientras que los sufijos, por los últimos.

Así, por ejemplo: La cadena ***efg***, tiene como:

- Prefijos : ***λ***, ***e***, ***ef***, ***efg***
- Sufijos: ***λ***, ***g***, ***fg***, ***efg***

1.1.5 Concatenación de cadenas

La concatenación de dos cadenas es la nueva cadena que se forma al escribir la primera cadena seguida de la segunda sin que haya un espacio o algún otro símbolo entre ellas.

Así, dadas las cadenas:

1) ***efg***

2) ***hij***

La concatenación de 1y 2 es: ***efghij***

1.1.6 Alfabeto

Es un conjunto finito de símbolos.

Así pues, por ejemplo: En el idioma español encontramos las letras que van de la ***A*** a la ***Z*** y de la ***a*** a la ***z***.

1.1.7 Lenguaje

Un lenguaje, formal, es un conjunto de cadenas de símbolos tomados de algún alfabeto. También el conjunto vacío $\boldsymbol{\Phi}$, y el conjunto formado por la cadena { $\boldsymbol{\lambda}$ } son considerados lenguajes.

Las cadenas de un lenguaje obtenidos a partir de un alfabeto se puede mostrar de la siguiente forma: Supóngase el alfabeto {0,1} denotado por Σ, y a Σ* como lenguaje el conjunto de cadenas sobre dicho alfabeto. Se tendría la siguiente afirmación:

Si Σ = {0,1}, entonces Σ* = { $\boldsymbol{\lambda}$, 0,1,00,01,11,001,010,011,100,110,…}

1.2 Gramáticas de libre contexto

Para los lingüistas, "una gramática de libre contexto es un conjunto de variables cada una de las cuales representa un lenguaje. Los lenguajes representados por las variables se

describen de manera recursiva en términos de las mismas variables"[4]. Mientras los lingüistas estudiaban las gramáticas de libre contexto para describir los *lenguajes naturales*, los científicos que trabajaban en la computación comenzaron a describir *lenguajes de programación* mediante una notación conocida como forma de Backus – Naur (BNF), que es la notación para las ***gramáticas libres de contexto***[5] con cambios menores en su formato y algunas abreviaturas.

El uso de las gramáticas libres de contexto ha simplificado grandemente la definición de lenguajes de computación y la construcción de compiladores.

Una gramática de libre contexto tiene cuatro componentes:

1. Un conjunto de componentes léxicos[6], denominados símbolos terminales.

2. Un conjunto de no terminales.

3. Un conjunto de producciones, en la que cada producción consta de un no terminal, llamado lado izquierdo de la producción, una flecha y una secuencia de componentes léxicos y no terminales o ambos, llamado lado derecho de la producción.

4. La denominación de uno de esos no terminales como símbolo inicial.

A manera de ejemplo, observe la siguiente gramática:

S ⟶ AB
A ⟶ ***a*** | **a**A
B ⟶ **b** | **b**B

[4] INTRODUCCIÓN A LA TEORÍA DE AUTÓMATAS, LENGUAJES Y COMPUTACIÓN, Hopcroft John E., Ullman Jeffrey D., edit. CECSA, primera edición, 2000, pág. 84.
[5] A lo largo del texto será común hacer referencia a una gramática de libre contexto bajo las siglas: ***GLC***
[6] El apartado dedicado al Análisis léxico del siguiente capítulo, proporciona más detalles acerca de los componentes léxicos.

En este caso, las letras en minúscula representan símbolos terminales, mientras que las letras en mayúscula representan símbolos no terminales, en los cuales S es el símbolo inicial. El lenguaje que define la gramática anterior es:

Un conjunto infinito de cadenas, cada una escrita de la siguiente manera:
una secuencia de uno o más símbolos ***a*** *seguidos uno o más símbolos* ***b****.*

Así, la siguiente lista muestra algunos ejemplos de las cadenas válidas para el lenguaje formado:

ab	abb	aaaaaaaaaabbbbbbbbb
aab	abbb	aaaaaaaaaaaaaaaaaabb
aaab	abbbb	aabbbbbbbbbbbbbbbbbb

Tabla 1-2 Cadenas válidas para la gramática dada

Para validar que una cadena esté incluida en el conjunto de cadenas definidas por una gramática, la cadena a validar debe poder ser incluida en el no terminal considerado como inicial, este proceso la mayoría de las veces requiere de una serie de sustituciones. A continuación se muestra un ejemplo, considere la cadena ***ab***:

- La cadena de entrada será leída de izquierda a derecha[7].

- Verificar que la cadena de entrada corresponda a una producción. En el presente ejemplo no hay una producción que incluya al terminal **a** concatenado con el terminal **b**. Así que pasamos al siguiente punto.

- Leer el primer elemento. En éste caso se trata de un terminal, el proceso a seguir es buscar entre el total de producciones, alguna que incluya al terminal encontrado y

[7] Por simplicidad, en el ejemplo se consideraron símbolos unitarios, es decir no se incluyeron componentes léxicos de más de un símbolo.

sustituir al terminal. En este caso la producción encontrada es la del no terminal A. Luego de realizar la sustitución tendríamos como resultado la cadena de entrada A***b***.

- Verificar que la cadena de entrada corresponda a una producción. En el presente ejemplo no hay una producción que incluya al no terminal **A** concatenado con el terminal **b**. Así que pasamos al siguiente punto.

- Leer siguiente elemento, recordará que el primer elemento en la cadena de entrada original había sido leído. En éste caso se trata de un terminal, el proceso a seguir es buscar entre el total de producciones, alguna que incluya al terminal encontrado y sustituir al terminal. En este caso la producción encontrada es la del no terminal B. Luego de realizar la sustitución tendríamos como resultado la cadena de entrada AB.

- Verificar que la nueva cadena de entrada corresponda a una producción. Con AB como cadena de entrada, al buscar una coincidencia con el no terminal S. La producción encontrada coincide con la producción considerada como inicial, así que podemos afirmar que la cadena es válida.

1.3 Expresiones regulares

Las expresiones regulares son una secuencia de caracteres y símbolos que definen un conjunto de cadenas. Son útiles para validar una cadena de entrada y verificar que los datos recibidos estén en un formato específico.

Una expresión regular consiste de caracteres literales y símbolos especiales. El uso de expresiones regulares es soportado por algunos lenguajes de programación, y aunque en esencia funcionan de la misma forma, presentan algunas pequeñas variantes ligadas al lenguaje de programación de que se trate.

En seguida se presentan algunas de las simbologías soportadas por Java para expresiones regulares[8].

Simbología empleada	Validación realizada
[abc]	Los caracteres leídos sean: a, b, ó c
[^abc]	Los caracteres leídos no sean: a, b, ó c
\t	Se trate de carácter tabulador ('\u0009')
\n	Se trate de carácter de nueva línea ('\u000A')
\r	Se trate de carácter retorno de carro ('\u000D')
\d	Se trate de un dígito que va de 0-9
\D	No se trate de un dígito, [^0-9]
\s	Se trate de carácter \t, \n, \x0B, \f, \r
\S	No se trate de carácter \s
\w	Se trate de carácter alfanumérico o guión bajo
\W	No se trate de carácter \w
X*	La repetición del bloque X es cero o más veces
X+	La repetición del bloque X es una o más veces
X{n}	La repetición del bloque X es exactamente n veces

Tabla 1-3 Expresiones regulares soportadas por Java

En la sección anterior, el lenguaje definido por la gramática de libre contexto usada como ejemplo es:

> *Un conjunto infinito de cadenas, cada una escrita de la siguiente manera: una secuencia de uno o más símbolos **a** seguidos uno o más símbolos **b**.*

Así pues la expresión regular equivalente sería: ***a*** **+** ***b*** **+**

1.4 Autómatas

Es importante realizar una observación acerca de las gramáticas de libre contexto y las expresiones regulares. Las primeras permiten trabajar en un lenguaje natural, mientras que las segundas permiten acercarnos a la programación del mismo.

---

[8] Encontrará la tabla y descripción completa de las expresiones regulares soportadas por Java en la siguiente dirección electrónica**:** **http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html**.

Una forma adicional de expresar validaciones de cadenas es empleando diagramas de transiciones.

1.4.1 Diagramas de transiciones

Un diagrama de transiciones es un diagrama de flujo estilizado. En un compilador un diagrama de transiciones podría representar las acciones que tienen lugar cuando el analizador léxico es llamado por el analizador sintáctico en el proceso de validación. Hay dos elementos básicos en un diagrama de transición:

- **Estado**. Es una situación en la que se permanece un lapso de tiempo. En la vida real, un buen ejemplo[9] sería el estado civil de una persona, la cual permanece Soltera mientras no se case. De forma gráfica, los estados, son representados por círculos que contienen una etiqueta que describe el estado.

- **Transición**. Es un evento asociado a un estado, que provoca que se alcance un nuevo estado. Así; para el ejemplo referente al estado civil de una persona, el evento que podría hacer que cambiara el estado civil de una persona soltera, sería una boda, pasando entonces su estado civil a Casado. Gráficamente son representados por flechas, denominadas aristas, que apuntan al nuevo estado, contienen una etiqueta que señala bajo que condición se realiza el cambio de estado.

Un estado se etiqueta como el estado de inicio; es el estado inicial del diagrama de transición donde reside el control cuando se quiere validar una cadena o la estructura de una cadena. Ciertos estados pueden tener acciones que se ejecutan cuando el flujo de control alcanza dicho estado. Al entrar en un estado se lee el siguiente símbolo de entrada; si hay una arista del estado en curso de ejecución cuya etiqueta concuerde con ese símbolo de entrada, entonces se va al estado apuntado por la arista. Los diagramas de transición

[9] Ejemplo, basado en el libro electrónico: AUTÓMATAS Y LENGUAJES, Brena Ramón, Tec de Monterrey, 2003, pág. 27 encontrado en la dirección: http://homepages.mty.itesm.mx/rbrena/AyL.pdf.

pueden tener estados finales, ellos son señalados con un pequeño círculo dentro del estado que se desea señalar como final. Si se lee una cadena de entrada y después de leerla nos encontramos en un estado final, se dice que la cadena es válida.

1.4.1.1 Autómatas finitos

Un verificador de un lenguaje es un programa que toma como entrada una cadena **A** y responde "si" si **A** es una frase del programa y "no" si no lo es. Los autómatas, pueden ser vistos como una abstracción gráfica de procesos repetitivos.

Un autómata finito puede ser "determinista" o "no determinista", donde no determinista significa que en un estado se puede dar el caso de tener más de una transición para el mismo símbolo de entrada. Tanto los autómatas finitos deterministas como los no deterministas pueden reconocer con precisión a los conjuntos regulares. Por tanto ambos pueden reconocer con precisión lo que denotan las expresiones regulares. Esta es la razón por la que los autómatas también son importantes en el diseño de compiladores.

A continuación se presenta un ejemplo de autómata que valida que una cadena de entrada corresponda a los números enteros o a los reales tanto positivos y negativos.

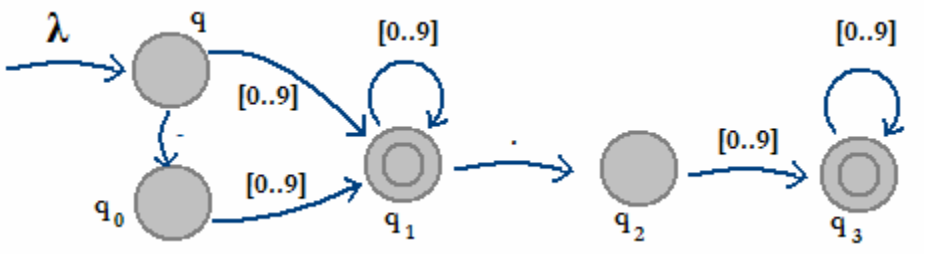


Figura 1-1 Autómata que valida números enteros y reales

En la práctica, el mayor inconveniente de usar autómatas en el diseño del proceso de verificación de un compilador, es que la cantidad de información gráfica escrita es tan grande que puede resultar confusa.

2 Compiladores

Existen en el mercado una cantidad enorme de compiladores enfocados a obtener resultados igualmente variados. Aunque en esencia todos ellos guardan cosas en común.

La historia dice que en 1946 se desarrolló el primer ordenador digital. En un principio, estas máquinas ejecutaban instrucciones consistentes en códigos numéricos que señalaban a los circuitos de la máquina los estados correspondientes a cada operación, lo que se denominó lenguaje máquina. Posteriormente surgieron los lenguajes ensambladores en los cuales los usuarios debían de escribir sus programas mediante claves más fáciles de recordar que los códigos binarios, al final, todas esas claves juntas se traducían manualmente a lenguaje máquina.

El presente capítulo se centra en presentar la teoría referente a Compiladores, tomando como base el "compilador tradicional", entendiendo por compilador tradicional el tipo de compilador comúnmente desarrollado y estudiado por los diferentes libros de texto del área. Sin embargo, dado que la teoría de compiladores es muy extensa, se buscará plasmar sólo el material necesario para colocar los cimientos del caso práctico presentado en la tercera parte del presente material.

2.1 Definición

La primera definición citada dice que un compilador es "*un programa que sirve para traducir un lenguaje de programa simbólico más alto a un lenguaje de máquina que sea comprensible para el procesador. La entrada a un compilador es un programa fuente. El compilador asigna partes de la memoria interna a los datos necesarios para el programa, lleva un registro temporal de nombres localizaciones y características de cada elemento-dato en una lista denominada tabla simbólica. Interpreta especificaciones de procedimiento del programa fuente y determina que instrucciones de la computadora son necesarias para realizar el trabajo. Después, el compilador genera las instrucciones de*

máquina, les asigna almacenamiento, proporciona al programa las direcciones correctas de datos y las constantes requeridas; produce así un programa ejecutable"[10]. El inconveniente que presenta ésta definición es que encierra un número considerable de detalles técnicos, que hay que conocer desde la misma definición de un compilador.

La siguiente definición: "*Los compiladores traducen los programas en lenguaje de alto nivel a programas en lenguaje de máquina*"[11], pudiera parecer que excluye, al igual que la definición anterior, a los compiladores que no entregan como resultado un lenguaje de máquina. Tenemos como ejemplo más claro a Java, del cual como producto del proceso de compilación se obtiene un archivo con extensión *.class* que tiene una estructura totalmente conocida y alejada de un lenguaje de máquina.

Una definición más general es la presentada en el texto *Compiladores. Principios, técnicas y herramientas de Alfred V. Aho y Jeffrey D. Ullman* que define un compilador como "*un programa que lee un programa escrito en un lenguaje, el lenguaje fuente, y lo traduce a un programa equivalente en otro lenguaje, el lenguaje objeto; y como parte importante de este proceso de traducción, el compilador informa a su usuario de la presencia de errores en el programa fuente*"[12]. La figura siguiente ilustra la presente definición.

Figura 2-1 Diagrama que ilustra el proceso de compilación

[10] FUNDAMENTOS DE PROGRAMACIÓN. Peñaloza R. Ernesto, edita UNAM, tercera edición, 2001, págs. 448, 454.
[11] JAVA, CÓMO PROGRAMAR. Deitel Harvey M., Deitel Paul J., quinta edición, 2004, pág. 22.
[12] COMPILADORES, PRINCIPIOS, TÉCNICAS Y HERRAMIENTAS. Aho Alfred V., Seti Revi, Ullman Jeffrey D., edit. Addison Wesley, primera edición, 1998, pág. 1.

Existe una gran cantidad y variedad de lenguajes de programación, desde FORTRAN, BASIC o pascal hasta los lenguajes actuales como Delphi, Java, los de la plataforma Microsoft .NET, Prolog, Python, etc[13]. Al igual que los lenguajes fuente, los lenguajes objeto son variados y pueden ser un lenguaje de máquina de una computadora o cualquier otro lenguaje de programación.

Aunque la definición anterior es bastante clara, e incluye a los compiladores actuales, presenta una ambigüedad. El texto siguiente: "*un programa que lee un programa…*" es un fragmento de dicha definición. La primera vez que aparece la palabra ***programa*** se refiere a la ejecución organizada de un conjunto de instrucciones y datos ubicados en memoria que se encargan de leer un archivo de texto conocido como ***fuente***. La segunda vez que aparece dicha palabra se refiere al conjunto de líneas capturadas en el ***código fuente***.

Una variante de la anterior definición de un compilador podría ser la siguiente: "*Un compilador es un sistema informático que lee un programa escrito en un lenguaje denominado **fuente**; y, devuelve un programa equivalente en otro lenguaje denominado **objeto**. Después de solicitar al sistema la compilación del **código fuente**, el usuario del compilador obtendrá un programa equivalente en otro lenguaje, si y sólo si, no se encuentran errores en el análisis del código fuente; en caso contrario se presentan, al usuario del compilador, los errores encontrados en el código fuente*". Aunque la definición es clara, se realizará una descripción más profunda de algunos conceptos introducidos en la anterior definición:

- Se entiende por error a la ruptura de alguna de las reglas léxicas, sintácticas o semánticas vigentes en el lenguaje sobre el que se ha desarrollado el programa fuente compilado. Aunque como se verá más adelante habrá algunas otras partes del proceso de compilación donde se podrían presentar y capturar dichos errores.

[13] Se hace la aclaración de que, no todos los lenguajes fuentes reciben un proceso de compilación ya que algunos de ellos son interpretados.

- Se entiende por sistema informático al Software (procedimientos, funciones, operaciones y rutinas que trabajan ordenada y armónicamente con la finalidad de cumplir un objetivo), que se encarga de verificar que el ***código fuente*** cumpla las reglas impuestas en la definición del lenguaje; y, como resultado entregue el ***código objeto*** o los ***errores*** que no permitieron la creación del código objeto.

- El ***código fuente*** se refiere al(a los) archivo(s) de texto escritos en algún lenguaje de programación en particular y que serán objeto del proceso de compilación.

2.2 Códigos objeto

La diversidad de ***códigos objeto*** que se pueden generar es muy grande. Algunos compiladores entregan un programa en lenguaje ensamblador como resultado del proceso de compilación, sí la meta es ir más allá y crear un programa objeto ejecutable, puede ser necesario apoyarse en otros programas además del compilador. Enseguida se listan algunos de ellos:

- El ***preprocesador*** es encargado del procesamiento de macros, la inclusión de archivos (como las cabeceras de C), enriquecer o extender los recursos de algún lenguaje. Además de ser el preprocesador el encargado de reunir el programa fuente en memoria, cuando éste fue dividido en módulos almacenados en diferentes archivos.

- Algunos compiladores producen código ensamblador que se pasa a un ***ensamblador*** para su procesamiento. Otros compiladores realizan el trabajo del ensamblador, produciendo código de máquina relocalizable que se puede pasar de forma directa al editor de carga y enlace.

- Editor de carga y enlace. Por lo general un programa llamado ***cargador*** realiza las dos funciones. El proceso de carga que consiste en tomar el código de máquina

relocalizable, modificar las direcciones relocalizables; además, se encarga de ubicar las instrucciones y los datos modificados en las posiciones apropiadas de memoria.

Como ya se comentó, el tipo de ***código objeto*** obtenido por un compilador puede ser muy amplio, tanto como los diferentes tipos de compiladores que se desarrollen. Así por ejemplo; el compilador Java obtiene, como resultado de un proceso de compilación en el que no se hayan detectado errores, uno o más archivos que puede ser interpretado por otro programa[14].

2.3 Análisis y Síntesis en la compilación

El proceso de compilación se divide regularmente en dos partes: ***análisis*** y ***síntesis***. La primera divide el ***código fuente*** en sus elementos componentes (identificadores, operadores, signos, etc.) y luego de verificarlo sintáctica y semánticamente, crea una representación intermedia del él. La segunda construye el ***código objeto***[15] deseado a partir de la representación intermedia. En la actualidad hay un gran número de herramientas, que sin ser compiladores, realizan primero algún tipo de análisis. A continuación se presentan algunos ejemplos:

- **Intérpretes**. Un intérprete realiza las operaciones que indica el *código fuente*. No produce un código objeto y sin embargo si realiza un análisis. Un ejemplo común de un intérprete se encuentra en BASIC.

- **Analizadores de texto**. También conocidos como editores de estructuras, hoy en día es común encontrar herramienta de desarrollo que señalan al usuario los errores encontrados en el texto que éste va redactando. Ejemplos de ellos son las interfaces de desarrollo de Visual Studio, Eclipse, Netbeans, etc.

[14] La máquina virtual de java de la cual se hablará más adelante.
[15] También conocido como programa objeto.

- **Verificadores estáticos**. Existen herramientas cuya finalidad no es obtener un *código objeto*; simplemente son creados con la finalidad de detectar si existen partes de un programa que nunca serán usadas, asignaciones erróneas de variables, etc.

2.4 Fases de un compilador

"*Conceptualmente un compilador opera en fases, cada una de las cuales transforma el código fuente de una representación en otra*"[16]. Como se muestra en la figura 2-2, la descomposición típica de un compilador muestra 6 fases.

- Analizador léxico
- Analizador sintáctico
- Analizador semántico
- Generador de código intermedio
- Optimizador[17] de código
- Generador de código

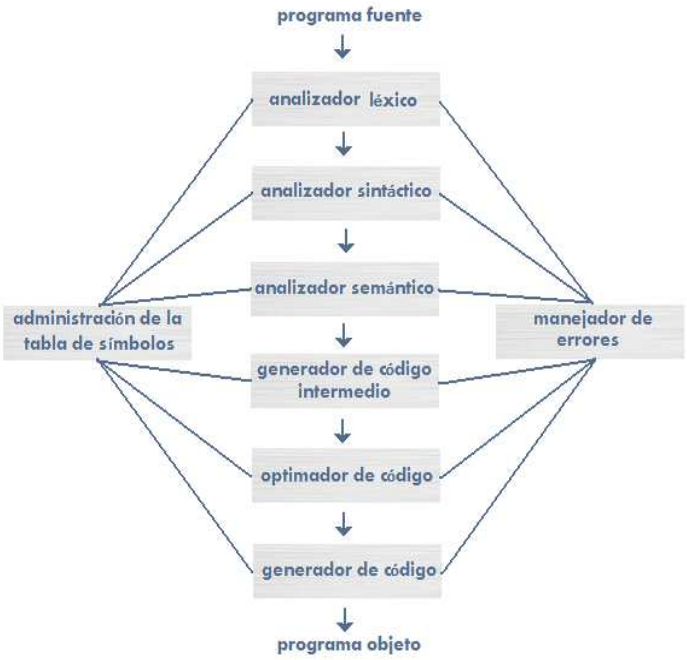


Figura 2-2 Fases en el proceso de compilación

En la práctica resulta común que más de una de las fases se realice al mismo tiempo, por lo cual se realizan agrupaciones de fases y no es necesario construir explícitamente las representaciones intermedias entre las fases.

---

[16] COMPILADORES, PRINCIPIOS, TÉCNICAS Y HERRAMIENTAS. Aho Alfred V., Seti Revi, Ullman Jeffrey D., edit. Addison Wesley, primera edición, 1998, págs. 4-14.

[17] Esta palabra no existe en el diccionario de la Real Academia de la Lengua Española y sólo refleja la traducción aproximada del idioma inglés.

La parte del análisis descrito en la sección anterior lo integran las fases de análisis léxico, sintáctico y semántico; y algunos compiladores incluyen la generación de código intermedio de forma explícita. La parte de síntesis está compuesta por las fases restantes. La administración de la tabla de símbolos y el manejador de errores tienen interacción con las 6 fases descritas.

2.4.1 Análisis léxico

En el análisis léxico la cadena de caracteres, que constituye el código fuente, es leída de izquierda a derecha y de arriba abajo formando grupos de *componentes léxicos*[18]. Donde cada componente léxico es una secuencia de caracteres que tienen un significado. Regularmente en esta fase:

- Se eliminan comentarios
- Se cargan librerías
- Se ejecutan macros
- Se agrupan caracteres en componentes léxicos
- Se crea y llena una tabla de símbolos

El análisis léxico podría generarse como parte del análisis sintáctico; sin embargo, pudiera ser más complicado realizar un analizador sintáctico que incluyera las convenciones de los comentarios, espacios en blanco, etc. En consecuencia, resulta recomendable que se trabaje por separado, ganando con ello un analizador léxico independiente que funcione como un componente especializado, potencialmente más eficiente y reutilizable. La figura 2-3

[18] También conocidos como tokens o lexemas.

ilustra el flujo seguido entre las fases de análisis léxico y sintáctico.

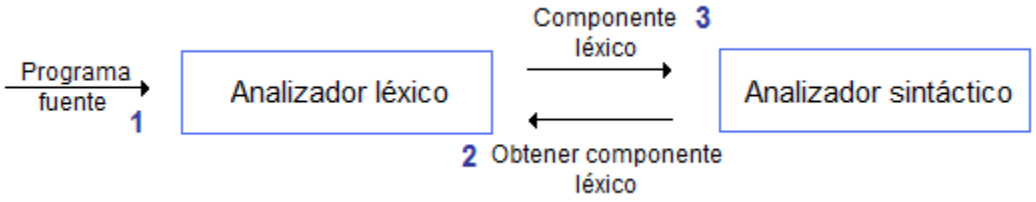


Figura 2-3 Flujo de un programa fuente en el análisis léxico y sintáctico

Se puede observar que el analizador léxico recibe como entrada el programa fuente y se encarga de procesarlo en la forma descrita anteriormente. Luego de procesar el programa fuente, el analizador léxico queda a la espera de que alguien, el analizador sintáctico en este caso, le solicite un componente léxico. El analizador sintáctico solicita un componente léxico al analizador léxico y recibe de éste un solo componente léxico; este proceso es cíclico mientras haya aún componentes léxicos.

2.4.1.1 Especificación de los componentes léxicos

Durante la fase de análisis léxico, entre otras cosas se realiza la agrupación de símbolos que componen el *código fuente* en componentes léxicos. Para poder realizar dicha separación, es necesario determinar cuales serán las reglas aplicadas con las cuales se construirán los componentes.

La figura 2-4 muestra un fragmento de código hecho en lenguaje Java. Sin importar por el momento la finalidad del código presentado, se puede ver claramente que, quizá, la forma mas simple en la que podría ser dividido el programa en sus componentes léxicos es:

```
private String obtenPaqueteDeClase(CodigoFuente cf){
      String strResp = null;
      if( cf.getLineasCodigo().size()> 0 ){
            StringBuffer sb = new StringBuffer();
            String strAux = null;
            try{
                  TokenVirtual  tv = cf.obtenTokenVirtual();
                  strAux = tv.getLexema();
            }catch(Exception e){}
            try{
                  while(! strAux.equals(";")){
                        sb.insert(sb.length(), strAux );
                        TokenVirtual  tvAux = cf.obtenTokenVirtual();
                        strAux = tvAux.getLexema();
                  }
            }catch(Exception e){}
            strResp = sb.toString();
      }else  strResp = "";
      return strResp;
}
```

Figura 2-4 Fragmento de programa en lenguaje Java

1. Por los espacios en blanco

2. Por los saltos de línea encontrados.

Como ejemplo, considere ahora la siguiente línea de código obtenida del fragmento de programa anterior:

```
if( cf.getLineasCodigo().size()> 0 ){
```

Dividiendo el código en componentes léxicos según los espacios en blanco encontrados, tenemos:

if(
cf.getLineasCodigo().size()>
0
){

Tabla 2-1 Lexemas obtenidos empleando espacios en blanco

No se debe perder de vista que un componente léxico debe ser significativo. El primer candidato a componente léxico es *if(*, ¿Puede entonces *if(* ser considerado un componente léxico?. Si tomáramos como referencia el idioma con el cual está hecho el programa, tendríamos que dado que el idioma ingles sobre el que está hecho el programa no tiene una palabra *if(* se podría determinar que la palabra estudiada no se trata de un componente léxico. Casi cualquier programador podría fácilmente reconocer la línea original en estudio, como una instrucción **if** empleada en el control de flujo de un programa. De hecho la mayor parte de lenguajes de programación tienen un mecanismo de control de flujo que inicia con la palabra **if** seguida de una expresión que puede ser evaluada como falsa ó verdadera, y dependiendo de la definición del lenguaje de programación, dicha expresión debe encontrarse o no encerrada entre paréntesis. Así pues, es la definición de la estructura del lenguaje de programación la que permitirá determinar como construir los componentes léxicos. Tenemos entonces que para dividir el código en elementos representativos, además de los ya presentados, necesitamos tener más elementos

que puedan fungir como discriminantes. Pudiera pensarse en usarse caracteres alfanuméricos, símbolos de puntuación, operadores lógicos, operadores numéricos, etcétera como mecanismos de generación de componentes léxicos. Aplicando dichos discriminantes, ahora tendríamos el siguiente grupo de componentes léxicos:

Componente léxico	Tipo encontrado
If	Cadena
(	Símbolo
Cf	Cadena
.	Símbolo
getLineasCodigo	Cadena
(	Símbolo
)	Símbolo
.	Símbolo
Size	Cadena
(	Símbolo
)	Símbolo
>	Símbolo
0	Número
)	Símbolo
{	Símbolo

Tabla 2-2 Nueva lista de lexemas de: if(cf.getLineasCodigo().size()> 0){

Retomando el ejemplo anterior, observe los lexemas señalados como de tipo Cadena. Puede ver claramente que aunque son diferentes, podría encontrar una expresión regular que le ayudará a describir y encontrar todos los tipos de elementos Cadena, no sólo eso, en caso de necesitar encontrar tipos de elementos más complejos, bastaría con diseñar e implementar la expresión regular adecuada en el analizador léxico.

2.4.2 Análisis sintáctico

"Todo lenguaje de programación tiene reglas que prescriben la estructura sintáctica de programas bien formados"[19].

[19] COMPILADORES, PRINCIPIOS, TÉCNICAS Y HERRAMIENTAS. Aho Alfred V., Seti Revi, Ullman Jeffrey D., edit. Addison Wesley, primera edición, 1998, pág. 163.

En apartados anteriores, fueron presentados los autómatas, las expresiones regulares y las gramáticas de libre contexto (en particular la forma Backus – Naur). Es común que para especificar la sintaxis de un lenguaje de programación se usen frecuentemente "Gramáticas de libre contexto"[20] ya que ofrecen algunas ventajas como las que a continuación se mencionan:

- Proporcionan una especificación sintáctica precisa y fácil de entender.

- Se puede construir un buen analizador sintáctico que determine si un programa está sintácticamente bien formado, a partir de algunas gramáticas.

- El punto anterior, permite la verificación de la estructura adecuada del lenguaje y la detección de errores en el mismo.

- Caso de ser necesario, se puede extender la funcionalidad de una gramática e implementar su construcción.

Para realizar su trabajo, el analizador sintáctico depende del analizador léxico. El analizador léxico le proporciona al analizador sintáctico los componentes léxicos que componen el programa fuente, en forma secuencial.

El analizador sintáctico se encarga de agrupar los componentes léxicos en frases gramaticales e informará de cualquier problema encontrado. Típicamente esta es una de las fases en las que se detectan más errores, así que su construcción debe ser lo suficientemente robusta para recuperarse de los errores más frecuentes y buscar con ello seguir procesando las otras fases a fin de proporcionar la mayor cantidad de errores.

[20] También citadas en algunos textos como "gramáticas".

2.4.2.1 Árboles de análisis sintáctico

El analizador sintáctico se encarga de agrupar los componentes léxicos en frases gramaticales.

Generalmente, las frases gramaticales del programa fuente se representan mediante un árbol de análisis sintáctico como el que se muestra a continuación:

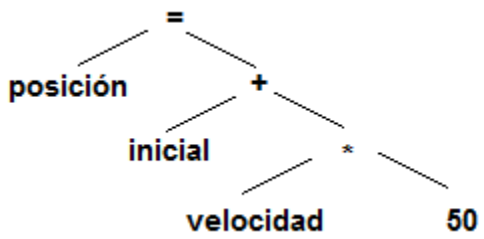


Figura 2-5 Ejemplo de árbol de análisis sintáctico

Una nueva definición del lenguaje generado por una gramática de libre contexto es, el conjunto de cadenas que pueden ser generadas por un árbol de análisis sintáctico. El proceso de búsqueda de un árbol sintáctico para una cadena de componentes léxicos se denomina análisis sintáctico de esa cadena. Una gramática de contexto describe de forma natural la estructura jerárquica de muchas construcciones de los lenguajes de programación, por ejemplo, la siguiente gramática de libre contexto podría usarse para verificar números reales y/o enteros:

```
NUMERO_SIMPLE -> - NUMERO_ENTERO | NUMERO_ENTERO | NUMERO_ENTERO .
                 NUMERO_ENTERO |  - NUMERO_ENTERO.NUMERO_ENTERO
NUMERO_ENTERO  ->  DIGITO | DIGITO NUMERO_ENTERO
DIGITO         ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |  8 | 9
```

Figura 2-6 Gramática de libre contexto que verifica números reales y enteros

Formalmente, dada una gramática independiente del contexto, un árbol de análisis sintáctico es un árbol con las siguientes propiedades[21]:

[21] COMPILADORES, PRINCIPIOS, TÉCNICAS Y HERRAMIENTAS. Aho Alfred V., Seti Revi, Ullman Jeffrey D., edit. Addison Wesley, primera edición, 1998, pág. 29.

1. La raíz está etiquetada con el *símbolo inicial*

2. Cada hoja está etiquetada con un componente léxico o con λ

3. Cada nodo interior está etiquetado con un *no terminal*

4. Si A es el *no terminal* que etiqueta a algún nodo interior y $X_1, X_2, \ldots X_n$ son las etiquetas de los hijos de ese nodo, de izquierda a derecha, entonces:

   A ⟶ $X_1, X_2, \ldots X_n$ es una producción. Donde $X_1, X_2, \ldots X_n$ representa un símbolo que es *un terminal* o un *no terminal*.

Observe el siguiente ejemplo. Dada la cadena: ***- 46 . 7***, la siguiente figura ilustra el árbol generado a partir de la gramática de libre contexto anterior:

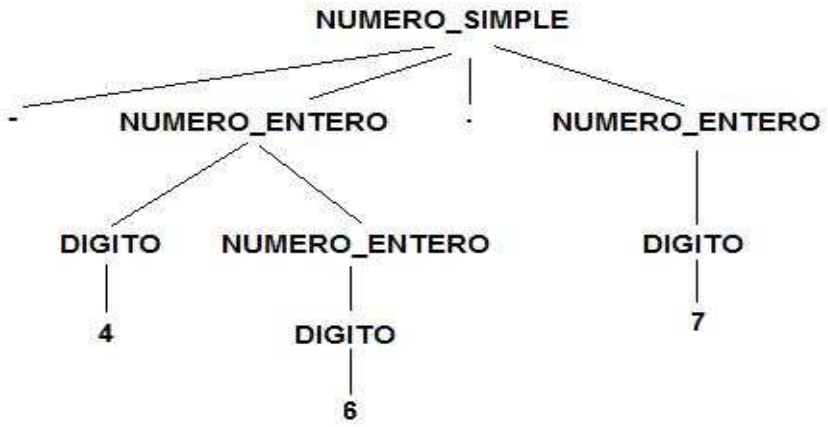


Figura 2-7 Árbol generado a partir de la cadena -46.7 y de la gramática de libre contexto que verifica números reales o enteros

Se podría pensar que la cadena presentada podría ser leída directamente como un componente léxico. Esto quizá sea correcto; sin embargo, adelantándonos un poco al diseño del compilador a realizar, implicaría realizar un analizador léxico más robusto, así que se decidió que fuera durante el análisis sintáctico donde se agrupara los componentes léxicos para formar números enteros y/o reales considerando su signo.

Por supuesto que las frases gramaticales analizadas en la fase de análisis sintáctico son mucho más complejas que la presentada, es más se podría afirmar que la gramática de libre contexto presentada es apenas una pequeña parte en una gramática de libre contexto de una sentencia de asignación o de paso de parámetros en alguna llamada a función.

Es importante tener presentes los siguientes puntos respecto de los árboles de análisis sintáctico, si se desea tener los resultados esperados:

- **Ambigüedad.** Se debe tener cuidado al considerar la estructura de una cadena según una gramática. Una gramática puede tener más de un árbol de análisis sintáctico que genere una cadena de componentes léxicos. Para demostrar que una cadena es ambigua, lo único que tiene que hacer es encontrar una cadena de componentes léxicos que tenga más de un árbol de análisis sintáctico.

- **Recursividad por la izquierda.** Es importante que cuando se tenga una gramática de libre contexto, se evite tener el caso en el cual se presenta una producción expresada en función de ella misma. La siguiente gramática ilustra el caso:

  S → **SA**
  A → abcd

  Observe que se cae en un ciclo infinito.

- **Asociatividad de operadores.** Al procesar algunas operaciones aritméticas, se puede presentar el caso en el que uno de los operandos tenga un operador aritmético a la izquierda y otro a la derecha. Se necesitan convenciones para decidir que operador debe considerar dicho operando. En general se *asocia a la izquierda*; así, un operador que tenga signo a ambos lados es tomado por el operador que esté a su izquierda. Así por ejemplo: 4 – 6 + 7 es equivalente a (4 – 6) + 7.

- **Precedencia de operadores.** La asociatividad aritmética no resuelve algunos problemas presentados al realizar operaciones aritméticas. Considerando sólo la asociatividad por la izquierda, para la expresión 5 + 70 / 10, hay dos posibles interpretaciones: (5+70) / 10 y 5 + (70 / 10), la segunda interpretación es la que aprendimos desde niños. Es decir, se necesita conocer la precedencia relativa entre operadores cuando hay presente más de un operador.

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

2.4.2.2 Análisis sintáctico descendente

Se puede considerar el análisis sintáctico descendente como un intento de encontrar una derivación por la izquierda para una cadena de entrada ó como un intento de construir un árbol de análisis sintáctico para la entrada comenzando por la raíz y creando los nodos del árbol en orden previo.

2.4.2.2.1 Análisis sintáctico por descenso recursivo

El *análisis sintáctico descendente recursivo* es un método descendente en el que se ejecuta un conjunto de procedimientos recursivos para procesar la entrada. A cada *no terminal* de una gramática se asocia un procedimiento. El análisis sintáctico por descenso recursivo puede incluir retrocesos; es decir, varios exámenes de la entrada. A continuación se presenta un ejemplo para ilustrar el proceso.

Dada la gramática:

S → ef**A**h

A → di | d

y la cadena de entrada *w* = efdh

Para construir el árbol de análisis sintáctico descendente primero se crea un árbol formado sólo por el nodo etiquetado con **S**. Sobre la cadena, un apuntador señala el primer elemento de la cadena *w*. Después se utiliza la producción de **S** para expandir el árbol como muestra la imagen (a) de la figura 2-8.

Dado que es posible emparejar el elemento al que se apunta en la cadena con el de la hoja situada más a la izquierda, movemos el apuntador de la cadena de entrada una posición y nos movemos a la siguiente hoja a la derecha. Puede verse en la misma figura que también es posible emparejar la entrada con la hoja. Así pues, al realizar nuevamente

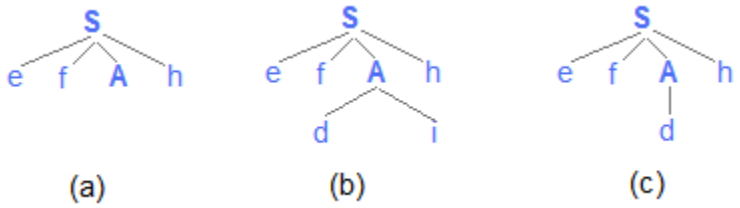


Figura 2-8 Análisis sintáctico por descenso recursivo

el proceso, se observa en el caso del árbol, que se encuentra un nodo que es un *no terminal* y al expandir la primera posibilidad de dicho no terminal se tiene el árbol mostrado en la imagen (b) de la figura. De leer el tercer elemento de la cadena de entrada, que es al que se apunta en ese momento, se puede emparejar el elemento con la hoja situada más a la izquierda de la producción **A**. Se mueve el apuntador una posición y también se desplaza a la siguiente hoja; sin embargo, se puede observar que ahora no es posible emparejar la hoja con el elemento de la entrada al que se está apuntando, pues se tiene **i** en el primer caso y **h** en el segundo. Un error es lanzado pues no se ha cumplido la primera posibilidad de la producción **A**, así que ahora se toma la segunda posibilidad de la producción **A** generando el árbol mostrado en la imagen (c) de la figura y también se regresa el apuntador a la posición en la que comienza la lectura de **A**. Puede ver que es posible emparejar los elementos leídos. Se avanza nuevamente una posición en la cadena de entrada y se desplaza

una hoja en el árbol. Puede ver que terminó la lectura de la producción **A** y se sigue ahora con una hoja perteneciente a la producción **S**.

Luego de todo el proceso anterior, es posible emparejar la entrada con la hoja y como es el último elemento de la producción podemos afirmar que la cadena de entrada es válida.

2.4.2.2.2 Analizador sintáctico predictivo

Existe una forma especial de análisis descendente recursivo, denominado predictivo, en éste se emplea un componente léxico para determinar sin ambigüedad el procedimiento seleccionado para cada no terminal. Observe la siguiente gramática:

prop ⟶ **mientras** exp **haz** prop | **si** exp **entonces** prop **otro** prop

Hay dos opciones o caminos a seguir en la producción **prop**. Sin embargo, las palabras clave: ***mientras*** y ***si*** indican que alternativa es la única con posibilidad de éxito para encontrar una proposición. De esa forma, en muchas oportunidades en la fase de análisis sintáctico, resultará innecesario el retroceso mostrado en el apartado anterior.

2.4.2.3 Análisis sintáctico ascendente

El análisis sintáctico ascendente, es conocido también como análisis sintáctico por desplazamiento y reducción. El análisis sintáctico por desplazamiento y reducción intenta construir un árbol de análisis sintáctico para una cadena de entrada que comienza por las hojas y avanza hacia la raíz.

Técnicamente, el proceso consiste en llevar una cadena ***w*** al símbolo inicial de una producción tomada como inicio, realizando sustituciones de subcadenas que son a su vez otras producciones de la misma gramática.

Considere la siguiente gramática de libre contexto a manera de ejemplo:

S → f**AB**g
A → **A**bc | b
B → d
y la cadena de entrada *w* = fbbcdg

Tomando como base la cadena de entrada, leyendo de izquierda a derecha la primera subcadena encontrada es f y a ella le sigue la subcadena **b**, se puede ver que **b** puede ser sustituida por **A**. Luego de realizar dicha sustitución, la cadena de entrada se convierte ahora en *f**A**bcdg*. Realizando el mismo proceso con la nueva cadena de entrada tenemos ahora se puede sustituir la subcadena **A**bc que corresponde a la producción **A**, realizando la sustitución correspondiente tenemos como resultado la cadena de entrada *f**A**dg*. La nueva cadena de entrada puede ser sustituida por *f**AB**g* que a su vez es la producción **S**, con lo cual se valida la cadena de entrada original.

2.4.3 Análisis semántico

Un compilador debe comprobar si el programa fuente sigue tanto las reglas sintácticas como las reglas semánticas impuestas al lenguaje fuente. Ésta comprobación, también denominada estática garantiza la detección y reporte de algunas clases de errores de programación en el código fuente. Entre las comprobaciones realizadas se encuentran:

1. **Comprobaciones de tipos.** El compilador debe informar si en el código fuente se han aplicado operadores incompatibles, por ejemplo si se suma un objeto con un número entero.

2. **Comprobaciones relacionadas con nombres.** En algunos casos un nombre debe aparecer al principio y al final del código, así que se debe verificar que esto ocurra.

3. **Comprobaciones de unicidad.** Hay situaciones en las que se debe definir un objeto exactamente una vez, por ejemplo la declaración de una variable.

4. **Comprobación de flujo de control.** Las proposiciones que hacen que se abandone el flujo actual, deben tener algún lugar a donde transferir el control. Por ejemplo la palabra ***return*** de Java que finaliza la ejecución del método en turno y devuelve el control al lugar donde fue llamado el método.

En esta fase se revisa el código fuente para encontrar errores semánticos y se reúne la información de los tipos para la fase posterior de generación de código. Una actividad importante realizada en esta fase es la verificación de tipos.

El diseño de un comprobador de tipos para un lenguaje se basa en información acerca de las construcciones sintácticas del lenguaje, la noción de tipos y las reglas para asignar tipos a las construcciones del lenguaje. A continuación se muestran algunos ejemplos de información con la que debería contar el diseñador de un compilador:

- Si los dos operandos de los operadores aritméticos de suma, resta, multiplicación y división son de tipo entero, entonces el resultado es de tipo entero.

- Si los operandos de los operadores aritméticos de suma, resta, multiplicación y división son de diferente tipo, entonces el resultado es de tipo de jerarquía más alta. Así, si uno es entero y otro es doble, el resultado será doble.

- Si una función es directamente asignada a un identificador, se debe verificar que el tipo devuelto por la función sea del tipo del identificador a modificar.

- Controlar las equivalencias de tipos, por ejemplo durante el uso de herencia.

2.4.4 Generación de código intermedio

Algunos compiladores generan una representación intermedia explícita del programa fuente. La representación intermedia tiene dos propiedades importantes: Debe ser fácil de producir y fácil de traducir al programa objeto.

Aunque un programa fuente puede ser traducido directamente al lenguaje objeto, una ventaja de utilizar una forma intermedia es que se puede aplicar un optimizador de código.

Los árboles sintácticos, la notación postfija y el código de tres direcciones son tres clases de representaciones intermedias[22].

La generación de código intermedio se puede intercalar en el análisis sintáctico, o puede realizarse después del análisis sintáctico y semántico. Generalmente se traducen a una forma intermedia construcciones de lenguajes de programación como declaraciones, asignaciones y proposiciones de flujo de control.

En el apartado del análisis sintáctico se expuso la importancia de tener presente la asociatividad, la precedencia de operadores y la eliminación de ambigüedades en los árboles de análisis sintáctico. Dada la gramática:

NUMERO → NUMERO + DIGITO | NUMERO – DIGITO |
NUMERO * DIGITO | NUMERO / DIGITO

DIGITO → 0 | 1 | … | 9

Suponga la cadena *w* = (9 – 4 + 5) * 7

[22] COMPILADORES, PRINCIPIOS, TÉCNICAS Y HERRAMIENTAS. Aho Alfred V., Seti Revi, Ullman Jeffrey D., primera edición, 1998, págs. 295-304, 480, 487.

Para determinar si la expresión denotada por la cadena *w* es válida empleando un árbol de análisis sintáctico, bastará obtener en la raíz el símbolo inicial de la gramática. La siguiente figura muestra el árbol obtenido.

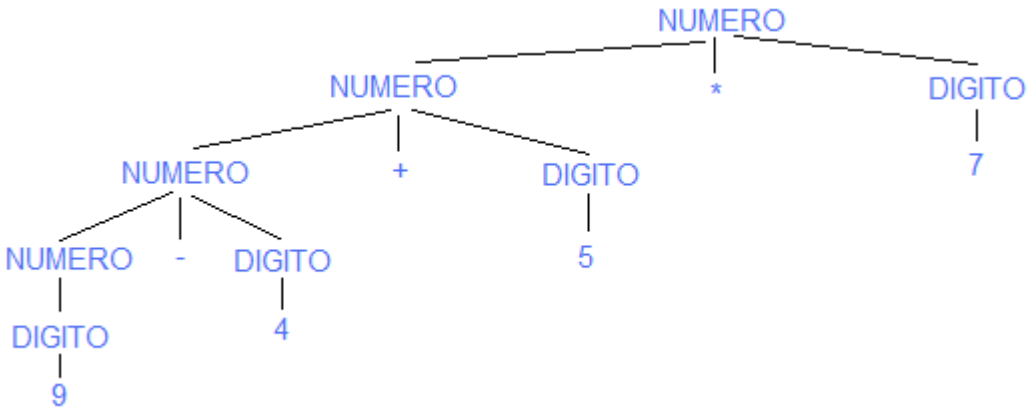


Figura 2-9 Árbol de análisis sintáctico para (9 - 4 + 5) * 7 para la gramática de libre contexto empleada

La lectura sería como sigue: Primero se evalúa lo que está dentro del paréntesis, así pues tenemos que 9 es un DIGITO mientras que este último también es un NUMERO, el siguiente símbolo leído es -, le sigue 4 que es un DIGITO. NUMERO – DIGITO es una producción equivalente a NUMERO. El siguiente símbolo leído es +, le sigue 5 que es un dígito por lo que ahora tenemos la producción NUMERO + DIGITO que es equivalente a NUMERO con lo que se ha terminado de evaluar la expresión entre paréntesis. El siguiente carácter leído es *, le sigue 7 que es un DIGITO y por último tenemos la producción NUMERO * DIGITO que es equivalente a NUMERO. De forma inmediata se puede afirmar que la expresión aritmética es correcta y que por lo tanto también lo es la cadena de entrada.

El lenguaje ensamblador tiene un número limitado de instrucciones. Resulta imposible evaluar expresiones como las mostradas en la figura 2-10. Entonces, la pregunta

```
2 * 3  + 4  - 6

6 + 6 – 6

Etc.
```

Figura 2-10 Instrucciones imposibles de evaluar mediante lenguaje ensamblador

obligada sería: ¿De que forma es posible presentar un código que pueda ser evaluado por el lenguaje ensamblador? La solución son los árboles binarios.

2.4.4.1 Árboles binarios

El siguiente es un ejemplo de árbol binario:

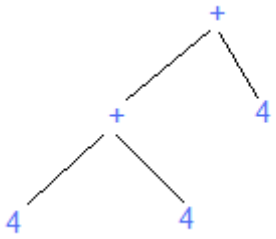


Figura 2-11 Ejemplo de árbol binario

Se puede observar que las hojas son los datos a operar, mientras que los nodos y la raíz en el árbol son las operaciones.

En un proceso de compilación es importante tener siempre en cuenta las reglas de precedencia en las expresiones que reflejan operaciones aritméticas con el fin de obtener el resultado esperado. La jerarquía de operadores empleada en aritmética es incorporada por muchos compiladores con la finalidad de que al compilar un *código fuente*, que incluya expresiones aritméticas, se entregue el resultado correcto. Así por ejemplo una operación de multiplicación o división tiene una jerarquía mayor que las operaciones de suma y resta, por lo que de encontrar una operación de multiplicación o división en una expresión donde el resultado es afectado también por una suma o resta, primero se realizará la multiplicación o división y después la suma o resta, según de lo que se trate. Dado que la división y multiplicación tienen la misma jerarquía, se realizará primero la operación que aparezca de izquierda a derecha, mismo caso se presenta con la suma y la resta que tienen la misma jerarquía. Existen elementos de mayor jerarquía que la multiplicación y la división como son los signos de agrupación, potenciación y radicación.

Es sumamente importante respetar las reglas aritméticas señaladas anteriormente a fin de obtener los resultados deseados. Suponga la expresión 2 + 3 * 5 y la expresión (2 + 3) * 5, la figura 2-12 muestra los que podrían ser los árbol generados.

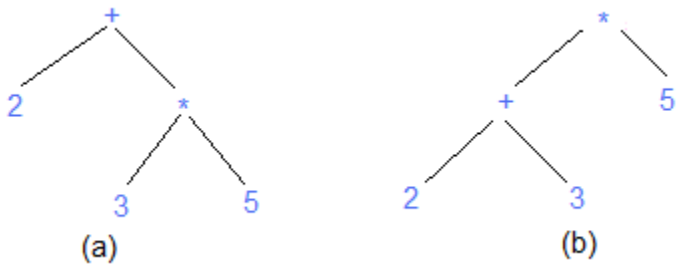


Figura 2-12 Árboles binarios obtenidos a partir 2 + 3 * 5 y (2 + 3) * 5 respectivamente.

La figura (*a***)** muestra el árbol generado por la expresión 2 + 3 * 5 donde el operador de mayor jerarquía es el de multiplicación, razón por la cual primero se realiza la multiplicación de los números 3 y 5 para luego a ese resultado sumarle 2. Por otro lado la figura (*b***)** utiliza paréntesis, que como se dijo tiene una jerarquía mayor que la multiplicación, razón por la cual primero se evalúa el contenido existente dentro de los paréntesis. Los resultados de las 2 expresiones serían: 17 y 25 respectivamente.

2.4.4.1.1 Recorrido de un árbol

El recorrido de un árbol se puede realizar de alguna de las siguientes formas:

- **Prefijo ó preorden.** El recorrido del árbol se realiza visitando primero la raíz, luego el hijo (u hoja) izquierdo y por último el hijo derecho.

- **Inorden.** Se realiza visitando el hijo izquierdo inorden, raíz inorden y por último el hijo derecho inorden.

- **Postfijo ó postorden.** El recorrido se realiza a partir del hijo izquierdo postfijo, hijo derecho postfijo y por último la raíz postfija.

Para aclarar el resultado en cada caso, suponga la cadena *w* = (2 * 5) + (5 * 4), cuyo árbol se muestra a continuación:

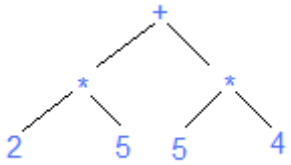


Figura 2-13 Árbol binario para la cadena (2 * 5) + (5 * 4)

La figura 2-14 (a) muestra el recorrido realizado en preorden, el cual da como resultado: +*25*54. La figura (b) muestra el recorrido inorden que da como resultado: 2*5 + 2*4. Por último la figura (c) que representa el recorrido postfijo da como resultado: 25*54*+.

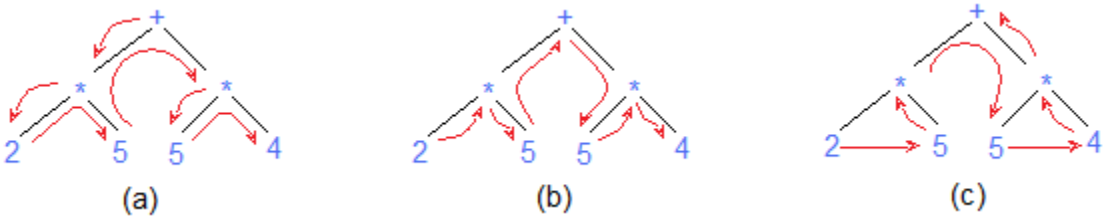


Figura 2-14 Recorridos sobre un árbol binario

En el proceso de evaluación numérica, el recorrido empleado es el postfijo, ya que permite que con una simple pila se puedan obtener operaciones intermedias más simples. Dada la expresión en postorden: 25*54*+, la lectura secuencial de dicha expresión tiene el siguiente proceso de evaluación:

1. Se introduce en la pila el primer elemento leído, 2 en éste caso.

2. Se introduce en la pila el siguiente elemento leído, 5 en éste caso.

3. El siguiente elemento leído es un operador de multiplicación, así pues se sacan de la pila los dos últimos elementos introducidos y se les aplica el operador leído, lo cual da como resultado 10. El resultado obtenido es introducido en la pila.

4. Se introduce en la pila el siguiente elemento leído, 5 en éste caso.

5. Se introduce en la pila el siguiente elemento leído, 4 en éste caso.

6. El siguiente elemento leído es un operador de multiplicación, así pues se sacan de la pila los dos últimos elementos introducidos y se les aplica el operador leído, lo cual da como resultado 20. El resultado obtenido es introducido en la pila.

7. El siguiente elemento leído es un operador de suma, así pues se sacan de la pila los dos últimos elementos introducidos (10 y 20 en éste caso) y se les aplica el operador leído, lo cual da como resultado 20. El resultado obtenido es introducido en la pila.

8. Al no haber más elementos por leer, se afirma que el resultado final es el encontrado en la pila, 30 en éste caso.

2.4.5 Optimización de código

Se puede lograr que el código directamente producido por el proceso de compilación se ejecute más rápido o que ocupe menos espacio o ambos. Los compiladores que aplican transformaciones para mejorar el código se denominan "*compiladores optimizadores*".

A continuación se presentan los criterios a seguir para aplicar las transformaciones para mejorar el código:

- Una transformación debe preservar el significado de los programas; es decir, Una "optimización" no debe cambiar el resultado producido por un programa o crear algo que no estuviera presente en la versión del programa fuente original.

- Una transformación debe justificar el tiempo invertido en ella. No tiene caso invertir tiempo en el proceso de optimación si el resultado que se obtendrá no es recompensado cuando se ejecuten los programas.

2.4.6 Generación de código

En general, la entrada para el generador de código consta de la representación intermedia del programa fuente, junto con la información de la tabla de símbolos. Se asume que antes de la generación de código, la etapa inicial ha hecho el análisis léxico y sintáctico, y traducido el programa fuente a una representación intermedia razonablemente detallada. La salida del generador de código es el *código objeto*. La salida del generador de código puede adoptar una variedad de formas: lenguaje de máquina absoluto, lenguaje ensamblador, otros compiladores realizan el trabajo del ensamblador produciendo código de máquina relocalizable que se puede pasar directamente al editor de carga y enlace.

El código ensamblador es una versión mnemotécnica del código de máquina, en la cual se usan nombres en lugar de códigos binarios, y también se usan nombres para las direcciones de memoria. Producir como salida un código en lenguaje ensamblador facilita el proceso de generación de código. Se pueden generar instrucciones simbólicas y utilizar las macros del ensamblador para ayudar a generar el código.

Producir como salida un código en lenguaje de máquina absoluto tiene la ventaja de que se puede colocar en una posición fija de memoria y ejecutarse inmediatamente. Mientras que producir como salida un programa en lenguaje de máquina relocalizable permite que los subprogramas se compilen por separado; así, un conjunto de salidas de código objeto relocalizables se puede alcanzar y cargar para su ejecución mediante un cargador enlazador.

Aunque muchos compiladores obtienen el producto final ya descrito, hay otros que producen simplemente otro archivo, cuyo formato previamente definido, puede ser interpretado por algún otro programa, que comúnmente se conoce como máquina virtual.

2.4.7 Detección y manejo de errores

Cada fase, en el proceso de compilación, puede encontrar errores. Después de detectar un error, cada fase debe tratar de alguna forma el error de forma que sea posible continuar con el proceso de compilación; y así, encontrar más errores en el código fuente. Resulta obvio que un compilador que se detiene en el momento en que encuentra un error no resulta funcional.

2.4.8 Administrador de la tabla de símbolos

La tabla de símbolos es una estructura de datos que contiene un registro por cada ***identificador,*** cada registro contiene los campos para los atributos del identificador en turno. La estructura de datos permite encontrar rápidamente el registro de cada identificador y almacenar o consultar datos de dicho registro.

Un compilador utiliza una tabla de símbolos para llevar un registro de la información sobre el ámbito y el enlace de los nombres. Se examina la tabla de símbolos cada vez que se encuentra un nombre en el texto fuente. Cuando el analizador detecta un identificador en el programa fuente, el identificador se introduce en la tabla de símbolos. Sin embargo, normalmente los atributos de él, no pueden ser determinados en el análisis léxico.

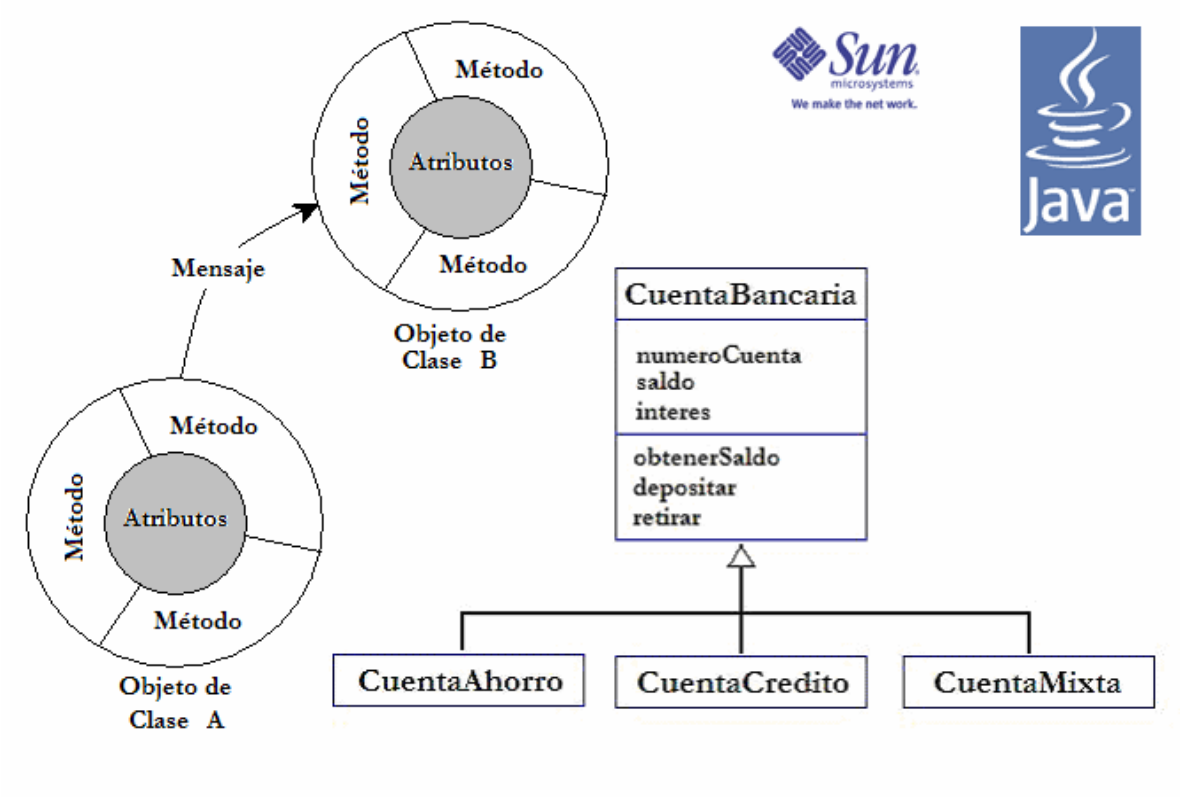
Una función esencial de un compilador es registrar identificadores utilizados en el programa fuente y reunir información sobre los distintos atributos de cada identificador. Esos atributos pueden proporcionar información acerca de la memoria asignada a un identificador, su tipo, su ámbito y en el caso de procedimientos cosas como el número y

tipos de parámetros o argumentos, la forma de pasar cada argumento (por valor o referencia) y el tipo devuelto.

2.5 Aplicaciones de los compiladores

La diversidad de aplicaciones que se dan a los compiladores es bastante amplia. Ahora mismo, en diversos lenguajes, se desarrollan aplicaciones para áreas humanísticas, médicas, financieras, de investigación, académicas, etc. Así por ejemplo, es común en este momento encontrar compiladores de diversos lenguajes para obtener programas objeto que puedan ser empleados por ejemplo en dispositivos móviles o electrodomésticos inteligentes; además, por supuesto, de los ya conocidos y rentables usos en el desarrollo de sistemas empresariales.

Segunda parte



Orientación a objetos y Java[23]

[23] Sun, el logotipo de Sun, el logotipo de Java y Java son marcas comerciales o marcas comerciales registradas de Sun Microsystems, Inc.

3 Orientación a Objetos

La Orientación a Objetos (OO) es un concepto que abarca cualquier tipo de desarrollo, de software, basado en la idea de "Objeto". En la que un objeto es una entidad con características y comportamiento. Es posible aplicar un enfoque OO tanto a la programación como al análisis y al diseño[24].

Una manera de ver la orientación a objetos es, como una forma de pensar en la que tanto el mundo que nos rodea como los problemas que se buscan solucionar, y que se quiere modelar, pueden ser descritos o considerados en términos de objetos.

3.1 Paradigmas de la programación

En los inicios de la programación los desarrolladores incorporaban los programas directamente en la memoria principal de la computadora mediante bancos de interruptores, escribían sus programas en lenguajes binarios comprensibles para las computadoras; pero con la gran desventaja de ser muy susceptibles a errores. La falta de una estructura consistente hacía prácticamente imposible dar mantenimiento a dicho código.

Conforme las computadoras se popularizaron, empezaron a aparecer los lenguajes procedurales (también conocidos como imperativos o de procedimiento) de alto nivel. Dichos lenguajes permiten al desarrollador reducir un programa a un mínimo de procedimientos, los cuales definen la estructura general del programa. Las llamadas en secuencia a estos procedimientos dan lugar a la ejecución del programa, el cual finaliza una vez que han sido llamados todos los procedimientos de la lista. Aunque el código programado con esta filosofía era fácil de entender, comparado con la programación directa en lenguaje de máquina, tenía la gran desventaja de una muy limitada reutilización de código y de que frecuentemente los desarrolladores producían lo que comúnmente se

[24] APRENDIENDO PROGRAMACIÓN ORIENTADA A OBJETOS, Sintes Anthony, edit. Prentice Hall, primera edición, 2002, págs. 8 -11.

conoce como “código espagueti”; es decir, código cuya ruta de ejecución semejaba un recipiente con espagueti.

```
private  sub  command_clientt()
   valor = 4
   if ( valor > 0 ) then
         if ( valor > 1 ) then
            GoTo cosa3
   cosa1 :
         MsgBox(“Cosa 1”)
         GoTo  cosa4
   cosa2 :
         MsgBox(“Cosa 2”)
         exit
   cosa3 :
         MsgBox(“Cosa 3”)
         GoTo  cosa1
   cosa4 :
         MsgBox(“Cosa 4”)
         GoTo  cosa2
  . . .
```

Figura 3-1 Programa hipotético. Ilustra la programación tipo espagueti

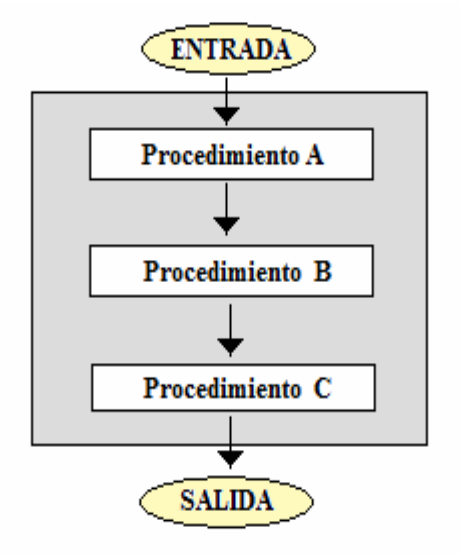


Figura 3-2 Programación modular

La programación modular (o por módulos), mostrada en la figura 3-2, divide un programa en una gran cantidad de módulos. Un módulo se compone de datos y procedimientos que manipulan a dichos datos. Cuando otras partes del programa necesitan aprovechar la funcionalidad de un módulo simplemente lo utilizan. Entre los inconvenientes de la programación modular se tiene que los módulos no son extensibles y que tampoco es posible basar un módulo directamente en otro en lo que hoy día conocemos como herencia.

La Programación Orientada a Objetos tiene sus orígenes en la programación basada en procedimientos, al igual que otros paradigmas intenta aprovechar las fortalezas y corregir las debilidades de los paradigmas predecesores. El enfoque de la programación orientada a objetos está dado por la observación de que en el mundo real gran parte de las cosas que nos rodean se pueden ver como objetos, donde, a su vez dichos objetos pueden estar basados o compuestos de objetos más pequeños.

Este último paradigma, agrega conceptos como la herencia, el polimorfismo y algunos otros conceptos usados en la implementación de la solución de un problema, con lo que cubre las carencias de su predecesor.

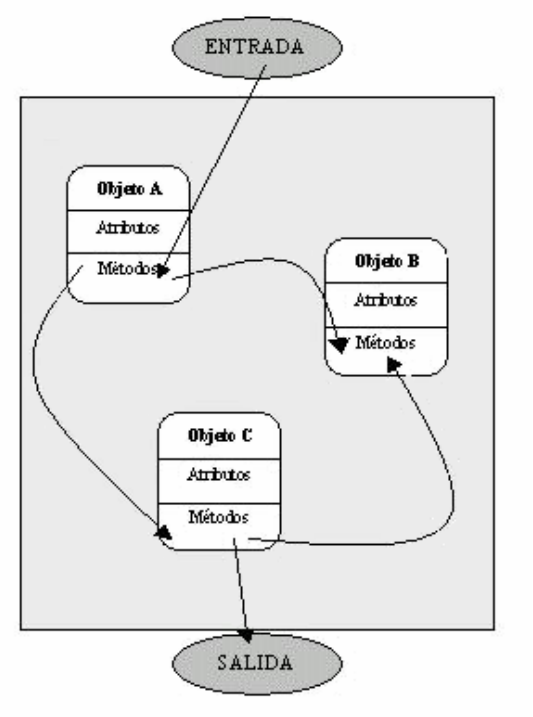


Figura 3-3 Programación orientada a objetos

3.2 Programación Orientada a Objetos

Como se comentó, los enfoques de la programación han cambiado drásticamente desde la invención de las computadoras. En un inicio se proporcionaban instrucciones en lenguaje de máquina, después mediante el uso de mnemónicos. Con el crecimiento en tamaño y complejidad de los sistemas, los lenguajes de alto nivel aparecieron para proporcionar al programador más herramientas con las cuales gestionar esa complejidad; sin embargo, cuando los proyectos alcanzan cierto tamaño, aún usando la programación estructurada, su complejidad se vuelve demasiado difícil de ser controlada.

La Programación Orientada a Objetos (POO) es una forma de enfocar la tarea de programación. La POO toma las mejores ideas de la programación estructurada, la

combina con nuevos y poderosos conceptos que animan o alientan una nueva visión de la tarea de la programación.

La definición de un programa en términos de objetos es una forma de visualizar al software. Los objetos permiten darle forma a un programa en términos naturales y reales, en vez de pensar en sólo un conjunto de procedimientos y datos.

La POO permite descomponer fácilmente un problema en subgrupos de partes relacionadas. La programación orientada a objetos se basa en la observación de que, en el mundo real, los objetos se construyen a partir de objetos más pequeños. Así que encontrará que en las grandes aplicaciones, realizadas con esta filosofía de la programación, la estructura del programa es dividida en una cantidad finita de objetos de alto nivel en la cual cada objeto le da forma a algún aspecto de un problema que se intenta resolver

El enfoque principal de la POO no es el flujo de un conjunto de llamadas a procedimiento, ahora los objetos interactúan entre si para dar lugar al flujo total del programa.

3.2.1 Objetivos de la programación orientada a objetos

Se debe esforzar el desarrollador en construir software que permita alcanzar las 6 metas que establece la POO:

- **Natural.** El encapsulamiento, del cual se hablará más adelante, divide la responsabilidad de una manera natural para la forma de pensar de la mayoría de la gente.

- **Confiable.** Al aislar la responsabilidad y ocultar la implementación, se puede verificar y validar cada componente de forma individual.

- **Reutilizable.** La abstracción, concepto abordado más adelante, permite producir código flexible que se puede utilizar en más de un contexto.

- **Mantenible.** El código encapsulado es más fácil de mantener, pues en general sólo se trabaja en el componente a modificar.

- **Extensible.** Se pueden realizar mejoras y cambiar la funcionalidad sin alterar el código existente.

- **Oportuno.** Al dividir el software en componentes autónomos, es posible repartir entre varios programadores la tarea de generación de dichos componentes.

3.2.2 Conceptos básicos dentro de la POO

La POO agrega la herencia y el polimorfismo con ello cubre carencias de su predecesor. Sin embargo, éstas características no son las únicas que son incluidas en la POO. En seguida se enuncian los principales conceptos que incluye la POO.

3.2.2.1 Objeto

Para la OO un objeto es un componente de software que encapsula un estado y un comportamiento. Los objetos permiten dar forma al software en términos y abstracciones reales.

Como se comentó anteriormente, la programación orientada a objetos se basa en la observación de que en el mundo real, los objetos se construyen a partir de objetos más pequeños. Así por ejemplo, si se considera al cuerpo humano como un objeto, éste a su vez está compuesto de una gran cantidad de objetos (los órganos de que se compone el cuerpo serían un ejemplo) donde cada uno de ellos tiene diferente estado y comportamiento. No

sólo eso, éstos últimos también podrían estar integrados por objetos y así sucesivamente hasta llegar a una unidad indivisible.

Adicionalmente, en la implementación de código en la POO, un objeto es una instancia de una clase. Tanto ***instancia*** como ***clase*** son conceptos que enseguida serán explicados.

3.2.2.2 Clase

Una Clase define los atributos y comportamientos que comparte un tipo de objeto. Los objetos de un tipo o clasificación determinado comparten los mismos comportamientos y atributos. Las clases actúan en forma parecida a una plantilla que se utiliza para crear o *instanciar* objetos.

Ha quedado claro que al agrupar elementos con características similares, estamos realizando un proceso de clasificación. Los osos, perros, gatos, patos, peces pueden entrar en una clasificación denominada "Animales". Los gatos, los leones, los tigres son "Animales" y también pueden ser clasificados como "Felinos", es decir que un mismo elemento puede entrar en más de una clasificación.

3.2.2.2.1 Instancia

Es un poco complicado explicar y asimilar el concepto de instancia, así que se recurrirá a un ejemplo para dar una idea clara del concepto.

¿Cómo describiría un Libro? quizá, como una colección de hojas que tienen un contenido escrito referente a algún tema en particular, es escrito por una o más personas, puede ser de uno o muchos colores, estar impreso en diversos tipos de papel, ser de diferente tamaño y con diferente tipo de letra, proporciona información, entretenimiento, etc. Creo que todos tenemos una idea clara de un "***Libro***". Pues bien, todas esas

características y usos que mencionamos podrían organizarse para formar un molde para hacer "*Libros*", es decir crear la *clase* ***Libro***. Una instancia de la *clase* ***Libro*** sería cada uno de los millares o millones de libros que existen en el mercado.

3.2.2.2.2 Atributos

Los atributos son las características externas y visibles que describen a un objeto. Así pues, el color de ojos, de pelo, el género, la edad, etc. podrían ser atributos de una clase *Humano*.

3.2.2.2.3 Comportamiento

El comportamiento es la acción que realiza un objeto cuando se le pasa un mensaje o en respuesta a un cambio de estado.

3.2.2.3 Encapsulamiento

En lugar de considerar la solución a un problema como un enorme programa, el encapsulamiento permite dividir un programa en componentes más pequeños autónomos e independientes. Cada componente toma una funcionalidad, oculta los detalles de la implementación al mundo exterior, de forma que cada entidad puede ser vista como una caja negra. Cada componente se comunica con el exterior a través de una *"interfaz"*, que no es otra cosa que la lista los servicios proporcionados por cada componente como un medio de comunicación con el mundo exterior. En la práctica, al programar se utilizan algunas instrucciones para modificar, limitar o permitir el acceso a un componente[25].

Las características de un encapsulamiento eficaz son:

[25] Java utiliza las palabras reservadas public, private y protected para controlar el acceso; sin embargo se hablará de ellas más adelante. En realidad Java cuenta con 4 tipos de acceso, pero con sólo 3 modificadores de acceso que son los ya citados.

- Abstracción

- Ocultamiento de la implementación

- División de la responsabilidad.

3.2.2.3.1 Abstracción

La abstracción es el proceso de simplificar un problema complejo, esto se logra enfocándose sólo en los aspectos relevantes en la solución del problema. Así, mediante la abstracción, se puede modelar el problema en términos de éste, no en los de una implementación específica. La abstracción, da la oportunidad de pensar y programar en un contexto general de modo que colabora en el proceso de reutilización.

Quizá, la abstracción sea uno de los conceptos más difíciles de asimilar dentro de la POO, así que se presenta a continuación un ejemplo con el fin de aclarar dicho concepto.

> Considere un establecimiento que elabora pizzas. Dicho establecimiento cuenta con una máquina que consiste de una banda a la cual se le aplica el calor suficiente para hornear cada pizza y que hace su recorrido en un tiempo determinado. En dicha banda se colocan en forma secuencial cada una de las pizzas que los clientes van solicitando. El proceso funciona de forma tal que la primera pizza solicitada y puesta en la banda de la máquina, será la primera que saldrá y será entregada al respectivo cliente.

Aunque el ejemplo presenta una situación específica; de manera indirecta se deduce que también el orden en el que vayan llegando los clientes a solicitar una pizza, será también el orden en que la reciban. Se puede observar que hay una descripción similar que funciona en ambas situaciones, así que se podría buscar crear una descripción genérica que funcione para los dos casos. Dicho de otra forma se podría crear una abstracción.

Observe que quizá para el proceso primero en entrar primero en salir, aplicado tanto a las pizzas como a los clientes, no es tan importante el objeto sobre el que aplica, sino más bien la forma en que se aplica, que en éste caso es la misma. Lo relevante es que los elementos que entren primero a la "*fila*" sean los primeros en salir de ella. Mediante la abstracción puede crear un programa que recree esta situación.

Se puede anticipar que el proceso: "el primero en entrar es el primero en salir" se puede presentar en un número muy grande de situaciones. En la actualidad muchos lenguajes de programación incluyen componentes o programas que recrean dicho proceso, el cual es conocido en la mayoría de ellos como "Cola". Queda claro que el propósito de la reutilización se cumple.

3.2.2.3.2 Ocultamiento de la implementación

Se ocultan los detalles de su implementación al mundo exterior, permitiendo comunicarse con él mediante una interfaz externa. Al ocultar la implementación detrás de una interfaz se protege al objeto de un uso destructivo o diferente del original.

El ocultamiento de la implementación conduce a un diseño más flexible porque evita que los usuarios del objeto dependan estrechamente de la implementación que sustenta al objeto. Así, el ocultamiento de la implementación no solo protege al objeto sino también a aquellos que lo utilizan, pues fomenta la creación de código moderadamente ligado al objeto. La dependencia de objetos y/o componentes no se puede eliminar totalmente; a pesar de ello, es importante minimizar la dependencia entre ellos.

3.2.2.3.3 División de la responsabilidad

Si se desea generar verdadero código moderadamente desligado, también debe contar con una división apropiada de la responsabilidad. Esto significa que cada componente en que es divido un programa, tiene una funcionalidad de la cual él mismo es responsable.

3.2.2.4 Herencia

Antes de hablar de herencia se abordará el tema de la generalización, "La generalización es una relación entre un elemento más general y un elemento más específico, donde el elemento más específico es totalmente coherente con el elemento más general pero contiene más información"[26].

Conceptualmente, la generalización es una idea simple. Para mostrar el concepto, imagine como elemento general un *árbol*, y luego elementos más específicos como un *roble*, un *pino*, etc. que son un tipo particular de *árbol*. Algo importante dentro de la generalización es que se puede usar el elemento más específico en cualquier lugar que se espere el elemento más general sin romper un sistema.

La generalización aplica también a las clases. Así, cuando organiza clases en una jerarquía de generalización, implícitamente tiene herencia entre las clases. La herencia permite tomar una clase existente como base para definir una nueva clase. Cuando una clase se basa en otra, la definición de la nueva clase, adquiere automáticamente todos los atributos comportamientos e implementaciones que contenga la clase base.

La herencia, permite a la clase que hereda, redefinir cualquier comportamiento que no se ajuste a su contexto o agregar nuevo comportamiento o ambas. Esa útil característica da la posibilidad de adaptar software conforme cambien los requerimientos. Así, si se necesita realizar una modificación, tan sólo se tiene que escribir una nueva clase que herede la funcionalidad anterior, acto seguido, redefine la funcionalidad que necesite cambiar o agregar la que hace falta. Se observa claramente que la herencia, de forma natural, permite:

- **Redefinición.** Es el proceso mediante el cual una clase hija toma un método de la clase madre y lo reescribe con el propósito de cambiar su comportamiento. La

[26] PROGRAMACIÓN UML 2, Arlow Jim, Neustadt Ila, edit. Anaya, 2006, pág 227.

forma en que el objeto sabrá cual definición de método usar es la siguiente: Primero se buscará la definición en el objeto al que se le pasa el mensaje; si no hay definición ahí, al ejecutarse el programa se recorrerá la jerarquía hacia arriba hasta encontrar una definición.

- **Especialización.** Es el proceso mediante el cual una clase se define a si misma en términos de los aspectos que la hacen diferente de su clase madre. Una clase hija se especializa respecto de su madre agregando nuevos atributos y métodos, así como redefiniendo métodos existentes.

La siguiente figura presenta un ejemplo de herencia. Suponga que tenemos el caso típico: Un Banco, en el cual se manejan varios tipos de cuentas:

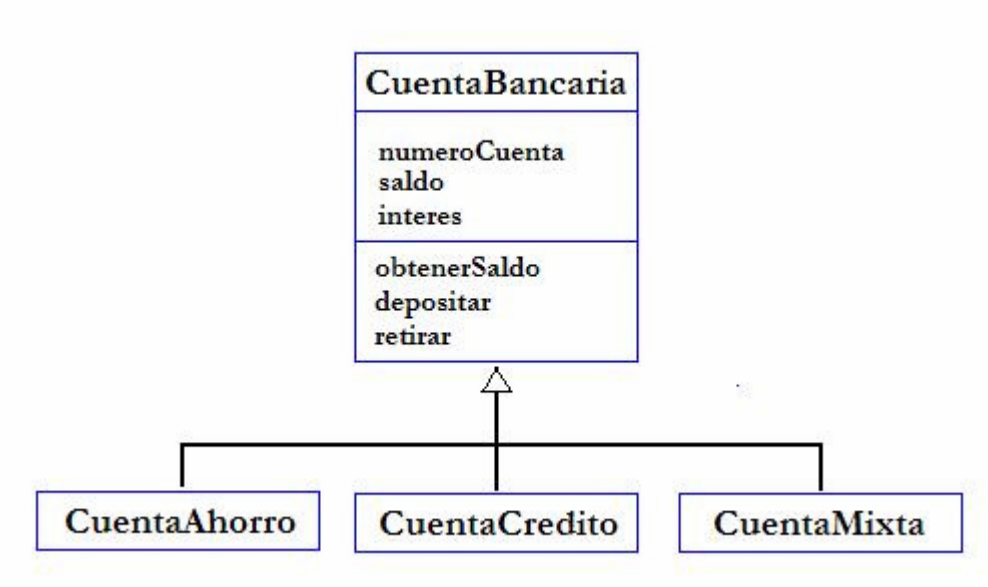


Figura 3-4 Clases que ilustran el empleo de Herencia

Se puede observar que la clase *CuentaBancaria* tiene en este momento tres atributos denominados: *numeroCuenta, saldo e interes*. En el primero se establece el número de cuenta que identifica a cada usuario, en segundo lugar está el saldo que será la cantidad que posee o debe el poseedor de la cuenta, por último está el *interes* generado por concepto de mantener la cuenta. Se encuentran, además, tres métodos: *obtenerSaldo, depositar y retirar*. Los cuales permitirán conocer el saldo en el primer caso, aumentar el saldo en el segundo y disminuir el saldo en el último.

Adicionalmente, en la figura 3-4 hay tres clases heredando de la clase *CuentaBancaria*. Es interesante, que por el simple hecho de heredar de *CuentaBancaría*, las tres clases mostradas, poseen los mismos atributos y los mismos métodos. Sin embargo, el nombre de las clases nos indica que deberían hacer cosas diferentes. Así, por ejemplo en cuenta de ahorro (*CuentaAhorro*) se esperaría que siempre hubiera un saldo mayor a cero; es decir que, un "*cuenta habiente*" tiene guardado su dinero en el Banco, mientras que en una cuenta de crédito (*CuentaCredito*) se esperaría que se tuviera un saldo negativo o igual a cero; es decir, el *cuenta habiente* le debe dinero al banco. *CuentaAhorro* y *CuentaCredito*, podrían tener un atributo más que determinará el periodo a partir del cual se generaran intereses. Se puede ver que a *CuentaCredito* se podría agregar un atributo (es obvio que podrían agregarse tantos como fueran necesarios) que determinará el límite de crédito que tendrá el cuenta habiente a el Banco. Por otra parte, *CuentaMixta* podría ser una cuenta que pudiera tener, por ejemplo, una combinación de los dos casos anteriores. Sin embargo, se puede ver que el universo de cuentas que se puede generar es amplio.

3.2.2.5 Polimorfismo

Significa "muchas formas". En términos de programación, el polimorfismo permite que un nombre de clase o método represente diferente código seleccionado mediante algún mecanismo automático que se presenta sólo a través de la herencia. La forma en que se logra el polimorfismo es utilizando un método que tienen en común las clases que provienen de la misma jerarquía de clase. Cada clase implementará el método común de forma que le resulte conveniente; por tal razón al hacer uso del método, éste responderá según la clase instanciada.

Retomando el ejemplo presentado en el apartado anterior. Se tenía que *CuentaAhorro* y *CuentaCredito* presentan un método denominado *depositar* y otro *retirar*. Es de esperar que tengan resultados diferentes al usar el método *depositar* en *CuentaAhorro* y en *CuentaCredito*. En el caso de *CuentaAhorro*, un cliente de un Banco entrega dinero a

un Banco esperando recibir, al final del periodo acordado, una compensación económica por concepto de intereses. En el segundo caso, un cliente al *depositar*, abona al Banco una parte o el total de la deuda que tiene con el banco, que ahora incluye el saldo inicial más los intereses generados en el periodo que ha mantenido el préstamo. Por otra parte, cuando un cliente *retira*, de *CuentaAhorro*, obtiene una parte o el total del dinero que el depositó junto con los intereses generados en el periodo acordado; mientras, en el caso de *CuentaCredito* se establece una cuota cada que el cliente realiza un retiro (a criterio de cada banco).

Se puede observar que el comportamiento de cada método es diferente en cada clase presentada. Sin embargo se pueden trabajar a partir de la clase base y el comportamiento obtenido dependerá, como ya se dijo, de la clase de la cual se genere una instancia. De esta forma, se verá un comportamiento polimorfito que aplicará según sea conveniente a las necesidades del desarrollo a realizar.

4 Java

De la forma en que la vida real está conformada por objetos, también lo está el software orientado a objetos. En un lenguaje de programación orientado a objetos puro, todo elemento es un objeto, desde los tipos de datos más básicos como los enteros y los booleanos, hasta las instancias de clases más complejas.

No todos los lenguajes orientados a objetos son considerados puros. Java y C++, que tienen sus antecedentes del enfoque basado en procedimientos, son un ejemplo de lenguajes híbridos donde se cuenta con tipos de datos primitivos simples como *int* y *float* que no son considerados como objetos.

El lenguaje Java fue desarrollado en Sun Microsystems en 1991 como parte del proyecto *Green*, un grupo de investigación con la misión de desarrollar software para el control inteligente de dispositivos electrónicos dirigidos al consumidor final. El proyecto tuvo serías dificultades, ya que los dispositivos electrónicos inteligentes de uso doméstico no se desarrollaban tan rápido como Sun había anticipado. Sin embargo, con la explosión de la World Wide Web en 1993, la gente de Sun se dio cuenta del potencial de Java para generar contenido dinámico y animaciones a las páginas Web. Esto trajo nueva vida al proyecto.

En la actualidad Java se utiliza para desarrollar aplicaciones empresariales a gran escala, para mejorar la funcionalidad de los servidores de World Wide Web, para proporcionar aplicaciones para los servicios domésticos fijos y móviles, y para muchos otros propósitos.

4.1 El lenguaje Java

Como la mayoría de los lenguajes de programación, los archivos que contienen el código fuente en Java, se almacenan como archivos de texto simple con extensión "*.java*". Cada archivo contiene típicamente una clase o una interfaz, aunque puede contener un número mayor de ellas.

Al escribir un programa en lenguaje Java, el programador diseña y construye un conjunto de clases. Cuando el programa se está ejecutando, los objetos se crean desde dichas clases y se usan conforme se van necesitando. La tarea del programador es crear, empleando los conceptos de la POO, un conjunto adecuado de clases para llevar a cabo lo que el programa requiere.

Cada clase desarrollada en Java es independiente, autónoma, tiene una funcionalidad y presenta una forma de comunicarse con el exterior. Los atributos y métodos, contenidos en una clase, interactúan para lograr que se obtengan los resultados deseados. Un conjunto secuencial de instrucciones son capturadas por el programador, dentro de cada método. Dichas instrucciones se apegan al teorema de la estructura o de *Böhm-Jacopini*, el cual básicamente dice que: "Todo algoritmo propio puede ser expresado en términos de solo tres tipo de estructura: secuencial, condicional y repetitiva". Un algoritmo propio es aquel que cumple las siguientes condiciones[27]:

- Tiene un único punto de entrada y uno de salida
- Se lee de forma secuencial
- Todo elemento del programa es accesible

[27] FUNDAMENTOS DE PROGRAMACIÓN. Peñaloza R. Ernesto, edita UNAM, tercera edición, 2001, pág. 72.

Sun[28], describe algunas características de Java como: lenguaje simple, orientado a objetos, distribuido, robusto, seguro, de arquitectura neutra, portátil y de alto desempeño.

4.1.1 Universo de palabras reservadas del Lenguaje Java

El lenguaje java contiene 49 palabras reservadas que el compilador usa para averiguar lo que el *código fuente* trata de hacer. Dichas palabras reservadas no pueden ser usadas para identificar clases métodos o variables.

A continuación se presentarán, de forma organizada, las palabras reservadas del lenguaje Java[29] que serán abordadas por el "Lenguaje Java Español" que se presentará en el siguiente capítulo.

4.1.1.1 Modificadores de acceso

Los siguientes son los modificadores de acceso:

- **private.** Hace un método o variable accesible sólo desde su misma clase.

- **protected.** Hace un método o variable accesible sólo para clases en el mismo paquete o subclases de la clase.

- **public.** Hace una clase, método o variable accesible desde alguna otra clase.

Java posee 4 tipos de acceso que son los arriba citados y uno más conocido como de: *paquete* ; sin embargo, sólo posee tres modificadores de acceso[30].

---

[28] JAVA EN POCAS PALABRAS, Flanagan David, edit. Mc Graw Hill, primera edición, 1999, pág. 3.
[29] THA JAVA LENGUAGE SPACIFICATION, Gosling James, Joy Hill, Steele Guy, Bracha Gilad, Addison Wesley, tercera edición, 2005. Permitirá a los lectores un obtener un conocimiento profundo de la especificación del lenguaje java.

4.1.1.2 Modificadores de clase, método y variable

En seguida se presentan algunos de los modificadores de clase, método y variable[31]:

- **abstract.** Usado para declarar una clase que no puede ser instanciada, o un método que debe ser implementado por una subclase no abstracta.

- **class.** Palabra reservada para especificar una clase.

- **extends.** Usado para indicar la superclase que una subclase está extendiendo.

- **implements.** Usado para indicar las interfaces que la clase implementará.

- **new.** Usado para instanciar un objeto invocando el constructor.

- **static.** Hace un método o variable para toda la clase, en vez de generar uno por cada instancia.

4.1.1.3 Control de flujo

Las siguientes palabras son usadas para controlar el flujo entre bloques de código[32]:

- **if.** Usado para realizar pruebas lógicas que como resultado entreguen falso o verdadero. En caso de que la condición evaluada sea verdadera se ejecuta el bloque de código que aparece a continuación de la expresión evaluada.

---

[30] Acudir al documento JAVA 2 SUN CERTIFIED PROGRAMMER & DEVELOPER Parte 1, Cap. 2 págs. 17-25, elaborado por SUN como preparación al examen de certificación; para una mayor ampliación del tema.

[31] La totalidad de modificadores de clase, método y variable, del lenguaje Java, se pueden encontrar en el documento electrónico JAVA 2 SUN CERTIFIED PROGRAMMER & DEVELOPER, Parte 1, Cap. 1, págs. 5-6, elaborado por SUN como preparación al examen de certificación.

[32] La totalidad de controladores de flujo, del lenguaje Java, se pueden encontrar en el documento electrónico JAVA 2 SUN CERTIFIED PROGRAMMER & DEVELOPER, Parte 1, Cap 1, págs. 6-7, elaborado por SUN como preparación al examen de certificación.

- **else.** Ejecuta un bloque de código alternativo en caso de que la condición **if** evaluada sea falsa.

- **while.** Ejecuta un bloque de código de forma repetitiva mientras la condición evaluada sea verdadera.

- **return.** Permite salir de un método sin ejecutar algún código que exista después de la palabra "*return*". Opcionalmente, permite regresar una variable al salir del método.

4.1.1.4 Control de paquete

A continuación se muestran las palabras reservadas usadas para el control de paquete:

- **import.** Sentencia para importar paquetes o clases dentro del código fuente.

- **package.** Especifica el paquete al que pertenece cada clase.

4.1.1.5 Datos primitivos

Las siguientes palabras reservadas son empleadas para señalar tipos de datos primitivos:

- **boolean.** Un valor indicando *false* o *true*.

- **int.** Un entero de 32 bits.

- **double.** Un número de punto flotante de 64 bits.

- **char**. Un carácter UNICODE de 16 bits.

4.1.1.6 Tipo de retorno void

La palabra reservada *void* es usada sólo en la declaración de un método, en el sitio empleado para declarar el tipo regresado.

- **void.** Indica que un método no regresa un tipo.

4.1.2 Estructura de una clase en Java

Cada clase desarrollada en Java es independiente, autónoma, tiene una funcionalidad y presenta una forma de comunicarse con el exterior. Los atributos y métodos, contenidos en una clase, interactúan para lograr que se obtengan los resultados deseados.

Una clase en Java se presenta con la siguiente estructura:

```
Declaración de paquete
Declaraciones de clases a importar
Declaración de clase
{
        Cuerpo de la clase
}
```

Figura 4-1 Estructura de una Clase

4.1.3 Estructura de una interfaz en Java

Una interfaz en Java, no es como tal una clase. Su función, básicamente, es definir los métodos que deberán ser implementados por las clases a las que les sean comunes.

De forma análoga a una clase, una interfaz tiene la siguiente estructura. Aunque, se cita en este apartado por estar involucrada en la declaración de una clase, no se realizará un estudio más profundo de ella.

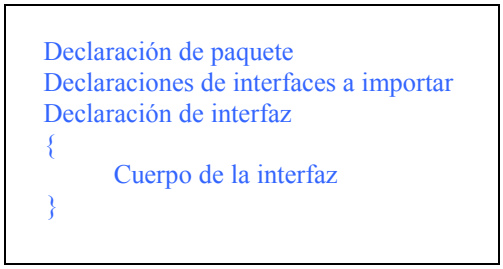


Figura 4-2 Estructura de una interfaz

4.1.3.1 Declaración de paquete

Java organiza la estructura de las *clases* desarrolladas en carpetas, es responsabilidad del programador indicar la carpeta a la que pertenece cada *clase*. En la sección de declaración de paquete se indica el paquete al cual pertenece la *clase* que se está desarrollando. Es opcional la declaración de paquete, en caso de no presentarse, el sistema busca la clase estudiada en la ubicación desde la cual fue llamado el compilador. Como máximo se puede realizar una vez y deberá ser la primera instrucción en el archivo.

package rutaDeCarpeta ;

rutaDeCarpeta, describe la ruta que permite localizar la carpeta contenedora de la clase que se desarrolla en ese momento. El árbol que describe la ruta a seguir, es delimitado empleando el símbolo *punto*.

4.1.3.2 Declaración de clases a importar

En esta sección se indican las clases que son necesarias para que la clase en desarrollo funcione. Es opcional realizar esta declaración, en caso de no presentarse, el sistema supone

que no hay necesidad de ninguna otra clase para que la clase en desarrollo funcione. Se puede agregar la cantidad de declaraciones que sean necesarias. El usar un asterisco al final de la ruta de carpeta, en lugar del nombre de la clase a importar, indica al compilador que deberá cargar todas las clases que están en la última carpeta a la que se hace referencia.

importe rutaDeCarpeta **.** nombreClase ; | ***importe*** rutaDeCarpeta **. *** ;

rutaDeCarpeta, describe la ruta que permite localizar la carpeta contenedora de la clase que se desarrolla en ese momento. El árbol que describe la ruta a seguir, es delimitado empleando el símbolo *punto*.

nombreClase, indica el nombre de la *clase* que se necesita para que el sistema funcione.

4.1.3.3 Declaración de clase

Permite definir las características de acceso y el nombre que tendrá la clase a desarrollar. La herencia se maneja de la siguiente forma: Será posible extender de una clase como máximo; además, será posible implementar el número de interfaces que sea necesario.

tipoAcceso atributoClase ***class*** nombreClase ***extends*** nombreClaseExt ***implements*** nombreClasesImp

tipoAcceso, indica el tipo de acceso que tendrá la clase: *public* ó de *paquete*. En caso de no emplear el modificador de acceso *public*, se entenderá que el tipo de acceso será el de *paquete*.

atributoClase, se pueden colocar las palabras: *static*, *abstract*, *final*. Que le darán características diferentes a la clase y que fueron comentadas en la sección anterior. No es obligatorio incluir este punto

nombreClase, se refiere al nombre que recibirá la clase y que debe coincidir con el nombre del archivo que contiene el código de la clase en estudio. Es obligatorio este punto.

nombreClaseExt, se refiere al nombre de la clase de la cual se extenderá. En caso de no presentarse, se deberá eliminar la palabra reservada *extends* de la declaración de clase.

nombreClasesImp, se refiere a los nombres de las interfaces que se deberán implementar por la clase en desarrollo. Cada interfaz estará separada por un símbolo *coma.* (,). En caso de no requerirse interfaz alguna, se deberá eliminar la palabra reservada *implements* de la declaración de clase.

4.1.3.4 Cuerpo de la clase

Java reconoce dos tipos de elementos dentro del cuerpo de su clase, los atributos y los métodos. Sin embargo, la forma en que en Java se definen los métodos, nos hace considerar a los métodos constructores por separado. Así, el cuerpo de una clase tendrá tres tipos de elementos:

- Atributos
- Métodos
- Constructores

El número de elementos agregados al cuerpo de cada clase, dependerá de las necesidades del programa desarrollado. El orden de aparición no es trascendente en el funcionamiento de la clase en desarrollo.

4.1.3.4.1 Atributo

Un atributo es una característica de la clase. Su declaración está dada de la siguiente forma:

tipoAcceso caractAtributo tipoDato nombreAtributo [= valorAtributo] ;

tipoAcceso, corresponde a la visibilidad y accesibilidad que tendrá el atributo: *private*, *protected*, *public* o de *paquete*. En caso de no emplear alguno de los modificadores de acceso, se entenderá que el tipo de acceso será el de *paquete*.

caractAtributo, en este punto determinamos algunas características del atributo, por ejemplo: *static*, *const*, *final*, etc.

tipoDato, el tipo de dato de un atributo puede corresponder a un tipo primitivo o a una clase previamente desarrollada. Si se trata de éste último caso, la clase empleada deberá estar incluida en la declaración de clases importadas o en su defecto pertenecer al mismo paquete que la clase en estudio.

nombreAtributo, el nombre del atributo será a elección del desarrollador, procurando que describa el objetivo del atributo. Es necesario que los nombres de los atributos declarados no se repitan.

valorAtributo, es el valor asignado a un atributo. Puede ser un valor primitivo o una nueva instancia. Se presenta de forma opcional, aunque Java asigna de forma automática uno en caso de que un atributo no tenga un valor asignado desde la declaración.

4.1.3.4.2 Método

Un método describe el comportamiento de una clase. Su declaración está dada de la siguiente forma:

tipoAcceso caractMetodo tipoDato nombreMetodo (parámetros) ; |
tipoAcceso caractMetodo tipoDato nombreMetodo (parámetros) { cuerpoMetodo }

tipoAcceso, corresponde a la visibilidad y accesibilidad que tendrá el método: *private*, *protected*, *public* o de *paquete*. En caso de no emplear alguno de los modificadores de acceso, se entenderá que el tipo de acceso será el de *paquete*.

caractMetodo, en este punto determinamos algunas características del método, por ejemplo: *static*, *final*, abstract, etc. Observe que se puede dar el caso de que el cuerpo del método es suprimido y sustituido por un símbolo de *punto y coma*, en tal caso caractMetodo deberá ser igual a *abstract* para indicar que se trata de la declaración de un método abstracto, que en consecuencia exige que la clase desarrollada sea abstracta.

tipoDato, el tipo de dato de un método puede corresponder a un tipo primitivo o a una clase previamente desarrollada. Si se trata de éste último caso, la clase empleada deberá estar incluida en la declaración de clases importadas o en su defecto pertenecer al mismo paquete que la clase en estudio. Existe una situación especial que se presenta cuando el tipo dato devuelto es: *void*, recordará simplemente indica que ningún valor será regresado por el método en estudio.

nombreMetodo, el nombre del método será a elección del desarrollador, procurando que describa el objetivo del método. Es necesario que cuando los nombres de los métodos declarados se repitan, los parámetros de ellos no correspondan en número y si lo hacen, que el tipo declarado (en orden de aparición) para cada uno de ellos no sea el mismo.

parámetros, separados por símbolos de coma serán declarados los tipos y nombres de variables que recibirá el método para su correcto funcionamiento.

cuerpoMetodo, es sin duda uno de los puntos más complejos, pues es en este lugar donde se escribirán las líneas de código que permitan que el método cumpla el objetivo con el que

fue declarado. Aquí se encontrarán desde sentencias simples hasta bloques de instrucciones o instrucciones de control de flujo que contendrán el orden necesario para cumplir el objetivo del método.

No se debe perder de vista que el lenguaje Java es robusto y extenso. Hay sin lugar a dudas extensiones a la declaración presentada que pueden brindar y/o agregar otras características a las ya descritas. Por otro lado, el código colocado en el cuerpo del método también puede tener muchos más elementos de los descritos en el presente apartado, así que se sugiere a los lectores referirse a algún libro de texto en caso de requerir extender sus conocimientos del área.

4.1.3.4.3 Constructor

Es un método con una declaración especial que generalmente se usa para inicializar atributos y/o preparar un objeto para su correcto funcionamiento. Su declaración está dada de la siguiente forma:

tipoAcceso nombreConstructor (parámetros) { cuerpoMetodoConstructor }

tipoAcceso, corresponde a la visibilidad y accesibilidad que tendrá el constructor: *private*, *protected*, *public* o de *paquete*. En caso de no emplear alguno de los modificadores de acceso, se entenderá que el tipo de acceso será el de *paquete*.

nombreConstructor, el nombre del método constructor será necesariamente el mismo que el proporcionado a la clase desarrollada. Es necesario que cuando haya más de un método constructor, los parámetros de ellos no correspondan en número y si lo hacen, que el tipo declarado (en orden de aparición) para cada uno de ellos no sea el mismo.

parámetros, separados por símbolos de coma serán declarados los tipos y nombres de variables que recibirá el método para su correcto funcionamiento.

cuerpoMetodoConstructor, es en general, es el punto en el cual se inicializan los atributos que será necesario emplear a lo largo del funcionamiento del objeto creado de a partir de la clase usada.

Cuando se crea una instancia de un objeto, es un método constructor el primero en ejecutarse, y son los parámetros de éste, quienes determinan cual de todos los constructores se ejecutará en caso de que haya más de uno. En caso de no existir declaración de métodos constructores, el compilador crea uno, el cual no recibirá parámetros y no tendrá una funcionalidad particular.

4.2 Compilación e interpretación de programas con Java

Java es independiente de la plataforma; es decir, que a los programas compilados en Java no será necesario realizar modificaciones a fin de ejecutarlos en diferentes entornos de sistemas operativos. Para poder lograrlo, como producto de la compilación de un programa escrito en lenguaje Java, se obtiene un archivo con extensión "*.class*[33]" que puede ser comprendido por la Java Virtual Machine (JVM).

Enseguida se presenta el camino que sigue un programa escrito en lenguaje Java en el proceso de compilación:

1. El compilador de Java, se encarga de revisar el código fuente proporcionado por el usuario, en archivos con extensión "*.java*", pudiéndose presentar dos situaciones:

    - El compilador encuentra que el código fuente no se apega a las reglas sintácticas y semánticas del lenguaje Java; es decir, que encuentra al menos un error y como resultado envíe un mensaje al usuario alertando del problema para que sea corregido.

[33] El contenido de dichos archivos también es conocido como Bytecodes.

- El compilador encuentra que el *código fuente* se apega a las reglas sintácticas y semánticas del lenguaje, entregando como producto de la compilación, uno o más archivos con extensión *.class*.

2. La JVM se encarga de interpretar los archivos con extensión *.class* obtenidos en el proceso de compilación.

4.2.1 Compilador Java

Además de contar con un lenguaje de programación definido, se necesita el sistema que procese el *código fuente* realizado con dicho lenguaje.

Un compilador traduce un programa escrito en un *lenguaje fuente* en otro equivalente en un *lenguaje objeto*. Típicamente un compilador contiene las fases, previamente descritas en el capítulo 2: Análisis léxico, análisis sintáctico, análisis semántico, generación de código, etc. El compilador de Java no es la excepción; a partir de uno o más archivos fuente, se encarga de verificar que el condigo fuente proporcionado por el usuario cumpla las reglas impuestas por el lenguaje Java. Una vez satisfechas dichas reglas se entrega uno o más archivos con extensión "*.class*" que podrán ser interpretados desde cualquier JVM.

El compilador de Java es un programa denominado javac, el compilador javac está totalmente escrito en Java, por lo que es disponible para cualquier plataforma que soporte el sistema de tiempo de ejecución de Java[34]. Con *javac* el *código fuente* de Java se convierte en un nuevo archivo que contiene los *bytecodes* que serán interpretados posteriormente por la JVM.

Un único archivo puede contener múltiples clases, siempre y cuando sólo una de las

[34] LEARNING JAVA, Knudsen Jonathan, Niemeyer Patrick, Mc Graw Hill, tercera edición, 2005, capítulo 3, sección 3.4.

clases sea señalada pública. El compilador javac permite una *clase* con acceso público por archivo y exige que el archivo tenga el mismo nombre que la *clase* en estudio. Si el nombre de archivo y nombre de la clase no coinciden, javac se encargará de enviar un mensaje de error.

Como ejemplo, suponga el archivo PresentaMensaje.java con el siguiente código fuente:

```
public class PresentaMensaje{
        public static void main(String arg[]){
                System.out.println("Este es un mensaje");
        }
}
```

El comando que ejecuta el programa *javac* necesita un nombre de archivo (con la extensión .java):

```
C:/javac PresentaMensaje.java
```

El comando producirá el archivo PresentaMensaje.class. Si hay que compilar varios archivos .java, indíquelos uno tras otro separados los nombres por espacio en blanco. Si las clases guardadas en dichos archivos tienen diferente paquete, use en la instrucción javac la opción –d y la ruta donde desea que se coloquen los archivos *.class*. El siguiente podría ser un ejemplo:

```
C:\Tigre>javac -d . OtraClase.java OtroSaludo.java Saludo.java
```

Por último, es importante señalar que *javac* puede compilar una clase, incluso si las clases importadas están disponibles sólo en versiones binarias. No es necesario el código fuente para todos los objetos.

4.2.2 Consideraciones del Compilador Java

Es necesario tener presentes las siguientes consideraciones, que el compilador de Java realiza respecto al *código fuente* desarrollado con lenguaje Java:

- La clase de mayor jerarquía en Java es *Object*.

- Todas las clases desarrolladas por un programador heredan de la clase Object, si y sólo si, la declaración de una clase no indica que está heredando de alguna otra clase.

- Es opcional señalar el paquete en que se desea incluir una clase. En caso de no señalar explícitamente el paquete, la clase será colocada en la carpeta que esté usando en ese momento el desarrollador.

- El paquete *java.lang* no necesita ser importado, ya que el compilador Java lo carga de forma automática. Por lo que las clases contenidas en ese paquete siempre estarán disponibles para el desarrollador. Las clases String, System, Integer, Double, Boolean, etc. son ejemplos de las clases contenidas por dicho paquete.

- Es opcional declarar un método constructor. En caso de no hacerlo, el compilador crea uno, carente de parámetros y de funcionalidad.

- Conversiones de tipos. Java realiza la conversión de tipos de dos formas: La primera es realizada cuando el usuario señala de forma explícita[35] el tipo de dato obtenido de alguna variable, método u operación aritmética. La segunda es realizada por el compilador de forma automática, por ejemplo al realizar operaciones. El compilador Java realiza operaciones de datos que tienen el mismo tipo, así que si los datos son

[35] Para mayor información acerca de la conversión de tipos de forma explícita, acudir a APRENDIENDO JAVA 2, Lemay Laura, Cadenhead Rogers, edit. Prentice Hall, primera edición, 1999, págs. 100-102.

de diferente tipo, el compilador modifica el tipo de dato para poder realizar las operaciones.

- Los comentarios encontrados en un programa fuente, son suprimidos por el compilador.

4.2.3 Java Virtual Machine

Java es independiente de la plataforma; es decir, que en vez de modificar los programas desarrollados en lenguaje Java para ajustarlos a alguna plataforma, bastará con tener una Java Virtual Machina (JVM) que se ajuste al Sistema Operativo de su interés ya que el compilador entrega un archivo intermedio que puede ser interpretado por cualquier Java Virtual Machine (*máquina virtual Java*). Así, la máquina virtual de Java es el componente tecnológico de software responsable de la independencia del hardware y del sistema operativo, del tamaño pequeño de los archivos compilados y su capacidad para proteger a los usuarios de los programas maliciosos.

La JVM[36] es en resumen un componente que funciona como si fuera una computadora real, que cuenta con un conjunto de instrucciones y manipula diversas áreas de memoria en tiempo de ejecución. La máquina virtual no sabe nada del lenguaje de programación Java, sólo de un formato binario, que es el formato de archivo producido por el compilador Java. Un archivo *.class* contiene instrucciones de la máquina virtual de Java (o bytecodes), una tabla de símbolos, así como otra información auxiliar.

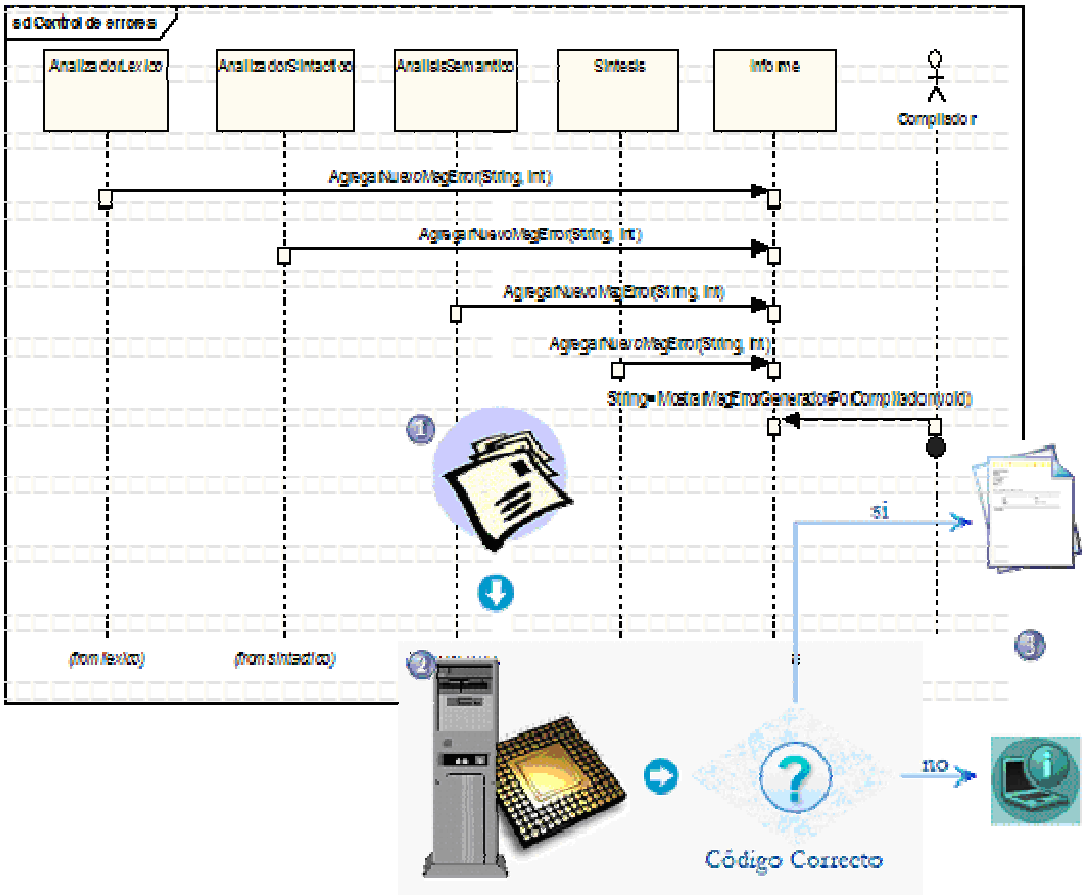
En aras de la seguridad la JVM impone fuertes limitaciones estructurales y de formato, en el código en un archivo *.class*. Sin embargo, cualquier lenguaje que pueda expresarse en términos de archivo *.class* válido podrá ser interpretado por la JVM.

---

[36] THE JAVA VIRTUAL MACHINE SPECIFICATION, Leindholm Tim, Yellin Frank, Addison Wesley, segunda edición, 1999, capítulo 1.

Actualmente, existen en el mercado JVM para casi cualquier plataforma, es pues, la JVM la responsable de comprender los *bytecodes* y encargarse de establecer la correcta comunicación con un determinado sistema operativo. Esa es la razón por la cual no resulta necesario modificar los archivos *.class* para trabajar en diferentes plataformas, vasta con usar la JVM adecuada.

Tercera parte



Caso práctico

Definición, diseño y construcción

5 Lenguaje Java en Español

En la primera parte del presente texto se conocen los conceptos básicos referentes a la teoría de lenguajes formales y autómatas, las técnicas empleadas en el manejo de algunas tareas importantes de un compilador, se presenta el objetivo de un compilador y la forma en que funciona. Una vez cubierto dicho material, quizá se podría iniciar el desarrollo de un nuevo lenguaje de programación. Sin embargo, dentro de los objetivos del presente texto no se describe la estructura ni las reglas que debe seguir el nuevo lenguaje de programación, excepto que las palabras reservadas empleadas deberán estar en español.

La segunda parte del presente texto presentó los conceptos básicos de la programación orientada a objetos con la finalidad de que el nuevo lenguaje de programación los tuviera presente. La segunda parte, también abordó el estudio de Java con el propósito de tomarlo como modelo de desarrollo del nuevo lenguaje. Al tomar, el nuevo lenguaje de programación, como modelo al *lenguaje Java* es de esperar encontrar semejanzas entre ambos.

Java, presenta un ejemplo real de un lenguaje de programación orientado a objetos, esa es la razón por la cual se toma como modelo para desarrollar el nuevo lenguaje de programación. El nuevo lenguaje a desarrollar, que de ahora en adelante será conocido como "*lenguaje Java en Español*", empleará palabras reservadas en español y buscará no perder de vista los conceptos básicos de la POO ya presentados. Sin embargo, es de esperar que tanto el lenguaje como el software desarrollado tengan limitaciones en cuanto a los puntos abarcados, hay que tener presente que Java es robusto y que sigue, desde su creación, agregando características, modificando otras y eliminando algunas más.

Así pues, en las siguientes secciones del presente capítulo, se presentará el alfabeto del nuevo lenguaje, la definición de la estructura del nuevo lenguaje y las reglas semánticas que aplicarán sobre el lenguaje.

5.1 Alfabeto del lenguaje Java en Español

Antes de desarrollar un compilador, deben establecerse las reglas que deberá verificar el compilador. Sin embargo es necesario, incluso antes de comenzar a determinar y delimitar las reglas del *lenguaje Java en Español*, se determinen los elementos del alfabeto al que serán aplicadas dichas reglas.

A continuación se presenta el universo de símbolos que serán usados en el lenguaje a desarrollar.

a	k	u	E	O	Y	9	/	-
b	l	v	F	P	Z	0	\	_
c	m	w	G	Q	1	¡	(	{
d	n	x	H	R	2	!	)	}
e	o	y	I	S	3	“	=	*
f	p	z	J	T	4	|	¿	[
g	q	A	K	U	5	#	?	]
h	r	B	L	V	6	$	,	^
i	s	C	M	W	7	%	.	:
j	t	D	N	X	8	&	;	

Tabla 5-1 Alfabeto del lenguaje Java en Español

5.1.1 Palabras reservadas

Como se comentó en el capítulo 1, un lenguaje es una colección infinita de cadenas sobre un alfabeto. Sin embargo, no podemos crear un compilador que maneje un número infinito de cadenas. Como se pudo ver en el capítulo 2, en un compilador se trabaja con estructuras gramaticales bien definidas y delimitadas. Además, en el proceso de validación de estructuras gramaticales es de gran ayuda contar con palabras, también conocidas como *palabras reservadas*, que ayuden a decidir el camino exacto que debe seguir una gramática de libre contexto en su recorrido de validación, ahorrando con ello el proceso de búsqueda del camino correcto de entre todos los recorridos posibles en los árboles sintácticos.

Se sabe que el lenguaje a desarrollar estará orientado a objetos y que por lo tanto se apoyará en la teoría presentada en el capítulo 3. En ese capítulo se presentaron algunos

conceptos que indudablemente será necesario abordar en el desarrollo del lenguaje. Sin embargo, antes de precisar como se aplicarán dichos conceptos en el *lenguaje Java en Español* es necesario definir las *palabras reservadas* que usará el nuevo lenguaje. Parte de la estructura del *lenguaje Java* será empleada en la definición del *lenguaje Java en Español*, y por comodidad algunas de las palabras reservadas que usa Java serán llevadas al *lenguaje Java en Español* en el idioma solicitado. A continuación se muestra la lista de palabras reservadas en español que serán usadas:

paquete	interfaz	abstracto	caracter	si
importar	publico	nuevaInstancia	Texto	otro
extiende	privado	entero	boleano	mientras
implementa	protegido	largo	falso	imprimir
clase	vacio	doble	verdadero	regresar

Tabla 5-2 Palabras reservadas del lenguaje Java en Español

Podrá observar con rapidez que el número de palabras reservadas es relativamente pequeño. Es necesario tener presente que el compilador diseñado estará limitado en funciones.

5.2 Definición del Lenguaje Java en Español

5.2.1 Clase

La teoría de compiladores dice, que un compilador "***traslada***" un ***código A*** a un ***código B*** o señala las razones por las cuales no fue posible trasladar dicho código. Para conseguir que un ***código A*** pueda trasladarse a un ***código B***, el compilador debe conocer cuales son las reglas o estructura que debe cumplir el ***código A***.

El ***código A*** ó ***código fuente***, en los lenguajes orientados a objetos se encuentra dividido en archivos de texto denominados "***clase***" y cada lenguaje de programación establece las reglas que han de aplicar en el proceso de compilación de dichos archivos de texto.

De la teoría de orientación a objetos, se sabe que una *clase* es un molde que servirá para crear objetos del mismo tipo. Así pues, los siguientes apartados serán empleados para dictar las reglas que aplicarán a cada *clase*.

Respecto a la redacción de *clases*, la primera regla a seguir será la sensibilidad o diferenciación entre mayúsculas y minúsculas. En cuanto a las convenciones usadas en el presente capítulo se tendrá que:

- Cuando una estructura o una parte de ella sea declarada entre corchetes, indicará que esa parte es opcional; es decir, se puede prescindir de ella.

- Encontrará entre signos de mayor y menor el texto que el programador deberá cambiar de acuerdo a sus necesidades.

- El símbolo | será empleado cuando se deba elegir de entre varios casos.

5.2.1.1 Comentarios

Los comentarios agregados al código fuente en realidad no tienen significado para el compilador, más bien el compilador se encarga de eliminarlos. Su función es meramente de apoyo con la finalidad de tener notas informativas. Hay algunos compiladores que a partir de cierta estructura en los comentarios pueden generar otros archivos empleados como documentación, Java es un ejemplo.

Observe la estructura de los dos tipos de comentarios que puede utilizar en el código fuente:

```
// Este es un comentario
```

Figura 5-1 Comentario en una línea

La regla a observar es la siguiente:

- El texto inmediatamente después de los dos símbolos barra diagonal será considerado un comentario.

El siguiente tipo de comentario es empleado cuando se desea realizar un comentario que abarca una o varias líneas.

```
/*      Este
        es también
        un comentario
*/
```

Figura 5-2 Comentario en varias líneas

La regla a observar es la siguiente:

- El texto inmediatamente después del símbolo barra diagonal y un asterisco será considerado un comentario hasta que se encuentre un símbolo asterisco seguido de un símbolo barra diagonal.

5.2.1.2 Declaración de paquete

Observe la estructura de la declaración de ***paquete***:

```
[ paquete < ruta >; ]
```

Figura 5-3 Declaración de paquete

Las reglas a observar en la declaración de ***paquete*** son las siguientes:

- La declaración de ***paquete*** (ó carpeta) en la que se encuentra una *clase*, puede ser realizada como máximo una vez.

- Se entiende que en caso de no existir la declaración de ***paquete***, la carpeta base empleada será la ruta configurada por defecto.

- Se entiende que en caso de existir la declaración de ***paquete***, la carpeta base, será la ruta por defecto más la ruta del paquete.

- La "*ruta*" está dada por uno o más identificadores separados mediante el carácter punto (.).

- Un identificador es la concatenación de uno o más caracteres

- La declaración ***paquete*** no es obligatoria; sin embargo, en caso de existir deberá ser la primera instrucción que se realice en una *clase*.

- Al realizar el proceso de compilación serán cargadas, de forma automática, todas las *clases* con acceso público y de paquete encontradas en la misma carpeta que la usada por la *clase* en estudio.

5.2.1.3 Declaración importar

Observe la estructura de la declaración ***importa*r**:

```
[importar < ruta ><nombre de la clase | *> ];
```

Figura 5-4 Declaración importar

Las reglas a observar en la declaración ***importa*r** son las siguientes:

- La declaración ***importa*r** es el lugar donde se establecerá la *clase* necesaria para que sea posible la compilación de la *clase* en turno.

- Dado que en algunos casos puede ser necesario usar más de una *clase* para que sea posible la compilación de la *clase* en turno, se podrá realizar la declaración ***importar*** el número veces que el programador considere conveniente.

- La "*ruta*" está dada por uno o más identificadores separados mediante el carácter punto (.) e indica la ubicación física de la clase que se desea importar. La ruta definitiva donde se encuentra la clase estará dada por la ruta señalada en la configuración del compilador más la "ruta" señalada en la clase.

- Caso de que se desee importar todas las clases de un mismo paquete, bastará con colocar el símbolo asterisco (*) después de la "*ruta*".

- La declaración ***importar*** no es obligatoria.

- Al realizar el proceso de compilación serán cargadas, de forma automática, todas las clases importadas y por cada clase importada se cargarán las clases con acceso público y de paquete encontradas en la misma carpeta que la usada por la clase importada.

5.2.1.4 Declaración de interfaz

Observe la estructura de la declaración de ***interfaz***:

```
[accesor] interfaz  <nombre interfaz> [extiende <nombre interfaz
          que extiende>] {
      // Cuerpo de la interfaz
}
```

Figura 5-5 Declaración de interfaz

Las reglas a observar en la declaración de ***interfaz*** son las siguientes:

- El "*accesor*" de una ***interfaz*** permitirá determinar si el tipo de acceso ella es ***publico***, ***privado***, ***protegido*** o ***de paquete***. En caso de requerir el último tipo de acceso, bastará con dejar en blanco el espacio dedicado al "*accesor*".

- En un archivo de texto se podrá realizar como máximo una declaración de ***interfaz*** o de ***clase*** y no deberán encontrarse en el mismo documento.

- El apartado "*nombre interfaz*" deberá ser el mismo que el nombre asignado al archivo que el usuario esté redactando.

- El apartado ***extiende*** con su correspondiente "nombre interfaz que extiende" es opcional y será usado una vez como máximo en la declaración de ***interfaz*** o ***clase***.

- Una ***interfaz*** que extiende de otra (denominada padre) adquiere todos su métodos y atributos que tengan acceso ***publico*** y/o ***protegido***.

- "*nombre interfaz que extiende*" se referirá al nombre de la interfaz padre.

- El cuerpo de una interfaz comenzará con un símbolo de llave apertura y terminará con el símbolo llave de cierre.

5.2.1.5 Declaración de clase

Observe la estructura de la declaración de ***clase***:

```
[accesor] clase <nombre clase> extiende <nombre del clase>
[accesor] [ non_accesor]  interfaz  <nombre clase> [extiende <nombre
          clase que extiende>] [implementa <nombre interfaz que
          implementada 1>,…,< nombre  interfaz que implementada N>]{
      // Cuerpo de la clase
}
```

Figura 5-6 Declaración de clase

Las reglas a observar en la declaración de ***clase*** son las siguientes:

- El "*accesor*" de una ***clase*** permitirá determinar si el tipo de acceso a ella es ***publico***, ***privado***, ***protegido*** o ***de paquete***. En caso de requerir el último tipo de acceso, bastará con dejar en blanco el espacio dedicado al "*accesor*".

- El "*non_accesor*" de una ***clase*** permitirá determinar si se trata de tipo ***abstracto***. Caso de no existir se interpretará que la clase no es abstracta.

- En un archivo de texto se podrá realizar como máximo una declaración de ***clase*** y no deberán encontrarse en el mismo documento.

- El apartado "*nombre clase*" deberá ser el mismo que el nombre asignado al archivo que el programador esté redactando.

- El apartado ***extiende*** con su correspondiente "*nombre clase que extiende*" es opcional y será usado una vez como máximo en la declaración de ***clase***.

- "*nombre clase que extiende*" se referirá estrictamente al nombre de la clase que se extiende.

- La clase que extiende de otra (denominada ***padre***), adquiere también sus atributos y comportamientos pudiendo sobre-escribir el contenido del cuerpo de los métodos que fueron declarados como públicos ó protegidos.

- El apartado ***implementa*** con su correspondiente "nombre interfaz que implementa" es opcional y será usado una vez como máximo en la declaración de ***clase***.

- En caso de requerir más de un "*nombre interfaz que implementa*" en el apartado ***implementa***, éstos deberán ser separados por el carácter coma (,).

- "*nombre interfaz que implementa*" se referirá estrictamente a interfaces implementadas.

- El cuerpo de una clase comenzará con un símbolo de llave apertura y terminará con el símbolo llave de cierre.

5.2.1.6 Cuerpo de una interfaz

De forma análoga al cuerpo de una clase, la estructura del ***cuerpo de una interfaz*** es:

```
[<accesor> <tipo de dato> identificador = <valor>];
[<accesor> <tipo_devuelto> identificador([<tipo1 parametro1,… ,tipo N
          parametroN>]);]
```

Figura 5-7 Cuerpo de la interfaz

Las reglas a observar en el ***cuerpo de una interfaz*** son las siguientes:

- En caso de agregar declaraciones atributos, serán considerados automáticamente constantes.

- El "*accesor*" de un atributo o método permitirá determinar si el tipo de acceso a ella es ***publico***, ***protegido*** o ***de paquete***. En caso de requerir el último tipo de acceso, bastará con dejar en blanco el espacio dedicado al "*accesor*".

- Es posible agregar tantas declaraciones de atributos como necesite el usuario.

- El identificador empleado para nombrar el atributo no puede estar repetido.

- En caso de agregar declaraciones de métodos, el cuerpo de ellos deberá ser sustituido por el carácter punto y coma (;).

- Es posible agregar tantas declaraciones de métodos como necesite el usuario.

- *"tipo"*, se refiere al tipo dato recibido como parámetro, mientras que "*parametro"* se refiere al identificador usado como variable.

- Puede haber cero o tantos parámetros como se desee. Caso de haber más de un "*parametro*" serán separados por un símbolo de coma (,).

- El identificador empleado en el nombre de la declaración de método podrá ser el mismo sólo cuando los parámetros de cada declaración de método sean: diferentes en cantidad, en tipo o en el orden (y en este último caso, que además los tipos no sean iguales).

5.2.1.7 Cuerpo de una clase

Se sabe que el cuerpo de una clase está formado por atributos y métodos. Observe la estructura del ***cuerpo de una clase***:

```
[<accesor> <tipo_de_dato> identificador [= <valor>]];  // Atributo
[<accesor><non_accesor> <tipo_devuelto> identificador([<tipo1
       parametro1,… ,tipo N parametroN>])[; | {    // Método
       // Cuerpo del método
}]]
```

Figura 5-8 Cuerpo de una clase

Las reglas a observar en el ***cuerpo de una clase*** son las siguientes:

- Es posible agregar tantas declaraciones de atributos como necesite el usuario.

- El "*accesor*" de un atributo o método permitirá determinar si el tipo de acceso a ella es ***publico***, ***privado***, ***protegido*** o ***de paquete***. En caso de requerir el último tipo de acceso, bastará con dejar en blanco el espacio dedicado al "*accesor*".

- El "*non_accesor*" de un método permitirá determinar si se trata de tipo ***abstracto***.

- El identificador empleado para nombrar un atributo no puede ser repetido.

- En caso de agregar declaraciones de métodos, el cuerpo de ellos iniciará y terminará con un símbolo de llave, apertura en el primer caso y cierre en el segundo. La excepción se dará cuando el método declarado sea de tipo abstracto, en tal caso en lugar del cuerpo del método se escribirá el símbolo punto y coma (;).

- *"tipo_de_dato"*, se refiere al tipo de dato que almacenará el atributo declarado en ese momento. Aplica también para declaración de variables que se encuentren dentro de un método.

- *"tipo_devuelto"*, se refiere al tipo de dato que regresará un método.

- *"tipo"*, se refiere al tipo dato recibido como parámetro, mientras que "*parametro"* se refiere al identificador usado como variable.

- Puede haber cero o tantos parámetros como se desee. Caso de haber más de un "*parametro*" serán separados por un símbolo de coma (,).

- Es posible agregar tantas declaraciones de métodos como necesite el usuario.

- El identificador empleado en el nombre de la declaración de método podrá ser el mismo solo cuando los parámetros de cada declaración de método sean: diferentes en cantidad, en tipo o en el orden (y en este último caso, que además los tipos no sean iguales).

Hay un método especial denominado constructor que tiene la siguiente declaración:

```
[<accesor> identificador([<tipo1 parametro1,… ,tipo N
      parametroN>]){
      // Cuerpo del método
}]
```

Figura 5-9 Declaración método constructor

Las reglas que aplican sobre un método constructor son las siguientes:

- El "*accesor*" de un atributo o método permitirá determinar si el tipo de acceso ella es ***publico***, ***privado***, ***protegido*** o ***de paquete***. En caso de requerir el último tipo de acceso, bastará con dejar en blanco el espacio dedicado al "*accesor*".

- En caso de agregar declaraciones de métodos, el cuerpo de ellos iniciará y terminará con un símbolo de llave, apertura en el primer caso y cierre en el segundo.

- *"tipo"*, se refiere al tipo dato recibido como parámetro, mientras que "*parametro"* se refiere al identificador usado como variable.

- Puede haber cero o tantos parámetros como se desee. Caso de haber más de un "*parametro*" serán separados por un símbolo de coma (,).

- Es posible agregar tantas declaraciones de métodos constructores como necesite el usuario.

- El identificador empleado en el nombre de la declaración de método constructor podrá ser el mismo solo cuando los parámetros de cada declaración de método sean: diferentes en cantidad, en tipo o en el orden (y en este último caso, que además los tipos no son iguales).

- La declaración de métodos constructores es opcional. De realizarse llevará el nombre de la clase en turno.

5.2.1.8 Cuerpo de un método

El cuerpo de un método puede ser encontrado en dos condiciones:

- Sin contener expresión alguna.

- Conteniendo una o más expresiones.

Una expresión puede ser una sentencia de código o un bloque de sentencias reunidas en una estructura de control. Por comodidad, serán clasificadas en: Sentencias de asignación, Sentencias de ejecución de métodos y Estructuras de control.

5.2.1.8.1 Sentencias de asignación

Al realizar una asignación se pueden presentar tres situaciones: Que en la asignación se genere una nueva instancia, que en la asignación sea asignado el resultado de una llamada a función y la tercera es que se asigne un valor numérico, de texto, carácter, boleano o un identificador que representa a una variable de algún tipo.

5.2.1.8.1.1 Sentencia de asignación vía nueva instancia

A continuación se muestra la estructura de este tipo de asignación:

```
[identificador1 = nuevaInstancia identificador2 ([<parametro1,… ,
      parametroN>]);]
```

Figura 5-10 Sentencia de asignación de una nueva instancia

Las reglas a observar son las siguientes:

- En el cuerpo de un método pueden aparecer cero o tantas sentencias de asignación vía nueva instancia como sean necesarias.

- identificador$_1$ corresponde al nombre de la variable que será modificada.

- "*nuevaInstancia*" es la palabra reservada empleada para indicar que se creará una nueva instancia.

- Después de la palabra reservada, identificador$_2$ corresponde al nombre del método constructor a usar. A su vez, indica el tipo de dato que se está asignando.

- Podrá presentarse el caso en que no sea pasado ningún parámetro.

- "parametro" será empleado para pasar un valor al método en turno. Podrán existir tantos parámetros como hayan sido declarados en el método.

- La sentencia terminará con un símbolo de punto y coma (;).

5.2.1.8.1.2 Sentencia de asignación simple

A continuación se muestra la estructura de una sentencia de asignación simple:

```
[identificador1 = < identificador2 | texto | numero | boleano |
      carácter | numero + numero | numero – numero |
      numero * numero | numero / numero | numero % numero >; ]
```

Figura 5-11 Sentencia de asignación simple

Las reglas a observar son las siguientes:

- En el cuerpo de un método pueden aparecer cero o tantas sentencias de asignación simple como sean necesarias.

- identificador**1** corresponde al nombre de la variable que será modificada.

- identificador**2** corresponde al nombre identificador que proporcionará su valor a identificador$_1$.

- "texto" es una secuencia de caracteres que se asigna a la variable a modificar. Comienza y finaliza con un símbolo doble comilla (**"**).

- "carácter" es un carácter que se asigna a la variable a modificar. Comienza y finaliza con un símbolo comilla simple (**'**).

- "numero" está dado por un número entero o real que puede estar signado.

- "boleano" está dado por la palabra reservada "***verdadero***" o "***falso***".

- Podrá encontrar dos operadores numéricos separados por un operador aritmético de suma.

- Podrá encontrar dos operadores numéricos separados por un operador aritmético de resta.

- Podrá encontrar dos operadores numéricos separados por un operador aritmético de multiplicación.

- Podrá encontrar dos operadores numéricos separados por un operador aritmético de división.

- Podrá encontrar dos operadores numéricos separados por un símbolo de porcentaje que representará el residuo de una división.

- La sentencia terminará con un símbolo de punto y coma (;).

5.2.1.8.1.3 Sentencia de asignación vía un método que devuelve un dato

A continuación se muestra la estructura de una sentencia de asignación de método que devuelve un valor:

```
[identificador1 = identificador2 ([<parametro1,… ,
      parametroN>]);] // Opción 1
[identificador3 = identificador4 .identificador5 ([<parametro1,… ,
      parametroN>]);] // Opción 2
```

Figura 5-12 Sentencia de asignación por medio de un método que devuelve un valor

Las reglas a observar son las siguientes:

- En el cuerpo de un método pueden aparecer cero o tantas sentencias, de asignación vía método que devuelve un valor, como sean necesarias.

- identificador$_1$ corresponde al nombre de la variable que será modificada.

- identificador$_2$ corresponde al nombre del método que devolverá un dato que será asignado a identificador$_1$. Aplica para métodos encontrados en la misma clase.

- Podrá presentarse el caso en que no sea pasado ningún parámetro.

- “parametro" será empleado para pasar un valor al método en turno. Podrán existir tantos parámetros como hayan sido declarados en el método.

- La sentencia terminará con un símbolo de punto y coma (;).

- Para llamar un método de otra clase, se usa una variable del tipo de objeto deseado mostrada en el cuadro anterior como identificador$_4$, se accede al método identificador$_5$ a través del símbolo punto (.). El método deberá estar acompañado de los parámetros correspondientes.

5.2.1.8.1.4 Sentencia de retorno de dato desde un método

A continuación se muestra la estructura de una sentencia de retorno de un método que devuelve un valor:

```
[ regresar DATO ;]
```

Figura 5-13 Instrucción para regresar un valor desde el interior de un método

Las reglas aplicadas son:

- La sentencia podrá aparecer cero o una vez en el cuerpo del método.

- En caso de aparecer, la sentencia será la última que aparezca.

- En caso de aparecer, no deberá estar contenida en el cuerpo de alguna expresión de control.

- "DATO", podrá aparecer un texto, un carácter, un número, un boleano o un identificador;

5.2.1.8.2 Sentencia de impresión

En caso de que se desee, mandar un mensaje de impresión al código de salida se usará la siguiente estructura:

```
[ imprimir( DATO );]
```

Figura 5-14 Sentencia de impresión

Las reglas aplicadas son:

- La sentencia podrá aparecer cero o tantas veces como sea necesario en el cuerpo de un método.
- Entre paréntesis será colocado el identificador que será señalado como de impresión en el código de salida.
- Se finalizará la sentencia con un símbolo de punto y coma (;).
- "DATO", podrá aparecer un texto, un carácter, un número, un boleano o un identificador;

5.2.1.8.3 Sentencia de ejecución de método

A continuación se muestra la estructura de una sentencia de ejecución de método:

```
[<Identificador_Clase>.] identificador([<parametro1,… , parametroN>]);]
```

Figura 5-15 Sentencia de ejecución de método

Las reglas a observar en una sentencia de ejecución de método son las siguientes:

- En el cuerpo de un método pueden aparecer cero o tantas sentencias de ejecución de método como sea necesario.

- "*Identificador_Clase*" es opcional y deberá aparecer cuando el método llamado pertenezca a otra clase, de la cual se declaró un atributo o variable. Siempre que aparezca, deberá ser seguido de un símbolo de punto (.).

- Podrá presentarse el caso en que no sea pasado ningún parámetro.

- "parametro" será empleado para pasar un valor al método en turno. Podrán existir tantos parámetros como hayan sido declarados en el método.

- La sentencia terminará con un símbolo de punto y coma (;).

5.2.1.8.4 Estructuras de Control

Serán desarrolladas estructuras de control de flujo y estructuras de control repetitivas.

5.2.1.8.4.1 Estructura de control de flujo

A continuación se muestra la estructura de una sentencia de control de flujo:

```
si ( exp_boleana ) cuerpo_expresiones1 [ otro cuerpo_expresiones2 ]
```

Figura 5-16 Estructura de control de flujo

Las reglas a observar en una sentencia de control de flujo son las siguientes:

- Se usará la palabra reservada **si** para indicar que se está declarando una estructura de control de flujo.

- Entre paréntesis estará una expresión que podrá ser evaluada como verdadera o falsa. Caso de ser verdadera se ingresará a cuerpo_expresiones$_1$.

- "cuerpo_expresiones" podrá funcionar de dos formas:

    1. A la declaración de la estructura de flujo le sigue sólo una expresión.

    2. A la declaración de la estructura le sigue un grupo de expresiones. Las expresiones que le sigan y que pueden ir de cero a N, deberán estar contenidas entre un símbolo de llave de apertura y uno de llave de cierre.

- La palabra reservada **otro** es opcional y en caso de existir permitirá acceder a cuerpo_expresiones$_2$.

5.2.1.8.4.2 Estructura de control repetitiva

A continuación se muestra la estructura de una sentencia de control repetitiva:

```
mientras ( exp_boleana ) cuerpo_expresiones1
```

Figura 5-17 Estructura de control repetitiva

Las reglas a observar en una sentencia de control repetitiva son las siguientes:

- Se usará la palabra reservada **mientras** para indicar que se está declarando una estructura de control repetitiva.

- Entre paréntesis estará una expresión que podrá ser evaluada como verdadera o falsa. Caso de ser verdadera se ingresará a cuerpo_expresiones$_1$.

- "cuerpo_expresiones" podrá funcionar de dos formas:

    1. A la declaración de la estructura de flujo le sigue sólo una expresión.

    2. A la declaración de la estructura le sigue un grupo de expresiones. Las expresiones que le sigan y que pueden ir de cero a N, deberán estar contenidas entre un símbolo de llave de apertura y uno de llave de cierre.

5.2.1.8.4.3 Estructura de una expresión boleana

La estará dada por alguna de las siguientes opciones:

```
IDENTIFICADOR OP_IGUALDAD NUMERO | IDENTIFICADOR | ! IDENTIFICADOR
          | IDENTIFICADOR OP_LOGICO IDENTIFICADOR
```

Figura 5-18 Estructura de una expresión boleana

Las reglas que aplicarán son las siguientes:

- Caso de presentarse: IDENTIFICADOR OP_IGUALDAD NUMERO, entonces OP_IGUALDAD deberá corresponder a los símbolos <, >, >=, <=, ==.

- Caso de presentarse: IDENTIFICADOR OP_LOGICO IDENTIFICADOR, entonces OP_ LOGICO deberá corresponder a los símbolos |, ||, & ó &&, que representan las operaciones OR y AND.

- Negación, será representada por el símbolo de admiración (!).

5.2.1.9 Tipos de datos

Se pueden clasificar los tipos de datos en dos bloques:

- Primitivos. Los rangos de valores permitidos corresponderán a su análogo en Java. Los valores primitivos manejados serán:

    a) entero.

    b) largo.

    c) doble.

    d) carácter.

    e) boleano.

- Por referencia. Las clases desarrolladas por el usuario, se agrega a este apartado la clase T*exto* usada para almacenar cadenas de caracteres.

5.2.1.10 Identificador

Se entiende por *identificador* a la concatenación de uno o más caracteres tomados del a*lfabeto* del *lenguaje de Java en Español*. Las condiciones que aplican en la formación de identificadores son las siguientes:

- Los identificadores inician con una letra de la a – z, una letra de la A –Z, el carácter guión bajo (_), el carácter signo de pesos ($).

- Los caracteres del *alfabeto* del *lenguaje de Java en Español* que no pueden ser empleados en la creación de un identificador son: El carácter punto y coma (;), el carácter punto (.), el carácter comillas dobles (“) ni el carácter comilla simple (‘).

- El límite máximo de caracteres concatenados será de 2^{31}-1.

- No podrá ser usada ninguna palabra reservada como identificador, aún cuando dichas palabras se apeguen a los puntos antes mencionados.

5.2.2 Código fuente empleando las reglas sintácticas

De acuerdo al conjunto de reglas propuestas, el código presentado en la figura 5-19 debería ser válido. Aunque sintácticamente el *código fuente* presentado es correcto, es necesario señalar que tiene algunas inconsistencias que son producto de la necesidad de aplicar reglas semánticas, a continuación se presentan algunos ejemplos de las inconsistencias:

- Algunas variables de un tipo se asignan a variables de tipo diferente.

- Algunas variables se usan sin haber sido declaradas.

- Algunas expresiones boleanas no son correctas si se considera el tipo propuesto e inexistente en algunos casos.

- Se usan palabras reservadas como identificadores de variables.

- Se asignan instancias de un tipo diferente al tipo que recibe la variable a modificar.

- El constructor de la *clase* posee un nombre diferente al de la *clase* ejemplo.

- La declaración del método *metodo2* indica que dicho deberá regresar un dato de tipo entero.

```
// Este es un ejemplo de una clase generada con el lenguaje propuesto
paquete a.b;
importar a.c.UnaClaseMas;
publico abstracto clase MiClase2{
        privado entero Texto;
        privado entero var2 = 20;
        publico MiOtraClase objeto1;
        privado texto var3 = "mi texto";
        publico MiClase(){
        }
        publico entero metodo2(entero param1, entero param2){
                doble var2 = 4.5;
                Texto var3 = "algun texto";
                MiNuevoTipo mnt = nuevaInstancia MiNuevoTipo();
                si ( var11 & var31){
                        caracter var11 = 'B';
                        caracter var13 = 'C';
                        entero var16 = 50;
                        MiNuevoTipo2 mnt2 = nuevaInstancia MiNuevoTipo5();
                }otro{
                        entero var21 = 500;
                        MiNuevoTipo3 mnt3 = nuevaInstancia MiNuevoTipo3();
                }
                mientras ( algo < 5 ){
                        boleano var1 = falso;
                        MiNuevoTipo4 mnt4 = nuevaInstancia MiNuevoTipo4();
                        si ( var15 || var40 )
                                MiSegundoTipo var20 = 4.5;
                }

        }
}
```

Figura 5-19 Ejemplo de código fuente válido, considerando el análisis sintáctico

5.3 Reglas semánticas

1. **Declaración importar.** Se verificará que el tipo de acceso de la clase importada sea público.

2. **Declaración de clase**. Con respecto a la declaración de la clase están las siguientes reglas:

- Si una clase es declarada como abstracta, entonces no se podrán crear instancias de ella. Además, las clases que hereden de una clase abstracta deberán implementar los métodos que hayan sido declarados como abstractos.

- Si una clase en su declaración implementa una interfaz, entonces también deberá implementar todos los métodos declarados en dicha interfaz.

3. **Declaración de método.** Si un método en una clase es declarado como abstracto, entonces se deberá verificar que la *clase* un turno haya sido declarada de tipo abstracto.

4. **Uso de un método.** Se verificará que el tipo, orden y cantidad de parámetros concuerde con la declaración del método respectivo.

5. **Método constructor.** Se deberá cumplir que cuando sea declarado un método constructor, el nombre del método sea el mismo que el de la clase en turno.

6. **Sobre-escritura.** La sobre-escritura de métodos se presenta cuando una ***clase A*** modifica el código fuente de un método que heredó de una ***clase B*** o ***interfaz C***.

7. **Sobrecarga.** La sobrecarga de métodos se presenta cuando una clase presenta declaraciones de métodos con el mismo nombre. Sin embargo, es necesario que sea diferente el número de parámetros ó el tipo de ellos. En caso de que el número de parámetros sea el mismo y que existan diferentes tipos de datos, el orden entre ellos deberá ser diferente.

8. **Variables locales y atributos.** Verificará que una variable antes de usarse se encuentre declarada.

9. **Asignación simple.** Verificará que el tipo de datos que se asignará a una variable es del mismo tipo que el que posee la variable a modificar.

10. **Nueva instancia.** Cuando se cree una nueva instancia, verificará que el constructor exista en la clase del tipo deseado y que los parámetros concuerden con el método constructor declarado en la clase de la cual se desea realizar una nueva instancia, además de verificar que el tipo que representa se asigne a una variable del mismo tipo o una de jerarquía mayor en el caso de herencia.

11. **Alcance de una variable.** Verificará que cuando se use una variable, ésta tenga vigencia dentro del bloque de código en el cual está siendo usada. Así un atributo tendrá vigencia a lo largo de la clase, mientras que una variable declarada dentro de un método tendrá vigencia sólo en el método en que fue declarado. De la misma forma, si una variable es declarada en el cuerpo de una expresión de control de flujo o repetitiva, sólo tendrá vigencia dentro del cuerpo de dicha expresión.

12. **Acceso público.** Indica que una clase, atributo o método pueden ser accedidos por otra clase.

13. **Acceso privado.** Indica que el atributo o método, con dicho tipo de acceso, sólo pueden ser accedidos dentro de la misma clase en que fueron declarados.

14. **Acceso protegido.** Indica que el atributo o método, con dicho tipo de acceso, puede ser accedido desde la misma clase o desde clases que hereden de ella.

15. **Acceso de paquete.** Indica que el atributo o método de una clase puede ser accedido por las clases del mismo paquete. En el caso de una clase, indica que esa clase sólo es visible para las clases que se encuentran en el mismo paquete o carpeta.

16. **Acceso a un método desde un objeto.** Cuando se accede a un método desde un objeto, se deberá verificar que el tipo de acceso del método empleado sea público

17. **Expresiones boleanas.** Verificar que los tipos de datos proporcionados, en la comprobación boleana de una expresión ***si*** o una expresión ***mientras***, puedan ser evaluados a ***falso*** ó ***verdadero***. Los tipos empleados en los identificadores sólo podrán ser numéricos y boleanos. Además de aplicar las siguientes reglas:

    - En caso de que la expresión boleana sea sólo un identificador, éste deberá ser de tipo boleano.

    - En caso de emplear operadores lógicos, los identificadores dentro de la expresión boleana deberán ser de tipo boleano.

    - En caso de emplear operadores aritméticos, los identificadores dentro de una expresión deberán ser de tipo numérico.

18. **Identificadores.** Una palabra reservada no puede ser empleada como identificador.

19. **Retorno de método.** Verificará que cuando la declaración de un método indique que regresará un dato, éste método en verdad regrese un dato.

20. **Asignación vía método.** Se verificará que el tipo de dato devuelto por un método corresponda al tipo que recibe la variable que se desea modificar.

21. **Tipo de retorno de datos desde un método.** Verificará que cuando se regrese un dato desde un método, este corresponda al tipo de dato declarado o a uno de jerarquía mayor.

5.3.1 Código fuente empleando las reglas semánticas

De acuerdo al conjunto de reglas ahora establecido, el código mostrado en la figura 5-20 sería un código correcto sintáctica y semánticamente; es decir un código que no presentaría mensajes de error al usuario y que por tanto tendría que generar el correspondiente *código objeto* por el compilador:

```
// Este es un ejemplo de una clase generada con el lenguaje propuesto
paquete a.b;

importar a.c.UnaClaseMas;
publico abstracto clase MiClase4{
        privado entero var2 = 20;
        publico UnaClaseMas objeto1;
        privado texto var3 = "mi texto";
        publico MiClase4(){
                si ( var2 > 15)
                        var3 = "otro texto";
        }
        publico MiClase4(entero e){
                var2 = e;
                si ( var2 > 10)
                        var3 = "Un nuevo texto";
        }
        publico entero metodo2(entero param1, entero param2){
                doble var4 = 4.5;
                UnaClaseMas mnt = nuevaInstancia UnaClaseMas();
                si ( var2 < var4)
                        var3 = "Un texto más";
                boleano var5 = falso;
                si (! var5 ){
                        caracter var6 = 'B';
                }otro{
                        var2 = 15;
                }
                regresa var2;
        }
}
```

Figura 5-20 Ejemplo de código fuente válido, considerando el análisis semántico

6 Compilador en español

Java, presenta un ejemplo real, de un lenguaje de programación orientado a objetos. Cubierto el capítulo 2 se sabe que el código de salida del proceso de compilación es determinada por el diseñador del compilador, como no hay una restricción en el producto a obtener del proceso de compilación del compilador a desarrollar, se tomó la decisión de realizar como salida un archivo con extensión "*java*" que a su vez pueda ser compilado por Java. El capítulo 4 se centra en el conocimiento del lenguaje de programación Java, el proceso de compilación del código fuente hecho en dicho lenguaje y la forma en que éste es ejecutado. Se elige el lenguaje Java entre otras cosas, por que es un lenguaje orientado a objetos, simple, robusto y con una gran aceptación en el mundo.

6.1 Plan de trabajo

En este momento, se conocen con precisión los detalles del ***lenguaje Java en Español***, que será el lenguaje con el cual se realice el código fuente que recibirá como entrada el compilador a desarrollar. Se sabe además que la salida pretendida es un código en *lenguaje Java* que podrá ser compilado por el compilador Java.

En el entendido de que la construcción de cualquier sistema implica cierta complejidad, antes de iniciar la construcción de un sistema generalmente se realiza un diseño o modelo, donde se describen las piezas u objetos que se agruparán en componentes que se emplearán para construir el sistema completo. La popular expresión "Divide y vencerás", es bien aplicada en el proceso de diseño y desarrollo de software.

Formalmente, el diseño no es otra cosa que un proceso que busca definir, de forma abstracta, un producto con suficientes detalles que permitan su realización física; dicho proceso puede aplicarse usando distintas metodologías y principios asentados en documentos que modelan y/o describen las diferentes partes en las que podría ser dividido

un sistema y la integración de esas partes[37]. El diseño es empleado como un mecanismo de apoyo en la construcción de un sistema; es decir, es empleado en la obtención de una solución, y como tal el diseño no es la solución.

Para su diseño y construcción el compilador será desarrollado en dos partes: La de *análisis* y la de *síntesis*. Como se comentó en el capítulo 2, durante la parte de *análisis* se realizan las correspondientes comprobaciones sintácticas, semánticas, y se deja preparado el escenario para que en la parte de síntesis se pueda realizar la generación y optimización del *código objeto*. Bajo el citado esquema de diseño y desarrollo, el lector podría optar desarrollar por su cuenta, por ejemplo:

- Directamente un código binario relocalizable.
- Un código que pudiera ser leído por una máquina virtual de un dispositivo portátil.
- Un código que pudiera ser leído en alguna máquina virtual en algún sistema operativo.
- Un código objeto en algún otro lenguaje, diferente al usado por Java.

A lo largo del capítulo se buscará plasmar las diferentes fases en las que se divide un compilador, según se pudo ver en el capítulo 2. En caso de ser necesario, se describirá la forma en que interactúa la fase estudiada con su entorno.

6.1.1 Integración Lenguaje Java Español – Compilador en Español

Se ha delimitado ya el lenguaje de programación y con ello han sido definidas las reglas que deberá hacer cumplir el compilador desarrollado.

---

[37] Para una mayor descripción, dirigirse al texto INGENIERÍA DEL SOFTWARE, UN ENFOQUE PRÁCTICO, Pressman Roger S., quinta ed., edit. McGraw Hill describe algunas metodologías.

La figura siguiente muestra el flujo que seguirá el proceso de compilación.

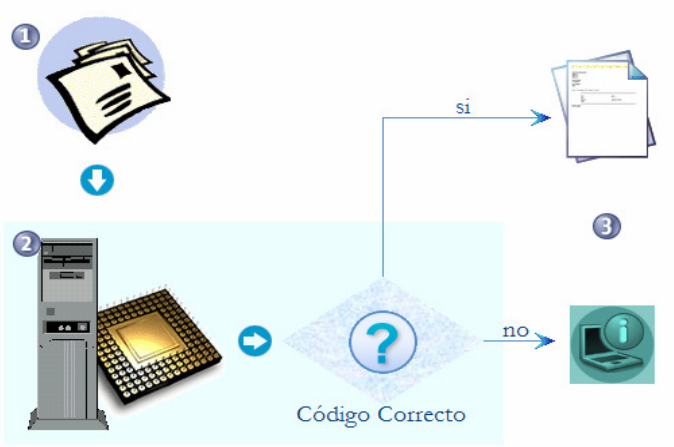


Figura 6-1 Flujo que sigue el proceso de compilación

A continuación se describe el flujo presentado en figura 6-1:

1. El usuario se encarga de capturar los archivos del *código fuente*, que necesita compilar, los cuales deberán seguir las reglas marcadas en el *lenguaje Java en Español*.

   Además de capturar los archivos de texto con el *código fuente*, es el usuario del *compilador en Español* el que asignará un nombre a cada archivo a compilar. Sin embargo, existen algunas condiciones básicas:

   - La extensión del archivo leído será ***jave*.**
   - El número de caracteres empleados en el nombre del archivo debe ser permitido por el sistema operativo del usuario.
   - En caso de necesitar compilar más de un archivo, deberá nombrar sus archivos de forma tal que no se presenten ambigüedades. Por ejemplo, dos clases con el mismo nombre.

2. El usuario hace uso del *compilador en Español*, sistema a desarrollar, una vez que cuenta con el código fuente dividido en ***clases***. El sistema se encargará de verificar que las ***clases*** capturadas cumplen las reglas establecidas en el *lenguaje Java en Español*.

   Es en éste punto en el cual el usuario le indicará al Compilador en Español que **clase** será compilada. Se pueden presentar dos situaciones:

   - Se compile una sola ***clase***.

   - Se desee compilar una ***clase*** que necesita de otras ***clases*** para poder ser compilada.

   En ambas situaciones será el compilador el que resuelva lo que deberá hacer.

3. Luego de procesar el *código fuente*, el compilador entregará uno de los siguientes resultados:

   - Un informe que permita al usuario conocer las correcciones que debe realizar sobre el *código fuente* original, a fin de ajustarse a las reglas del *lenguaje Java en Español*. Hechas las correcciones el usuario debe realizar el proceso de compilación.

   - El sistema devolverá un archivo con extensión ***java*** por cada una de las ***clases*** que hayan sido compiladas. Los archivos podrán ser usados por el compilador Java sin necesidad de adecuaciones, excepto en el caso de que deseara emplear el método estático **main** de Java.

Es obvio, que el escenario ideal es aquel en el cual el resultado obtenido es: un conjunto de 1 o más archivos con extensión ***java***, producto de una compilación exitosa; es decir, sin errores.

6.2 Diseño de Compilador en Español

Según se comentó en el apartado anterior; para su diseño y construcción el compilador será desarrollado en dos partes: La de *análisis* y la de *síntesis*. Durante la parte de *análisis* se realizan las correspondientes comprobaciones sintácticas, semánticas, y se deja preparado el escenario para que en la parte de síntesis se pueda realizar la generación y optimización del *código objeto.* Bajo el citado esquema de diseño y desarrollo, el lector podría optar elegir desarrollar, en la parte de síntesis: directamente un código binario relocalizable, un código que pudiera ser leído por una máquina virtual de un dispositivo portátil o de alguna máquina virtual en algún sistema operativo, incluso la obtención de código en algún otro lenguaje.

En las secciones siguientes, se buscará plasmar en la medida de lo posible las diferentes fases en las que se divide un compilador, según se pudo ver en el capítulo 2.

Se puede ver que la cantidad de material acumulado hasta ahora es muy grande. En base a la experiencia en el desarrollo de software adquirida a lo largo de los años de trabajo, se puede afirmar que el diseño y el desarrollo del "*Compilador en Español*" requerirá una inversión de una cantidad considerable de tiempo. Con la finalidad de concluir con el desarrollo en un tiempo razonable y no quedar inmerso en un ciclo prolongado de desarrollo, se tomó la decisión de presentar sólo el diseño y desarrollo de software que se considerara más trascendente. En algunos casos, por considerarse importante se presentará el diseño y el desarrollo, o una buena parte de él, mientras que en otros casos se prescindirá de él, no hay que perder de vista que la POO es amplia y que Java al tener ya varios años en desarrollo en este momento es también muy grande.

Para llevar a buen término el diseño y desarrollo de la parte de análisis y síntesis del *compilador en Español* el proceso de compilación será realizado en etapas ó divisiones denominadas *pasadas*, en las cuales se realizan diferentes procesos, se analizan fases y se obtienen diferentes productos. A lo largo cada *pasada* se presentará la descripción de los objetivos buscados.

6.3 Primera pasada

Se debe tener presente que la estructura de cada archivo representará una *clase*, resulta natural suponer que la primera pasada centrará su atención en el análisis de cada una de las *clases* a compilar de forma individual; así pues, los objetivos a cubrir en esta parte serán las siguientes:

- Diseñar y construir el módulo de control y manejo de los ***Informes*** que serán presentados al usuario. La razón de incluir el presente módulo como parte de la primera *pasada* está dado por el hecho de que cualquiera de las fases, en las que sea dividido el compilador, deberá reportar al usuario los errores encontrados.

- Diseñar y construir el módulo de análisis léxico, el cual se encargará de leer cada archivo fuente y entregar los componentes léxicos según se le vaya solicitando, además de realizar los procedimientos necesarios para liberar el archivo fuente luego de terminar su lectura.

- Diseñar y construir un módulo de análisis sintáctico que verifique la declaración de paquete, las declaraciones de archivos importados, la declaración de *clase*. De forma paralela, deberá recopilar información referente a cada apartado y almacenarla en la tabla correspondiente. Un producto más del presente módulo será realizar la ubicación atributos y métodos, realizar la separación de los mismos colocándolos en una tabla la cual deberá poseer, en el caso de los métodos, el *código fuente* correspondiente al cuerpo y que será examinado en una segunda *pasada*. Se considera conveniente ya no manejar directamente al archivo de texto que representa el *código fuente* para evaluar el contenido de los métodos pues se espera que eso sea bastante complicado cuando se trate de más de una *clase*, en su lugar se creará un componente que permita el manejo de las líneas de código almacenadas en memoria.

- Los tres puntos anteriores aplican perfectamente en el caso de examinar una sola *clase*. En este punto se puede comenzar a realizar ahora la verificación del código asentado en cada método; sin embargo, se juzga conveniente que al analizar el *código fuente* también se fuera construyendo la tabla que almacenara las variables declaradas y su respectiva información, la razón es que dicha información será necesaria durante el análisis semántico. Para realizar el análisis semántico se necesitaría tener recopilada información acerca de las variables, tipos y asignaciones empleadas en los métodos no sólo de las variables locales sino también de las variables y métodos procedentes de otras *clases*; así que, con el propósito de contar con la información necesaria para el análisis semántico, en esta primera *pasada* se incluirá el módulo que realiza el proceso descrito en los tres primeros puntos para cada *clase* involucrada.

- Es importante que el resultado de la primera *pasada* esté disponible y sirva de entrada de la segunda *pasada* con el fin de que ésta última cumpla con su objetivo.

6.3.1 Control y manejo de errores

Como puede observarse en el capítulo 2, las fases que típicas en las que puede descomponerse un compilador están directamente relacionadas con el "*Controlador de errores*" el cual se encarga de almacenar y presentar al usuario un informe de todos los problemas encontrados en las diferentes fases analizadas.

6.3.1.1 Diseño del controlador de errores

Una vez conocido el objetivo del controlador de errores, se presenta en la figura 6-2 lo que podría ser el diseño del módulo dedicado al control y manejo de errores del *compilador en español*.

La figura 6-2 permite observar que el objeto ***Informe*** es notificado cuando se genera un evento indicando un error en el analizador léxico, sintáctico, semántico o en la parte de síntesis. ***Informe*** se encarga de almacenar los mensajes de error recibidos y los entrega cuando el compilador los solicita.

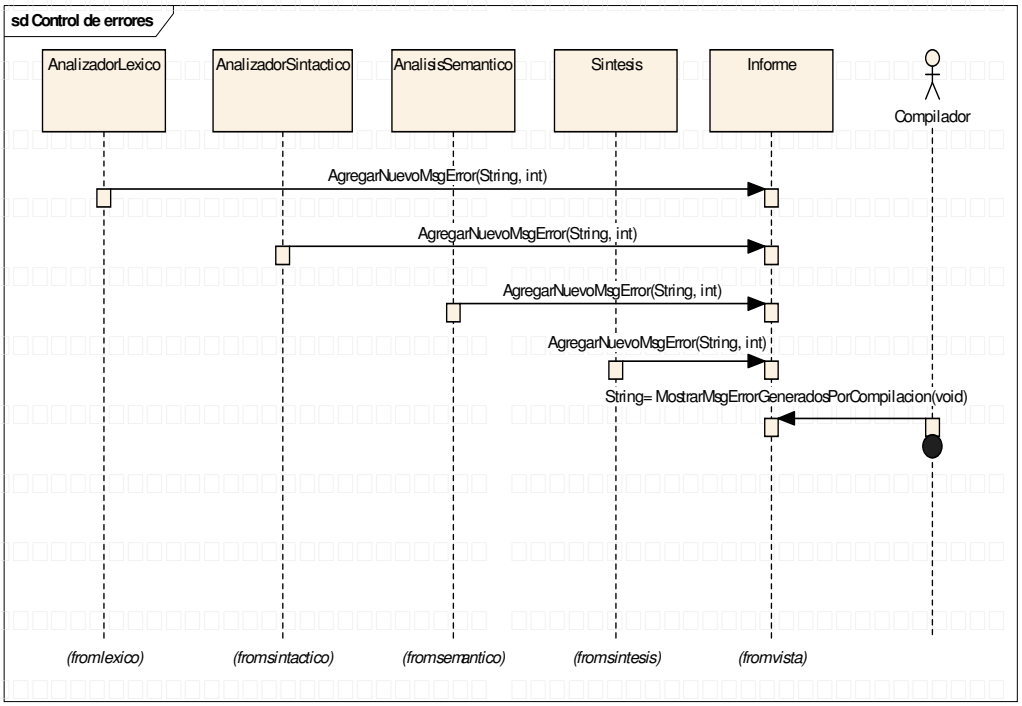


Figura 6-2 Diagrama que esquematiza como interactúa el controlador de errores con las diferentes partes de un compilador

Sin duda, el lector anticipa en este momento, que si posterior al proceso de compilación, ***Informe*** no contiene mensajes de error, se procederá a la generación del correspondiente archivo ***java***.

6.3.2 Análisis léxico

Como se mencionó, en el capítulo 2, la presente etapa se centra en la obtención de lexemas que sean significativos en el lenguaje desarrollado. Observe el siguiente fragmento de código, con él se busca ser más descriptivo acerca del proceso realizado durante el análisis léxico:

entero var= 5;

entero otraVar= var;

Puede ver, en la tabla 6-1, que el número de lexemas que pueden generarse a partir de las dos líneas de código leídas podría ser muy grande.

e	**entero v**	**var**	**otraVar**
en	**entero va**	**var=**	**otraVar=**
ent	**entero var**	**var= 5**	**otraVar= v**
ente	**entero var=**	**var= 5;**	**otraVar= va**
enter	**=**	**;**	**otraVar= var**
entero	**…**	**…**	**otraVar= var;**

Tabla 6-1 Ejemplos de lexemas que se podrían generar a a partir de las líneas de código señaladas

Es obvio que no todos los lexemas, mostrados en la tabla 6-1, resultarán útiles. Es necesario acotar el número de lexemas; así que, se excluyen aquellos que no sean significativos para el lenguaje desarrollado.

Una manera de generar lexemas, de hecho la habitual, consiste en formarlos a partir de separadores previamente definidos, por ejemplo: Espacios, signos de puntuación, de asignación, de comparación, etc. Así pues; la siguiente lista de lexemas, leída de arriba a abajo y de izquierda a derecha, pudiera ser considerada más adecuada.

entero	**5**	**otraVar**	**;**
var	**;**	**=**	
=	**entero**	**var**	

Tabla 6-2 Lexemas generados cuando se tienen reglas bien definidas que determinan el lugar donde inicia y donde termina un lexema

6.3.2.1 Diseño de un analizador léxico

Se ha descrito ya la fase de análisis léxico. A continuación se presenta el diagrama de interacción que muestra la forma en que podría ser desarrollado un sistema que automatice la fase de análisis léxico.

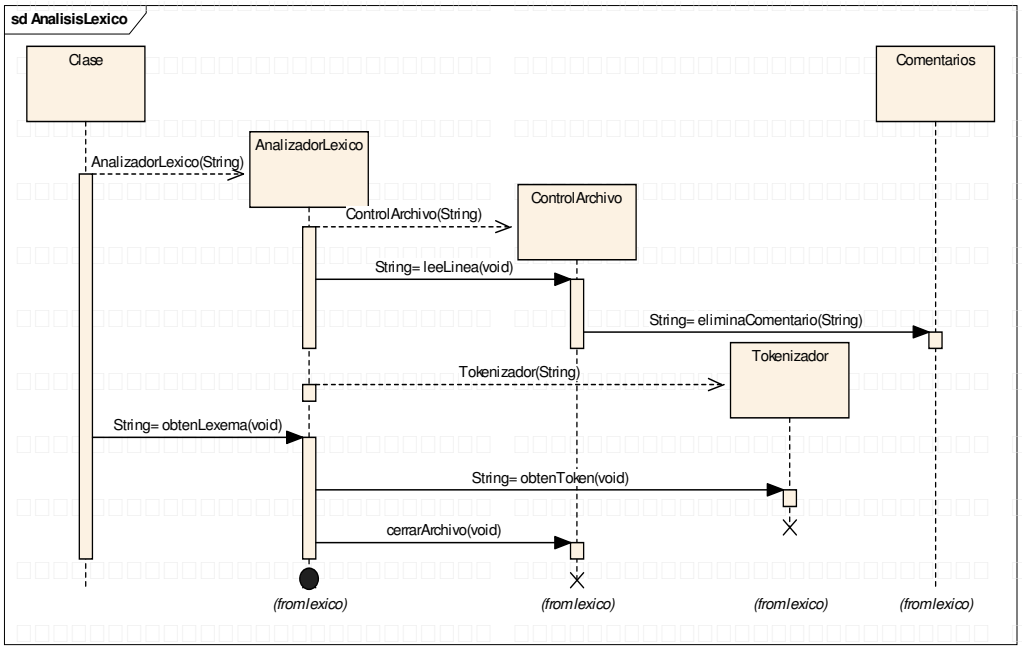


Figura 6-3 Diagrama que esquematiza el flujo que sigue el proceso de análisis léxico

El análisis sintáctico necesita de un buen proceso de análisis léxico que lo alimente. En el diagrama anterior se puede observar la secuencia a seguir cuando se desea realizar el análisis léxico de una ***clase***, a continuación la descripción:

1. En la lógica de negocio del sistema, en el apartado dirigido al análisis sintáctico del objeto *Clase* existe un componente denominado ***AnalizadorLéxico*** al que se recurre cada que es necesario obtener un lexema y que funciona como controlador del módulo de análisis léxico.

2. Antes de poder obtener un lexema es necesario crear una instancia del ***AnalizadorLexico***, la cual se encarga de controlar todo el módulo de análisis léxico.

3. *ControlArchivo* recibe el nombre del archivo a estudiar. En caso de no encontrar el archivo, se encarga de avisar al controlador (***AnalizadorLexico***) para que éste se comunique con la clase ***Informe*** y sea guardado el mensaje correspondiente.

4. En caso de haber logrado realizar la instancia del objeto *ControlArchivo*, el controlador solicita leer una línea del archivo en estudio.

5. ***ControlArchivo*** lee una línea y la envía al objeto ***Comentarios*** que se encarga de regresar la línea de texto sin comentarios, claro que sólo sucedería en el caso de que la cadena hubiera presentado o representado un comentario, en caso contrario regresaría la misma cadena. La cadena regresada a *ControlArchivo* en caso de que la cadena recibida por el objeto *Comentarios* fuera en su totalidad un comentario, sería una cadena con longitud cero.

6. Una vez que se tiene una línea de texto sin comentarios, el controlador la envía al objeto ***Tokenizador***, el cual se encarga de seccionar la línea de texto en lexemas. Además, *Tokenizador* se encarga de guardar y administrar dichos lexemas, para que el analizador léxico disponga de ellos cuando le sean requeridos.

7. Cuando el objeto *Clase* necesita un lexema acude a ***AnalizadorLexico*** que a su vez solicita un lexema al objeto ***Tokenizador***, si *Tokenizador* ya entregó todos los lexemas que poseía, lanzará una excepción a la cual responderá ***ControlArchivo*** repitiendo los puntos 4, 5 y 6.

8. En caso de que ya no sea posible obtener un nuevo lexema; se solicita a ***ControlArchivo*** cerrar el archivo y se envía una alerta al objeto *Clase* indicando el evento. Será éste último quien decidirá, debido a la construcción del sistema, si solicita guardar en ***Informe*** la imposibilidad de leer un lexema más ó decida no hacerlo por el simple hecho de haber terminado la lectura y con ello el análisis satisfactorio de la primera pasada.

6.3.3 Análisis sintáctico

En la sección anterior se abordó el uso y diseño del análisis léxico. En el análisis sintáctico, como se mencionó en el capítulo 2, se realiza la verificación de la estructura del código fuente ingresado por el usuario del compilador en español.

En el capítulo 1, se presentaron algunas técnicas empleadas especialmente en la verificación de un archivo. De entre dichas técnicas vistas, el uso de ***grafos***, aunque relativamente fácil de implementar, es descartado simplemente por el número de expresiones a validar; ya que sería complicado manejar de forma gráfica, el gran número de elementos y posibles caminos a seguir (al verificar carácter por carácter) para señalar la validez del *código fuente* capturado por el usuario. En seguida se muestra un diagrama que ilustra el uso de grafos para describir al conjunto de los números enteros y los reales en el desarrollo de compiladores.

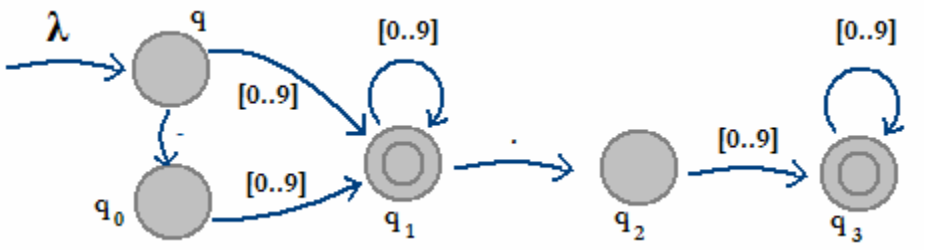


Figura 6-4 Autómata que valida números enteros y reales

Las ***gramáticas de libre contexto*** (***GLC***)[38] permiten, sin ligarnos a algún lenguaje de programación, definir de una forma simple la estructura o patrón que deberá seguir el código fuente.

Retomando el ejemplo anterior, en el cual se describe al conjunto de los números enteros y los reales mediante el uso de grafos; la figura 6-5 servirá para ilustrar la potencia de las *gramáticas de libre contexto* al describir el mismo conjunto.

---

[38] El capítulo 1, presenta un apartado más extenso referente a las gramáticas de libre contexto. Para mayor referencia acudir a: COMPILADORES, PRINCIPIOS, TÉCNICAS Y HERRAMIENTAS. Aho Alfred V., Seti Revi, Ullman Jeffrey D., primera edición, 1998.

Como puede observar, la estructura de la GLC presentada en la figura es simple. Existen tres producciones, cada una de ellas esta compuesta por expresiones terminales, no terminales o ambas; por convención se emplearán mayúsculas para denotar expresiones no terminales.

```
NUMERO_SIMPLE -> - NUMERO_ENTERO | NUMERO_ENTERO | NUMERO_ENTERO.
                 NUMERO_ENTERO   |  - NUMERO_ENTERO.NUMERO_ENTERO
NUMERO_ENTERO    ->  DIGITO | DIGITO NUMERO_ENTERO
DIGITO           ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |  8 | 9
```

Figura 6-5 Gramática de libre contexto que valida números enteros y reales

La lectura que se tendría de dicha gramática sería la siguiente: DIGITO está compuesta por un símbolo que denota los números naturales[39], NUMERO_ENTERO está dada por un DIGITO ó por una secuencia de ellos; mientras que NUMERO_SIMPLE además de ser una expresión no terminal también corresponde a una gramática inicial que da validez a toda la GLC. NUMERO_SIMPLE está dada por uno digito o una secuencia de ellos a la que puede anteceder el signo menos (-). Para incluir los números reales, NUMERO_SIMPLE deberá ser seguido de un símbolo punto (.) y de NUMERO_SIMPLE; la primera aparición de NUMERO_SIMPLE puede estar precedida del signo menos (-). Ahora, fácilmente se puede establecer que lexemas cumplen las reglas impuestas por la GLC, a continuación se presentan algunos ejemplos correctos e incorrectos del uso de la gramática libre de contexto presentada:

CORRECTO	*INCORRECTO*
1	-
-23	19.
-9	.2324
43.34543	-.879
-54343.435	-876.

Tabla 6-3 Ejemplos de cadenas válidas y no válidas para números enteros y/o reales

[39] El símbolo | es empleado para separar las opciones disponibles en una producción de una GLC.

6.3.3.1 Diseño del analizador sintáctico (Primera pasada)

Las GLC no están ligadas a ningún lenguaje de programación, y en muchos casos resulta simple convertir una GLC en una expresión regular. Hoy en día existen muchos lenguajes de programación que presentan apartados dedicados al manejo de expresiones regulares, Java es uno de ellos.

Las ***expresiones regulares*** son una secuencia de caracteres y símbolos que definen un conjunto de lexemas. Son útiles para validar una secuencia de lexemas y asegurar que dichos lexemas estén en un formato específico. Una aplicación de las expresiones regulares es facilitar la construcción de un compilador. A menudo se usan expresiones regulares para validar largas y complejas cadenas de texto, si el código fuente no concuerda con la expresión regular, el compilador sabe que hay un error de sintaxis en el código fuente. Así para la GLC usada como ejemplo, la expresión regular empleada pudiera ser:

(-){2,}(\p{Digit})+(.(\p{Digit})+){2,}[40]

Se presenta ahora la misma expresión regular, pero segmentada en tres partes, con la finalidad de explicar el significado de cada una de ellas:

(-){2,} (\p{Digit})+ (.(\p{Digit})+){2,}

La primera parte indica que el signo menos puede aparecer cero o una vez, la segunda parte indica que deberá venir una secuencia de 1 o más números naturales, la tercera parte indica que de existir, deberá ser como máximo una vez e incluirá a un punto seguido de una secuencia de 1 o más números naturales. Queda pues por validar el tamaño que podrá alcanzar NUMERO_SIMPLE pues el universo de números enteros y reales es infinito.

[40] Basado en la documentación J2SE.
http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html#sum

Se emplearán pues GLC cuyo lenguaje generado se apegue a la definición realizada en el "*lenguaje Java en Español*". En las siguientes secciones encontrará descritas las GLC necesarias en ésta primera *pasada*. El comportamiento esperado en el tiempo será mostrado en la siguiente figura:

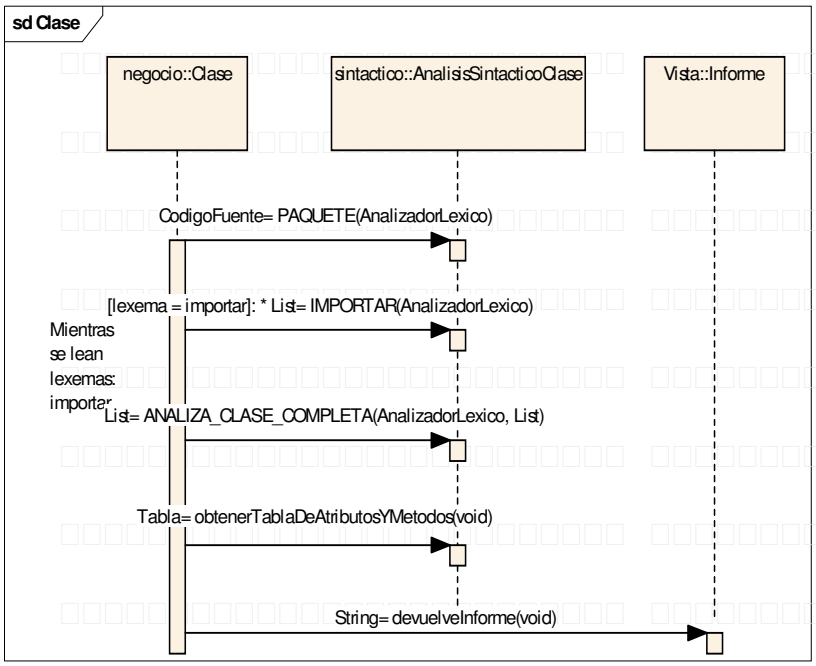


Figura 6-6 Diagrama que ilustra el flujo del proceso de análisis sintáctico

La lectura del diagrama presentado es simple. Primero, Clase desarrollará la GLC PAQUETE, devolviendo el código fuente empleado. En segundo lugar entrará en ciclo donde verificará la GLC IMPORTAR, regresando una lista que contiene el código empleado en cada GLC IMPORTAR. Por otro lado, ANALIZA_CLASE_COMPLETA se encarga de varias cosas:

a) Verificar la GLC DECLARACION_CLASE,

b) Encontrar y validar los atributos existentes en el cuerpo de la *Clase*. Además, deberá agregarlos a una *Tabla*.

c) Encontrar y validar las declaraciones de métodos existentes en el cuerpo de la *Clase*. Además, deberá agregarlos a una *Tabla* que incluirá el código fuente de cada método.

Producto de todo el proceso descrito en este apartado y del referente al Analizador Léxico, se solicita a ***Informe*** el reporte de errores encontrados. Solo habrá un lugar donde se almacenarán los mensajes de error que el compilador vaya encontrando, así que la clase *Informa* trabajará para todas las clases que necesiten ser compiladas en un mismo proceso de compilación; es decir todas las *clases* relacionadas. Es necesario entonces que el acceso de la clase *Informa* sea de *clase*, es decir sin generar *instancia*.

El proceso comentado hasta ahora en el presente apartado, funciona para cada una de las clases que deban ser compiladas. La figura 6-7 muestra que la clase *Análisis* será la encargada de hacer que la primera pasada se aplique para las clases involucradas.

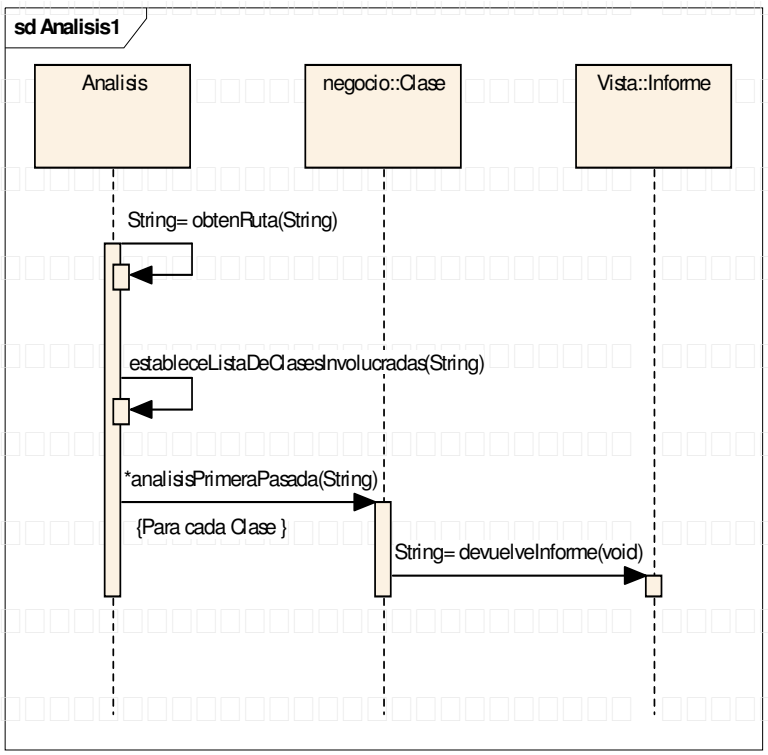


Figura 6-7 Diagrama que extiende la funcionalidad del diagrama presentado en la Figura 6-6

Se observa en el diagrama anterior que la clase *Análisis* en primer lugar tiene que obtener la ruta física donde se encuentra la *clase* base; es decir la *clase* con la que se inicio el proceso de compilación. Posteriormente, *Análisis* establece la lista de *clases* que están relacionadas con la *clase* base (esto incluye las *clases* del mismo paquete, las *clases* públicas importadas y las clases publicas importadas por éstas). Una vez que se cuenta con la lista total de *clases* involucradas se crea la instancia de ellas y se realiza la primera pasada de análisis sintáctico. Según se vayan solicitando las *clases*, se irá guardando el informe de errores registrados en dicha *clase*.

En los siguientes apartados se presentan las gramáticas que se emplearán en ésta primera *pasada*.

6.3.3.1.1 GLC para la declaración de paquete

La GLC que describe la declaración de *paquete* se muestra a continuación:

```
PAQUETE -> paquete CADENA_NOMBRE_ENCABEZADO;
CADENA_NOMBRE_ENCABEZADO ->  IDENTIFICADOR | IDENTIFICADOR .
                             CADENA_NOMBRE_ENCABEZADO
IDENTIFICADOR      ->    _ ALFANUMERICO_COMP | $ ALFANUMERICO_COMP |
                          ALFABETO  ALFANUMERICO_COMP
ALFANUMERICO_COMP  ->    ALFANUMERICO | _ | $
ALFANUMERICO       ->    ALFABETO | DIGITO | ALFABETO ALFANUMERICO |
                           DIGITO  ALFANUMERICO
ALFABETO           ->    a | ... | z | A | ... | Z
DIGITO             ->    0 | ... | 9
```

Figura 6-8 Gramática de libre contexto: Declaración de paquete

El tipo de análisis previsto para la presente gramática es descendente predictivo, así que solamente se podrá iniciar el estudio de la presente GLC cuando durante la lectura del código fuente sea encontrada la palabra reservada ***paquete***.

6.3.3.1.2 GLC para la declaración importar

La GLC que describe la declaración *importar* se muestra a continuación:

```
IMPORTAR       -> importar RUTA_PAQ_CLASE;
RUTA_PAQ_CLASE -> CADENA_NOMBRE_ENCABEZADO.* |
                  CADENA_NOMBRE_ENCABEZADO
```

Figura 6-9 Gramática de libre contexto: Declarción importar

El tipo de análisis previsto para la presente gramática es descendente predictivo, así que solamente se podrá iniciar el estudio de la presente GLC cuando durante la lectura del código fuente sea encontrada la palabra reservada ***importar***.

Aplican las mismas reglas para CADENA_NOMBRE_ENCABEZADO que fue desarrollado en la sección anterior, así que no es necesario volver a desarrollarlo.

6.3.3.1.3 GLC para la declaración de clase

La GLC que describe la declaración de clase se muestra a continuación:

```
DECLARACION_CLASE -> TIPO_ACCESO CLASE   |
                     TIPO_ACCESO CLASE HERENCIA
TIPO_ACCESO       -> ACCESO NON_ACCESO | ACCESO |
                     NON_ACCESO | LAMDA
ACCESO            -> publico | privado | protegido
NON_ACCESO        -> abstracto | estatico | final
CLASE             -> clase IDENTIFICADOR |
                     interfaz IDENTIFICADOR
HERENCIA          -> EXTENDER IMPLEMENTAR
EXTENDER          -> extiende CADENA_NOMBRE_ENCABEZADO | LAMDA
IMPLEMENTAR       -> implementar CADENA_IDENTIFICADORES_COMA
                     | LAMDA
CADENA_IDENTIFICADORES_COMA -> CADENA_NOMBRE_ENCABEZADO |
                  CADENA_NOMBRE_ENCABEZADO ,
                  CADENA_IDENTIFICADORES_COMA
```

Figura 6-10 Gramática de libre contexto: Declaración de Clase

Aplican las mismas reglas para CADENA_NOMBRE_ENCABEZADO que fue desarrollado en un apartado anterior, así que no es necesario volver a desarrollarlo.

6.3.3.1.4 GLC para declaración de atributos y métodos

La GLC que describe la declaración de atributos y métodos se muestra a continuación:

```
DECLARACION_COMPLETA -> TIPO_ACCESO DECLARACION
TIPO_ACCESO       ->  ACCESO | NON_ACCESO | ACCESO NON_ACCESO
DECLARACION       ->  TIPO IDENTIFICADOR; |
                      TIPO IDENTIFICADOR ( PARAMETROS ) CUERPO_METODO
                      | TIPO IDENTIFICADOR ( PARAMETROS ); |
                       IDENTIFICADOR ( PARAMETROS ) CUERPO_METODO
TIPO              ->   _ ALFANUMERICO_COMP | $ ALFANUMERICO_COMP |
                      ALFABETO  ALFANUMERICO_COMP
PARAMETROS        ->  LAMDA | PARAMETRO
PARAMETRO         ->  TIPO IDENTIFICADOR |
                      TIPO IDENTIFICADOR , PARAMETRO
```

Figura 6-11 Gramática de libre contexto: Declaración de atributos y métodos

Aplican las mismas reglas para TIPO_ACCESO, IDENTIFICADOR y ALFANUMERICO_COMP que fueron desarrolladas en un apartado anterior.

6.4 Segunda pasada

Llegado a este punto, el proceso de compilación ha realizado la validación sintáctica de algunas partes del código fuente. La segunda *pasada* del proceso de análisis tendrá como objetivos:

- Diseñar y construir un módulo que verifique sintácticamente el código fuente dentro del cuerpo de un método. De forma paralela en el proceso de desarrollo se construirán y agregarán las tablas correspondientes a las variables locales encontradas en cada bloque parte del método en estudio.

- Es importante que el resultado obtenido en la presente etapa cumpla con los requerimientos solicitados en la tercera etapa, etapa en la cual se realizará básicamente el análisis semántico.

6.4.1 Análisis sintáctico

De forma análoga al proceso empleado para el análisis sintáctico durante la primera *pasada*, en ésta segunda pasada serán empleadas *gramáticas de libre contexto* para definir la estructura de las sentencias internas de cada *método*. El presente apartado centra su estudio en el análisis sintáctico de cada *método* encontrado en cada una de las *clases* involucradas en el proceso de compilación.

6.4.1.1 Diseño del analizador sintáctico (Segunda pasada)

En este momento ya se debería contar con un sistema que realizará el análisis léxico, parte del análisis sintáctico de una clase que incluye la declaración de paquete, la importación de clases, la declaración de clase, la declaración de atributos y la declaración de métodos.

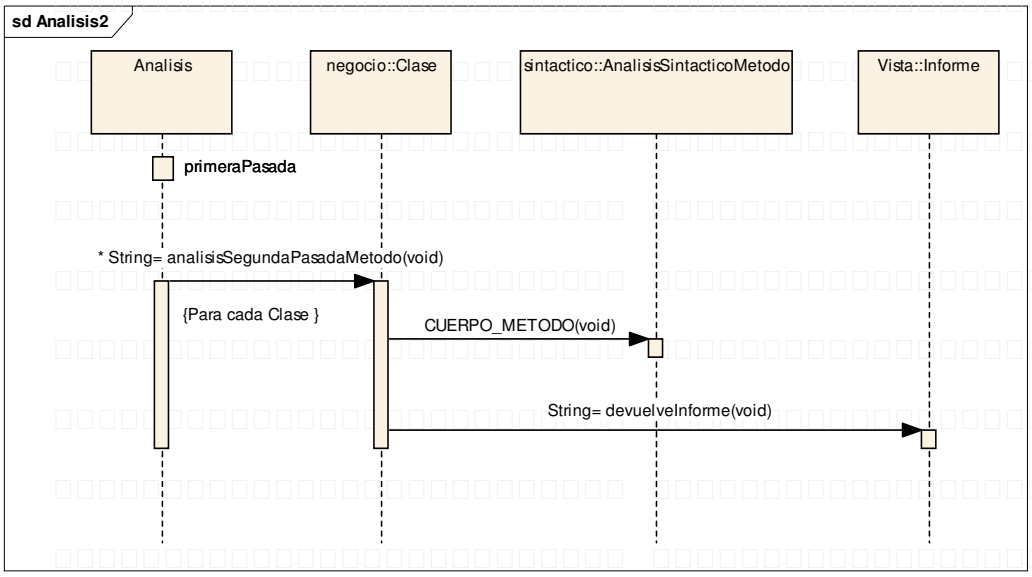


Figura 6-12 Diagrama que ilustra el proceso seguido en la segunda pasada

El presente apartado centra su estudio en el análisis sintáctico de cada *método* encontrado en cada una de las *clases* involucradas en el proceso de compilación. Observe la figura 6-12, el proceso seguido en esta segunda *pasada* es simple:

1. A cada *clase* que necesite ser compilada a consecuencia de su relación con la *clase* base, se deberá realizar la segunda pasada.

2. Cada clase se encargará, durante la segunda *pasada*, de realizar el análisis sintáctico del cuerpo de cada método encontrado en la clase estudiada.

3. CUERPO_METODO generará, durante el análisis sintáctico de cada método de la *clase* en estudio, la lista de variables locales de cada módulo interno del método en estudio.

4. Los tipos de expresiones encontrados y analizados en CUERPO_METODO serán:

    - **Asignaciones,** cubrirá las GLC: ASIGNACION_SIMPLE, ASIGNACION _VIA_FUNCION y ASIGNACION_NUEVA_REFERENCIA.

    - **Expresiones de control,** cubrirá las GLC: EXP_IF y EXP_WHILE.

    - **Llamadas a función,** cubrirá la GLC: EXP_METODO.

    - Serán incluidas también una expresión para *impresión* y una expresión de *retorno* de datos desde una función.

Una vez terminado el análisis sintáctico de los métodos de una clase, se solicitará el informe de errores registrados y dicho informe será regresado al objeto *Análisis* que es quien se encargará de procesar la información.

Es momento de presentar las GLC que se emplearán para verificar sintácticamente el cuerpo de cada método de una clase. Más adelante, durante la tercera *pasada*, se presentarán algunas GLC que servirán de apoyo en la determinación de tipos leídos para las variables encontradas en el código fuente. De esta forma, en la siguiente pasada se comparará el tipo leído contra el tipo declarado, si hay diferencias se lanzará el mensaje correspondiente.

6.4.1.1.1 GLC para verificar expresiones

El cuerpo de un método pude ser encontrado en dos condiciones:

- Sin contener expresión alguna.

- Conteniendo una o más expresiones.

Una expresión puede ser una sentencia de código o un bloque de sentencias reunidas en una estructura de control. Por comodidad, serán clasificadas en: Sentencias de asignación, sentencias de ejecución de métodos y estructuras de control de flujo. Así la GLC para expresiones es:

```
EXPRESION    -> EXP_IF | EXP_WHILE | EXP_UNITARIA |
                EXP_METODO | EXP_UNITARIA |
                regresar DATO;  imprimir ( DATO );
```

Figura 6-13 Gramática de libre contexto: Declaración de una expresión

La figura 6-13 muestra las GLC que sería necesario desarrollar para verificar sintácticamente el cuerpo de cada *método* de cada *clase* estudiada. Pareciera simple desarrollar las gramáticas de libre contexto ubicadas en el lado derecho de la producción presentada; sin embargo, en los apartados siguientes observará que su desarrollo puede resultar bastante complejo de implementar.

6.4.1.1.2 GLC para verificar asignaciones

La GLC para una expresión donde se verifican asignaciones podría ser:

```
EXP_UNITARIA -> ASIGNACION_SIMPLE  |  ASIGNACION_VIA_FUNCION  |
                ASIGNACION_NUEVA_REFERENCIA
```

Figura 6-14 Gramática de libre contexto: Declaración de expresiones unitarias

El desarrollo de las GLC mostradas en la figura 6-14, será presentado en las siguientes secciones.

6.4.1.1.2.1 GLC para verificar asignaciones simples

La GLC para realizar una asignación simple es:

```
ASIGNACION_SIMPLE -> COMUN_ASIGNACION DATO ;
COMUN_ASIGNACION  -> TIPO IDENTIFICADOR = | IDENTIFICADOR =
DATO              -> DATO_NUMERICO | BOOLEANO_SIMPLE | TEXTO_SIMPLE |
                     DATO_CARACTER | IDENTIFICADOR
BOOLEANO_SIMPLE   -> falso | verdadero
DATO_NUMERICO     -> NUMERO_SIMPLE OP_ARITMETICO NUMERO_SIMPLE |
                     NUMERO_SIMPLE OP_ARITMETICO DATO_NUMERICO
NUMERO_SIMPLE     -> - NUMERO_ENTERO | NUMERO_ENTERO |
                     NUMERO_ENTERO.NUMERO_ENTERO  |
                     - NUMERO_ENTERO.NUMERO_ENTERO
NUMERO_ENTERO     ->  DIGITO | DIGITO NUMERO_ENTERO
DIGITO            ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |  8 | 9
OP_ARITMETICO     -> + | - | / | * | %
CARACTER          -> 'ALFANUMERICO' | 'SIMBOLO'
```

Figura 6-15 Gramática de libre contexto: Verificar asignaciones simples

Se puede ver que de haber desarrollado todas las expresiones incluidas en la figura 6-13, dentro de la misma figura, hubiera sido un motivo de confusión tanto para el lector como para el adecuado diseño y desarrollo de la GLC; ya que como puede observar, el desarrollo de apenas una de ella ha resultado bastante extenso.

6.4.1.1.2.2 GLC para verificar asignaciones vía función

La GLC para verificar una asignación vía función son:

```
ASIGNACION_VIA_FUNCION  -> COMUN_ASIGNACION IDENTIFICADOR
                           ( PARAMETROS_PASADOS ) ; |
                           COMUN_ASIGNACION IDENTIFICADOR .
                           IDENTIFICADOR ( PARAMETROS_PASADOS ) ;
PARAMETROS_PASADOS      ->  PARAMETRO_SIMPLE | PARAMETRO_SIMPLE,
                            PARAMETROS_PASADOS
PARAMETRO_SIMPLE        ->  LAMDA | TEXTO_SIMPLE | CARACTER_SIMPLE |
                            NUMERO_SIMPLE |  BOOLEANO_SIMPLE |
                            IDENTIFICADOR
```

Figura 6-16 Gramática de libre contexto: Verificar asignaciones vía función

Las GLC TEXTO_SIMPLE, CARÁCTER_SIMPLE, NUMERO_SIMPLE y BOOLEANO _SIMPLE serán abordados en la siguiente *pasada*.

6.4.1.1.2.3 GLC para verificar una asignación vía nueva instancia

La GLC para verificar una asignación vía nueva instancia es:

```
ASIGNACION_NUEVA_REFERENCIA -> COMUN_ASIGNACION nuevaInstancia
                               IDENTIFICADOR( PARAMETROS_PASADOS ) ;
```

Figura 6-17 Gramática de libre contexto: Verificar asignaciones vía nueva instancia

Aplican las mismas reglas para COMUN_ASIGNACION, IDENTIFICADOR y PARÁMETROS_PASADOS que fueron desarrolladas en un apartado anterior.

Una nueva observación importante, es que el análisis de las GLC de libre contexto presentadas, en muchos casos no soportarán el diseño descendente predictivo y se tendrá que recurrir al retroceso a fin de encontrar la gramática adecuada.

6.4.1.1.3 GLC para verificar sentencias de ejecución de método

La GLC para verificar una sentencia de ejecución de método:

```
EJECUTA_METODO -> IDENTIFICADOR ( PARAMETROS_PASADOS ) ; |
      IDENTIFICADOR . IDENTIFICADOR ( PARAMETROS_PASADOS ) ;
```

Figura 6-18 Gramática de libre contexto: Verificar una sentencia de llamada de método

Aplican las mismas reglas para la GLC PARÁMETROS_PASADOS y IDENTIFICADOR que fueron desarrolladas en un apartado anterior.

Se puede comentar que se podría desarrollar una GLC más compleja que incluya llamadas recursivas de método; pero, no se realizará para dar simplicidad al diseño e implementación.

6.4.1.1.4 GLC para verificar sentencias de control de flujo

La GLC para verificar una expresión de control de flujo:

```
EXP_IF -> si ( EVALUACION_BOLEANA ) CUERPO_EXPRESION |
          si ( EVALUACION_BOLEANA ) CUERPO_EXPRESION
          otro CUERPO_EXPRESION
EVALUACION_BOLEANA   -> IDENTIFICADOR OP_IGUALDAD NUMERO |
          IDENTIFICADOR  OP_LOGICO IDENTIFICADOR | IDENTIFICADOR |
          ! IDENTIFICADOR
CUERPO_EXPRESION     -> EXPRESION | { CONJUNTO_EXPRESIONES }
CONJUNTO_EXPRESIONES -> EXPRESION | EXPRESION CONJUNTO_EXPRESIONES
EXPRESION            -> EXP_IF | EXP_WHILE | EXP_UNITARIA |
                        EXP_METODO | EXP_UNITARIA |
                        regresar DATO ; | imprimir ( DATO ) ;
```

Figura 6-19 Gramática de libre contexto: Verificar sentencias de control de flujo

No está demás recordar que dentro de una GLC las palabras escritas en minúsculas representan terminales en el lado derecho de una producción, además de que se referirán a palabras reservadas y serán señaladas en **negritas**.

6.4.1.1.5 GLC para verificar sentencias repetitivas

La GLC para verificar una sentencia repetitiva:

```
EXP_WHILE -> mientras ( EVALUACION_BOLEANA ) CUERPO_EXPRESION
```

Figura 6-20 Gramática de libre contexto: Verificar sentecias de control repetitivas

Aplican las mismas reglas para la GLC EVALUACIÓN_BOLEANA y CUERPO_EXPRESION que fueron desarrolladas en eL apartado anterior.

6.5 Tercera pasada

Llegado a este punto, el proceso de compilación ha realizado la validación sintáctica completa del código fuente. La tercera *pasada* del proceso de análisis tendrá como objetivos:

- Diseñar y construir un módulo que verifique que se cumplan las reglas semánticas impuestas por el *lenguaje Java en Español*. apoyado en las tablas obtenidas a lo largo de las pasadas anteriores.

- Es importante que el resultado obtenido en la presente etapa cumpla con los requerimientos solicitados en la cuarta etapa, etapa en la cual se realizará básicamente la construcción del *código objeto*.

6.5.1 Analizador semántico

En este punto, ya debió haber sido verificada la estructura sintáctica de cada una de las sentencias capturadas en cada *clase* desarrollada. Ahora, el análisis semántico se encargará de verificar que la estructura de los lexemas encontrados tenga sentido. Por lo que será en este momento cuando se realice la verificación de tipos. Observe el siguiente fragmento de código de la clase Muestra[41]:

```
7.  int a = 5;
8.  char b = 'f';
9.  boolean c = a + b;
10. Matriz matriz = otraMatriz;
```

Figura 6-21 Fragmento de código en lenguaje Java

El fragmento de código anterior es sintácticamente correcto. Pero, se pueden realizar las siguientes observaciones:

1. El tipo de dato declarado en la línea 9 es boleano; es decir, está preparado para recibir valores *falso* o *verdadero*. Se puede ver que en realidad se está asignando la *suma* de dos variables declaradas previamente. Podríamos afirmar que dicha asignación no tiene sentido, es decir es semánticamente incorrecta.

2. La operación de suma que se está realizando en la línea 9 está aplicándose a datos de tipo entero y carácter, lo cual aunque lo permitiera el lenguaje Java, pudiera devolver un resultado no esperado, así que pareciera no tener sentido.

3. En la línea 10 se realiza una declaración de una variable de tipo Matriz, lo cual obliga a que la clase Matriz exista y haya sido importada por la *clase* Muestra o cuando menos la clase Matriz pertenezca al mismo paquete la *clase* Muestra. En caso de que Matriz no existiera, no tendría sentido su presencia en el código fuente mostrado, aún cuando sintácticamente fuera correcto.

[41] Fragmento de código de la *clase* Muestra, basada en lenguaje de programación Java.

6.5.1.1 Diseño de analizador semántico

En el análisis sintáctico, según el diseño planteado, se genera la tabla de variables declaradas en cada módulo del *código fuente*. Ahora, será responsabilidad del analizador semántico la verificación de tipos en las asignaciones, llamadas a métodos y en general el cumplimiento de las reglas impuestas en la definición del lenguaje.

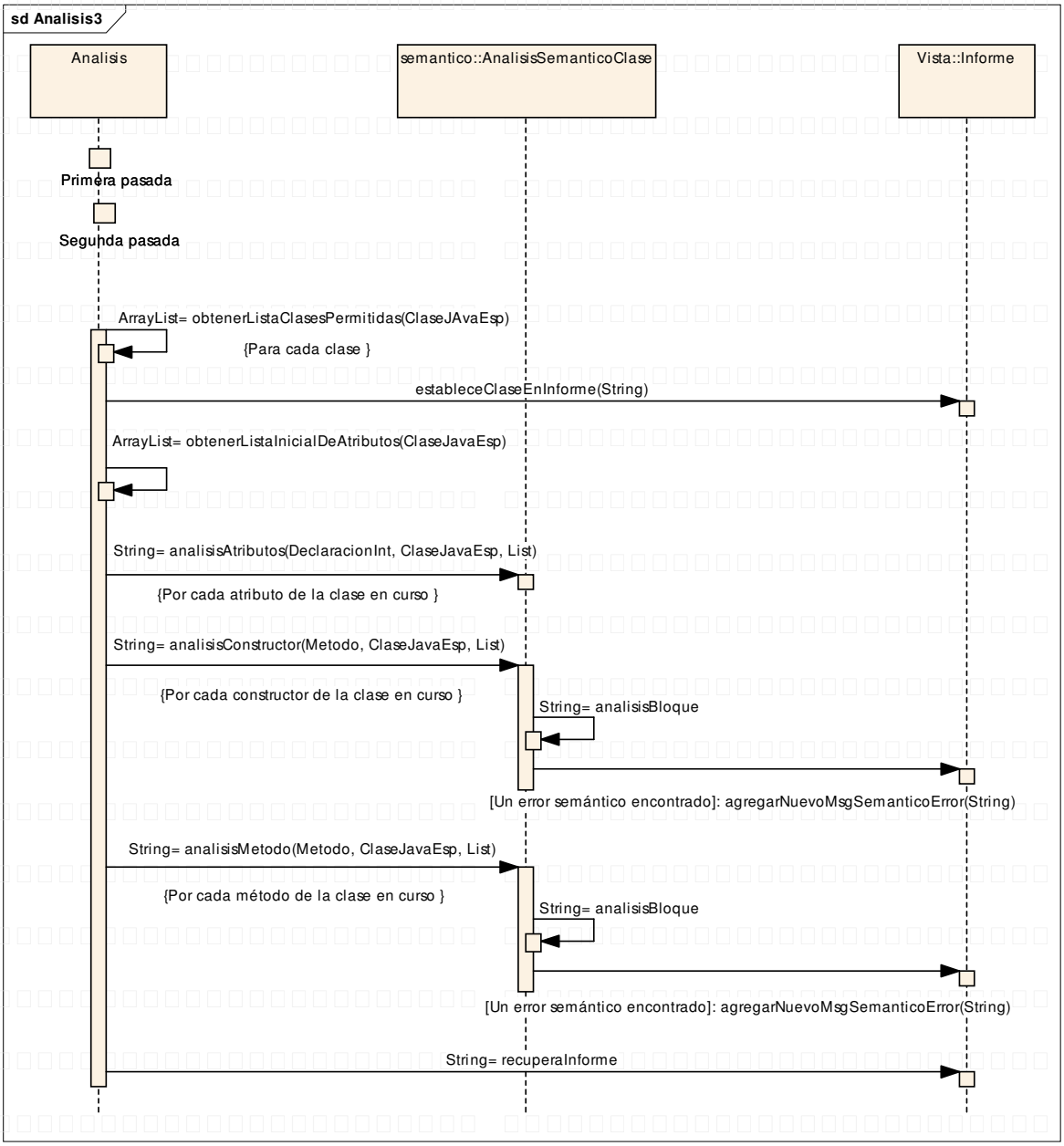


Figura 6-22 Diagrama que esquematiza el proceso de análisis semántico

Es conveniente recordar que durante la *pasada* anterior, entre otras cosas se generó la tabla de variables locales, para lo cual se realizó la lectura y asignación del tipo encontrado en el *código fuente* para cada variable a la que se le asignó un valor. Así en la presente *pasada*, se comparará el tipo leído contra el tipo declarado, si hay diferencias se lanzará el mensaje correspondiente. La figura 6-22 muestra el flujo que podría seguir la información en el proceso de análisis semántico.

Observará en el diagrama presentado que, al igual que en los diagramas mostrados en la primera y segunda *pasada*, es desde la *clase* ***Analisis*** donde se inicia el respectivo proceso de análisis que interesa en ese momento. Según el diagrama mostrado, la lectura de la presente *pasada* podría ser de que por cada una de las clases que serán compiladas la *clase* ***Analisis*** se encarga de:

1. Determinar la lista de clases a las que tendrá acceso la *clase* en estudio. Se puede deducir que el presente punto es de gran trascendencia por ser la lista de clases obtenidas la que determinará los tipos no primitivos que podrán ser empleados dentro del cuerpo de la *clase* en estudio para declarar tipos de retorno de métodos, tipos de variables de instancias y locales.

2. ***Analisis*** prepara la *clase* ***Informa*** para permitir el almacenado de los mensajes de error encontrados en la *clase* en estudio.

3. Obtener la lista de atributos de la *clase* en estudio y apoyado por la clase ***AnalisisSemanticoClase*** realizar su análisis semántico.

4. Obtener la lista de constructores de la *clase* en estudio y apoyado por la clase ***AnalisisSemanticoClase*** realizar su análisis semántico.

5. Obtener la lista de métodos de la *clase* en estudio y apoyado por la clase ***AnalisisSemanticoClase*** realizar su análisis semántico.

6. Luego del estudio de cada *clase*, ***Analisis*** recupera los mensajes de error generados.

Dentro del diagrama presentado en la figura 6-22, algunos métodos citados llevan como parámetros los tipos: ***DeclaracionInt***, ***Metodo***, ***ClaseJavaEsp***. *DeclaracionInt*, que viene de "declaración interna", es empleada para contener los elementos que describen un atributo y por consecuencia a una variable local. *Metodo* comparte herencia con *DeclaracionInt*, es usada para contener tanto los elementos de la declaración de un método, como los elementos que componen el cuerpo del método y que en general se refieren a cada una de las *expresiones* encontradas en el cuerpo del método estudiado; dichas *expresiones* fueron obtenidas durante la segunda *pasada* del proceso de análisis. *ClaseJavaEsp*, será abordada durante la etapa de *síntesis*, pues su creación apoya un punto muy especial dentro del desarrollo del *compilador en Español*, por lo pronto sólo se enuncia su existencia.

Es claro que en el diagrama presentado no fueron incluidos todos los puntos que abarca el análisis semántico; pero, refleja los principales. Habría que agregar la verificación de cuestiones de herencia manejadas a nivel de *método* y *clase*, el uso de modificadores de acceso de clase (palabra reservada *abstracto*), etc. Aunque éstos puntos no son incluidos en el diagrama presentado, algunos son incluidos en el desarrollo del compilador y se dará cuenta de ellos al final del presente capítulo.

6.5.1.1.1 GLC de apoyo en el análisis semántico

Uno de los objetivos del análisis semántico es la verificación y validación de tipos. Al realizar una asignación de alguna expresión en una variable es necesario verificar que los datos asignados correspondan al tipo asignado en la declaración de la variable modificada, ya sea que se hable de asignación por funciones que regresan un valor, creación de nuevas instancias, variables o datos directos.

Respecto a la asignación de datos directos diremos que, si no fue declarada una variable no existe registro del dato en la tabla de variables y por lo tanto no es posible determinar su tipo. Si la variable fue declarada previamente, la forma de proceder consiste en evaluar la expresión a asignar para determinar el tipo de que se trata y poder realizar la comparación correspondiente. En los siguientes apartados serán presentadas algunas GLC en apoyo a la detección de tipos desde un archivo que contiene un programa hecho *en lenguaje Java en Español*. El tipo leído será comparado contra el esperado en una asignación, en caso de no haber concordancia un mensaje de error será enviado a la *clase Informe*.

6.5.1.1.2 GLC para verificar tipos de datos boleanos

A continuación se presenta la GLC de apoyo en la identificación de datos de tipo boleano:

```
BOOLEANO_SIMPLE ->   falso | verdadero
```

Figura 6-23 GLC para identificación de datos boleanos

6.5.1.1.3 GLC para verificar tipos de datos carácter

A continuación se presenta la GLC de apoyo en la identificación de datos de tipo carácter:

```
CARACTER_SIMPLE ->   'SIMBOLO' | 'ALFABETO' | 'DIGITO'
SIMBOLO         ->   ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - |
                     . | / |  : | ; | < | = | > | ? | @ | [ | \ | ] | ^
                     | _ | ` | { | | | } | ~
```

Figura 6-24 GLC para identificación de datos de tipo carácter

En secciones anteriores fue desarrollada la GLC para ALFABETO y DIGITO, así que no será realizado nuevamente.

6.5.1.1.4 GLC para verificar tipos de datos texto

A continuación se presenta la GLC de apoyo en la identificación de datos de tipo texto:

```
TEXTO_SIMPLE     ->  "TEXTO_CORTO" | "LAMDA"
TEXTO_CORTO      ->  ESPACIO | ESPACIO TEXTO_CORTO | SIMBOLO |
                     SIMBOLO TEXTO_CORTO | ALFABETO TEXTO_CORTO  |
                     ALFABETO | DIGITO | DIGITO TEXTO_CORTO
ESPACIO     ->
```

Figura 6-25 GLC para identificación de datos de tipo texto

En secciones anteriores fue desarrollada la GLC para ALFABETO, DIGITO y SÍMBOLO, así que no será realizado nuevamente.

6.5.1.1.5 GLC para verificar tipos de datos real y entero

A continuación se presenta la GLC de apoyo en la identificación de datos de tipo real y entero:

```
NUMERO_SIMPLE -> - NUMERO_ENTERO | NUMERO_ENTERO |
                  NUMERO_ENTERO.NUMERO_ENTERO   |
                 -  NUMERO_ENTERO.NUMERO_ENTERO
NUMERO_ENTERO     ->  DIGITO | DIGITO NUMERO_ENTERO
DIGITO            ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |  8 | 9
```

Figura 6-26 GLC para identificación de datos de tipo numérico

Se debe tener presente que para el caso de los números enteros no sería posible cualquier combinación que incluyera el punto decimal. Sin embargo se tiene una consideración más, la longitud del dato introducido deberá ser validada conforme los valores máximos soportados por J*ava* para sus tipos de datos *int* y *float*. Tampoco hay que olvidar que incluso aquellos tipos de datos que soportan números de mayor magnitud, como los tipos de datos *long* y *double*, también tienen límites ya definidos.

Los tipos de datos *int* y *float* del *lenguaje Java* representarán a los tipos de datos *entero* y *real* del *lenguaje java en español* respectivamente. Mientras que los valores *long* y *double* representarán a los tipos de datos *largo* y *doble* respectivamente.

Por cuestiones de tiempo, únicamente se tomarán en cuenta dos tipos de datos numéricos: enteros y dobles.

6.6 Cuarta pasada

Llegado a este punto, el proceso de compilación ha completado el análisis léxico, la validación sintáctica y semántica de cada *clase* involucrada con la *clase* base. Es tiempo de la *parte de síntesis*. La cuarta *pasada* tendrá como objetivos:

- Teniendo como entrada las N *clases* verificadas durante la *parte de Análisis*, diseñar y construir un módulo que entregue como resultado un archivo con extensión *java.* El archivo obtenido podrá ser compilado por el compilador Java; es decir, en el presente módulo se realizará la traducción de cada c*lase* verificada y proporcionada inicialmente *en lenguaje Java en Español* en una clase en *lenguaje Java*.

6.6.1 Generación de código intermedio

Hay una amplia variedad de interpretaciones de lo que ésta fase debe realizar, incluido hasta donde debe llegar, y aún cuando fue presentada la teoría en el capítulo 2, vale la pena enunciar cuando menos otro punto de vista en la generación de código intermedio. “El código intermedio puede tomar muchas formas: existen casi tantos estilos de código intermedio como compiladores. Sin embargo, todos representan alguna forma de *linealización* del árbol sintáctico, es decir, una representación de árbol sintáctico en forma secuencial. El código intermedio puede ser de muy alto nivel, representar de todas las

operaciones de manera casi tan abstracta como un árbol sintáctico, o parecerse mucho al código objeto"[42].

La teoría descrita a lo largo del capítulo 2 dice que en la fase de generación de código intermedio se crea una representación que sea fácil de traducir al *código objeto*. Dicha generación de código se puede iniciar desde que se está realizando el análisis sintáctico ó desde el semántico. Es común encontrar relacionada la construcción de árboles sintácticos del análisis sintáctico a la generación de código intermedio denominado "*de tres direcciones*", el cual a su vez es fácilmente aterrizado en lenguaje ensamblador. Hasta el momento no se ha planteado la construcción de árboles sintácticos, pues se espera que no sea necesario dado el tipo de *código objeto* de salida que se pretende del compilador.

En la práctica, desde el desarrollo del análisis sintáctico (durante la construcción de la denominada segunda *pasada*) se ha trabajado en una clase especial denominada ***ClaseJavaEsp***. *ClaseJavaEsp* almacena durante la segunda pasada, adecuadamente organizada, todos los elementos que permiten la descripción de: La declaración de paquete, de importación de clases, la declaración de clase, la declaración de atributos, la declaración de constructores, la declaración de métodos y las *expresiones* contenidas en el cuerpo de cada método.

6.6.2 Optimización de código

Toca el turno a la optimación de código. Al hablar de optimizar el código ya obtenido se quiere decir que se busca mejorar o hacer más eficiente el código ya generado. Típicamente hay dos formas en las que se puede optimizar el código:

- ***Optimización en tiempo***. Realizada para permitir que la ejecución del *código objeto* sea más rápida. Es común incluir y hacer uso de nuevas variables denominadas temporales donde se almacenen nuevos fragmentos de código, producto de la

[42] CONSTRUCCIÓN DE COMPILADORES: PRINCIPIOS Y PRÁCTICA, Kenneth C. Louden, pág 398.

división y/o simplificación de expresiones complejas con el fin de reducir el tiempo de procesamiento. Dado que en este tipo de optimación es incluido código nuevo, es común que el *código objet*o sea más grande de lo que era el primer código generado.

- ***Optimización en espacio***. Realizada para hacer posible la obtención de archivos *código objeto* de menor tamaño, ideales para dispositivos de poca capacidad de almacenaje como pueden ser los dispositivos móviles actuales. La optimación en espacio típicamente repercute en el tiempo de ejecución.

Como se comentó en capítulo2, debe de justificarse el tiempo empleado en el desarrollo de un optimizador un código. Ya que uno de los fines del desarrollo del compilador es de tipo didáctico y dado que en el compilador sólo fueron incluidos los aspectos básicos del lenguaje de programación Java, se tomó la decisión de no invertir tiempo en el diseño y desarrollo de esta fase.

6.6.3 Generación de código

Esta será la última fase realizada por el *Compilador en Español*; el objetivo de la presente fase, según se describió en el capítulo 2, consiste en obtener el denominado "*código objeto*" como producto del proceso de compilación. La figura 6-27 ilustra el proceso.

Será necesario, en este momento, establecer las reglas con las cuales se obtendrá el ***código objeto***. El siguiente apartado presenta el proceso a seguir en la generación y obtención del *código objeto compilador en Español*.

6.6.3.1 Equivalencias en el proceso de generación de código

La filosofía con la que nació el *lenguaje Java en Español* es la orientación a objetos y su estructura es muy parecida a la estructura del *lenguaje Java*. El *compilador en Español*

deberá pasar de un código hecho en un lenguaje orientado a objetos recibido como entrada a un código hecho en otro lenguaje orientado a objetos. Al haber similitudes en la estructura podría pensarse que desde un inicio bastaba hacer una traducción de las palabras encontradas en el primer lenguaje al segundo lenguaje. Sin embargo, como se ha explicado a lo largo del presente material, ese no es el objetivo. Se ha procurado en la medida de lo posible mantener un diseño y desarrollo congruente a la teoría de construcción de compiladores; teniendo por último objetivo, luego de contar ya con un *código intermedio* muy cercano al *código objeto,* el de crear cada una de los archivos que contengan el programa equivalente ahora en *lenguaje Java*, generados producto de una compilación exitosa en la que no fueron encontrados errores.

Una vez que ha sido analizado el código intermedio y que se encuentra debidamente organizado, puede ser llevado al lenguaje de interés. La forma en la que será organizado el código asentado en cada archivo con extensión *java*, será la que a continuación se presenta:

1. Declaración de paquete.

2. Declaración de *clases* importadas.

3. Declaración de *clase*. Ésta incluirá, en caso de ser necesario, la declaración de *clase* de la cual hereda.

4. En el siguiente orden, el cuerpo de una clase incluirá:

    - Declaración de atributos.

    - Declaración de constructores. En caso de que no haya sido declarado constructor alguno y la clase no sea abstracta, el compilador generará uno con una funcionalidad nula; es decir, sin contener expresiones en su cuerpo.

- Declaración de métodos. El cuerpo de un método será representado por un símbolo punto y coma (;) en caso de tratarse de un *método abstracto*.

El inicio de cada uno de los puntos anteriores será debidamente señalado con alguna leyenda, manejada como comentario.

Bajo el esquema comentado, ahora es posible realizar una analogía entre los fines perseguidos con las sentencias del *lenguaje fuente* y el *lenguaje de salida*. Luego de evaluar los fines de cada sentencia en los dos lenguajes, se pudo determinar los elementos del *código intermedio* que necesitaban cambiar para obtener el *código objeto* que concilie con el lenguaje de salida, obteniendo el resumen mostrado en la siguiente tabla.

Lenguaje Java En Español	Lenguaje Java
paquete	*package*
importar	*import*
extiende	*extends*
implementa	*implements*
interfaz	*interfaz*
publico	*piblic*
privado	*private*
protegido	*protected*
estatico	*static*
doble	*double*
real	*float*
verdadero	*trae*
nuevaInstancia	*new*

Lenguaje Java En Español	Lenguaje Java
caracter	*char*
Texto	*String*
boleano	*bolean*
falso	*false*
abstracto	*abstract*
clase	*class*
entero	*int*
vacio	*void*
si	*if*
otro	*else*
mientras	*while*
imprime	*System.out.print*
regresar	*return*

Tabla 6-4 Equivalencias empleadas en el proceso de síntesis, durante la generación de código

La figura 6-27 presenta un diagrama en el cual se puede ver la simplicidad con la que se plantea la cuarta *pasada*. ***Compilador*** le indica a la instancia de ***Analisis*** que realice las primeras tres *pasadas*. Luego de realizar las tres primeras *pasadas*, compilador solicita la lista de objetos ***ClaseJavaEsp*** que fueron generados a lo largo de dichas *pasadas*. En caso de que no existan errores, *Compilador* solicitará la generación de clases *java* a la instancia de ***Sintesis***, a la cual le proporcionará la lista de objetos *ClaseJavaEsp* obtenidos anteriormente. *Sintesis* se auxilia en ***ClaseJava*** para obtener el *código objeto* de cada uno

de los objetos *ClaseJavaEsp*. En éste justo momento, en caso de no haberse presentado errores, se dispone del *código java* deseado. Sin embargo, será la clase Archivo la encargada de almacenar en el paquete correspondiente la nueva *clase* Java obtenida previamente.

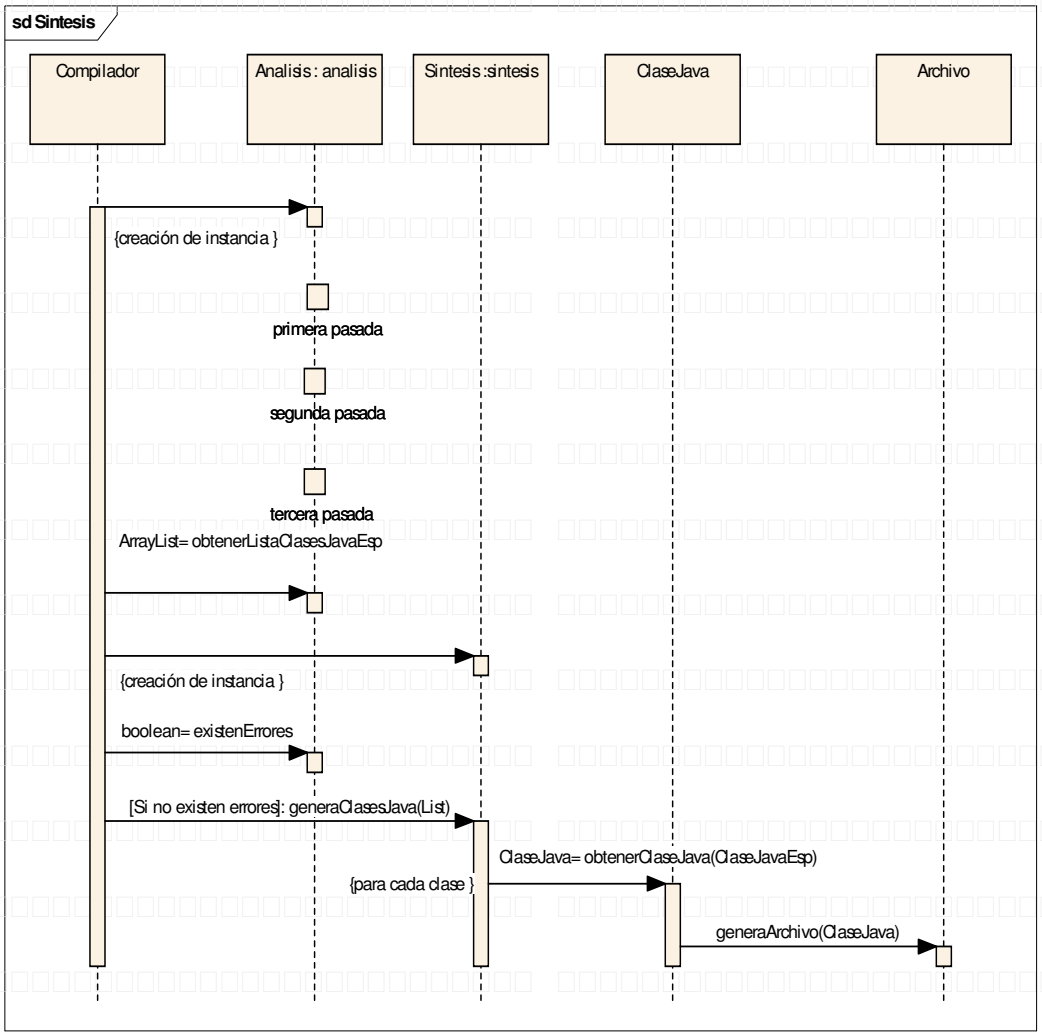


Figura 6-27Diagrama que esquematiza el proceso de generación de código

6.7 Implementación

El sistema será desarrollado en *lenguaje Java*, haciendo uso de:

- Java, en particular el software J2SE versión 5.0.

- Eclipse, que es una herramienta de desarrollo que soporta J2SE, a fin de facilitar la organización del software a desarrollar.

6.7.1 Requerimientos del sistema

Los requerimientos mínimos con los que se debe contar, son los siguientes:

- Sistema operativo Windows XP.
- Una memoria RAM de 0.5Gb.
- J2SE versión 5.0.
- Eclipse como herramienta de apoyo de Java.

Se parte del supuesto que se reúnen estas condiciones en el equipo de cómputo donde sea probado el sistema.

6.8 Compilador en Español en acción

De forma breve en los siguientes apartados encontrará información que describe el proceso de configuración del sistema desarrollado, su uso y ejemplos descriptivos.

6.8.1 Configuración del Compilador en Español

El sistema incluye una serie de archivos de configuración con propósitos diferentes. Ubique la siguiente dirección:

CompiladorEspJava\configuracion

En la ruta donde descargó el sistema. En ella encontrará los siguientes archivos:

- **confInit.properties.** En él se determina la ruta a partir de la cual el compilador buscará diferentes archivos que compilará. Desde este mismo archivo se podrá determinar la extensión de los archivos a compilar.

- **log.properties.** En él se determina el lugar donde serán generados los archivos referentes a informes, depuración ó errores que sean generados al realizar una compilación.

- **MSGerror.properties.** En él se encuentra la lista de mensajes, que producto de una compilación no exitosa, son presentados al usuario.

- **palabrasReservadas.properties.** En él se encuentra la lista de palabras reservadas empleadas por el *compilador en Español*.

6.8.2 Uso del Compilador en Español

El uso del *compilador en Español* es muy simple. La forma más corta de hacer uso de él es simplemente ejecutando desde consola de DOS la instrucción de ejecución de *java* junto con la ruta, el nombre de archivo y la extensión del archivo a partir del cual se desee realizar el proceso de compilación; es decir la *clase* que será considerada como base:

java [ruta .]nombreArchivo.jave

La descripción de los elementos que se incluyen en la instrucción, se comentan a continuación:

- **ruta.** Se refiere al paquete o carpeta en la que se encuentra la *clase* que será considerada como base. En caso de que la *clase* base no haya sido incluida en un paquete, no será incluida la *ruta*. Si la *clase* base se encuentra incluida en una jerarquía de carpetas, será separada cada una de esas carpetas con el operador punto (.).

- **nombreArchivo.** Se refiere al nombre del archivo que será considerado como *clase* base en el proceso de compilación.

- **jave.** Corresponde a la extensión que deberá guardar el archivo que se desea compilar. hay que tener presente que fue un requerimiento planteado durante el desarrollo del presente texto.

Existe una segunda posibilidad de iniciar un proceso de compilación. En ella se incluirá la forma ya presentada más una serie de parámetros, observe la siguiente instrucción:

java [ruta .]nombreArchivo.jave boleano1 boleano2 boleano3 boleano4

Algunos de los elementos que se incluyen en la instrucción ya fueron comentados, a continuación la descripción de los elementos restantes:

- **boleano1, boleano2, boleano3 y boleano4.** Cada uno de los cuatro elementos podrá tener el valor: **falso** ó **verdadero**. El colocar alguno de los cuatro valores a *verdadero* le indica al compilador que se requiere a la salida del proceso de compilación, un reporte generado de alguna de las 4 *pasadas* realizadas en el proceso de compilación. El número concatenado a la palabra *boleano* indica el número de *pasada* de la que se desea el reporte. Es obligatorio incluir los cuatro parámetros adicionales en este modo de compilación.

6.8.3 Ejemplos acerca del uso del Compilador en Español

A lo largo del presente apartado serán presentados algunos ejemplos ilustrativos del uso del compilador y de los reportes que podría obtener del mismo.

6.8.3.1 Ejemplo de una compilación exitosa

Suponga el siguiente código fuente como entrada del compilador:

```
// Este es un ejemplo de una clase generada con el lenguaje en Español
//Aquí va la declaración de paquete
paquete operacion;
//Aquí va la declaración de clase
publico abstracto clase Operacion{
        /**Realizará la operación de dos números de tipo entero
            el método regresará un dato de tipo entero*/
        publico abstracto entero realizarOperacion(entero param1, entero param2) ;
        /**Realizará la operación de dos números de tipo doble
            el método regresará un dato de tipo entero*/
        publico abstracto doble realizarOperacion (doble param1, doble param2) ;
}
```

Figura 6-28 Programa en lenguaje Java Español

El código fuente no presenta ningún error que rompa las reglas establecidas para el lenguaje, así que la salida presentada al usuario al final del proceso de compilación será:

```
<<<<<<<<<<<<<<<<<<  RESULTADOS DE COMPILACIÓN   >>>>>>>>>>>>>>>>>>
Se encontraron  ==  0 == Errores de compilación

================     CLASES GENERADAS  ================

\operacion\Operacion.java


<<<<<<<<<<<<<<<<<<  FIN RESULTADOS DE COMPILACIÓN   >>>>>>>>>>>>>>>>>>
```

Figura 6-29 Resultado de una compilación exitosa

Se puede observar en la figura 6-29 que un proceso de compilación exitoso además de informar al usuario acerca de la inexistencia de errores, también le informa de los archivos generados como producto de la compilación y de su respectiva ubicación.

La figura 6-30 muestra el código almacenado en el archivo *Operación.java* citado en la figura 6-29. Aún si tiene los conocimientos básicos del lenguaje Java podrá determinar que el código presentado compilará sin errores. Desde luego que también puede realizar la prueba usando el compilador de Java.

```
//Paquete
package operacion;
//Clases importadas
//Ninguna clase fue importada
//Declaración de clase
public abstract class Operacion{
     //Atributos
     //Ningún atributo
     //Constructores
     //Clase abstracta
     //Métodos
     public abstract int realizarOperacion  ( int param1, int param2 ) ;
     public abstract double realizarOperacion  ( double param1, double param2 ) ;
}
```

6.8.3.2 Ejemplo de una compilación con errores sintácticos

Suponga el código fuente presentado en la figura 6-30 como entrada del compilador. El código fuente presenta errores que rompen las reglas sintácticas establecidas para el lenguaje. Se tiene por ejemplo que en la línea:

privado entero caracter ;

Posterior a la declaración del tipo se utiliza como identificador, en la declaración de atributo, una palabra reservada. En la siguiente línea de código se tiene:

privado entero variable

Se puede ver que en la declaración del atributo no aparece el símbolo de término punto y coma (;). Las dos líneas de código violan las reglas sintácticas impuestas al *lenguaje Java en Español*.

```
// Este es un ejemplo de una clase generada con el lenguaje en Español
//Aquí va la declaración de paquete
paquete operacion;
//Aquí va la declaración de clase
publico abstracto clase Operacion{
        privado entero caracter ;
        privado entero variable1
        /**Realizará la operación de dos números de tipo entero
            el método regresará un dato de tipo entero*/
        publico abstracto entero realizarOperacion(entero param1, entero param2) ;
        /**Realizará la operación de dos números de tipo doble
            el método regresará un dato de tipo entero*/
        publico abstracto realizarOperacion (doble param1, doble param2) ;
}
```

Figura 6-30 Programa en lenguaje Java en Español

Continuando con el ejemplo, el proceso de compilación arroja la siguiente salida:

```
<<<<<<<<<<<<<<<<<<  RESULTADOS DE COMPILACIÓN   >>>>>>>>>>>>>>>>>>


Clase: Operacion
Linea -> 6 -- privado entero caracter;
El nombre de identificador no puede ser una palabra reservada == caracter ==

Clase: Operacion
Linea -> 10 -- publico abstracto entero realizarOperacion(entero param1, entero
param2) ;
El elemento encontrado no es el esperado --> == publico ==

Clase: Operacion
Linea -> 10 -- publico abstracto entero realizarOperacion(entero param1, entero
param2) ;
El elemento esperado es -- ; --

<<<<<<<<<<<<<<<<<<  FIN RESULTADOS DE COMPILACIÓN   >>>>>>>>>>>>>>>>>>
```

Figura 6-31 Resultado del proceso de compilación de la figura 6-29

Como podrá observar, una vez compilado el programa. En el caso de presentar errores, el compilador lista la mayor cantidad de errores que haya sido posible encontrar. Puede ver

que se ha tratado de presentar los mensajes de forma clara y que con un poco de intuición y conocimiento en el tema podría resolver dichos errores con la finalidad de que obtenga el *código objeto* al compilar nuevamente.

6.8.3.3 Ejemplo de una compilación con errores semánticos

Suponga el siguiente código fuente como entrada del compilador:

```
// Este es un ejemplo de una clase generada con el lenguaje en Español
//Aquí va la declaración de paquete
paquete operacion.suma;
//Aquí va la declaración de clase
publico clase Suma extiende operacion.Operacion{
        /**Realizará la operación de dos números de tipo entero
            el método regresará un dato de tipo entero*/
        publico entero realizarOperacion(entero param1, entero param2) {
            entero resultado  =  param1 + param2;
            regresar resultado;
        }
        /**Realizará la operación de dos números de tipo doble
            el método regresará un dato de tipo entero*/
        publico doble realizarOperacion (doble param1, doble param2) {
            Texto miTexto  =  "mi texto" ;
            boleano variable_boleana = miTexto;
            entero resultado  =  param1 + param2;
            regresar variable_boleana;
        }
}
```

Figura 6-32 Programa en lenguaje Java en Español

El código fuente presenta errores que rompen las reglas semánticas establecidas para el lenguaje. Se tiene por ejemplo que la línea:

boleano variable_boleana = miTexto;

Sintácticamente es correcta; sin embargo, está intentando asignar una variable de tipo *Texto* a una variable de tipo *boleano* lo cual es incongruente. La línea:

entero resultado = param1 + param2;

Sintácticamente es correcta; sin embargo, está intentando asignar el resultado de una operación de números de tipo *doble* a una variable de tipo *entero*. Por último, la línea:

regresar variable_boleana;

Sintácticamente es correcta; sin embargo, el tipo de retorno indicado por el *método* es doble. La figura siguiente muestra la salida vista por el usuario, producto del proceso de compilación del código fuente estudiado.

```
<<<<<<<<<<<<<<<<<<  RESULTADOS DE COMPILACIÓN   >>>>>>>>>>>>>>>>>>


Clase: Suma
Error encontrado en análisis semántico
Los tipos empleados no pueden ser usados en la asignación: Suma-realizarOperacion
Sentecia: variable_boleana = car
LI: boleano - LD: Texto   Válido: false

Clase: Suma
Error encontrado en análisis semántico
Los tipos empleados no pueden ser usados en la asignación: Suma-realizarOperacion
Sentecia: resultado = param1 + param2
LI: entero - LD: DOBLE    Válido: false

Clase: Suma
Error encontrado en análisis semántico
El tipo regresado no es correcto: Suma-realizarOperacion
Expresión de retorno encontrada
      Valor devuelto: variable_boleana - Tipo leído: IDENTIFICADOR
Tipo declarado: doble - Tipo regresado: boleano --> Válido: false

<<<<<<<<<<<<<<<<<  FIN RESULTADOS DE COMPILACIÓN   >>>>>>>>>>>>>>>>>
```

Figura 6-33 Resultado del proceso de compilación del código de la figura 6-32

Se observa en la figura 6-33, al igual que en el ejemplo anterior, que de forma simple se señala el tipo de error encontrado. Incluso se señala, en el ejemplo, el tipo de dato de la variable a asignar (LI – lado izquierdo de la expresión de asignación) y el tipo asignado (LD – lado derecho de la expresión de asignación). Para el caso del tipo de

retorno, se comenta cual es el tipo declarado y cual es el tipo regresado para indicar la razón por la que la validación no es correcta. Se observa en todos los casos que los errores señalados en la figura 6-33 concuerdan con los errores previamente comentados.

6.8.3.4 Resumen de la primera pasada

Ha sido explicado con anterioridad el proceso para obtener el reporte de cada pasada. Retomando el ejemplo presentado en la figura 6-28, la siguiente figura ilustra la salida:

```
<<<<<<<<<<<<<<<<<<<<<<<<<<                                  >>>>>>>>>>>>>
<<<<<<       Lista de clases en la carpeta: operacion          >>>>>>
<<<<<<<<<<<<<<<<<<<<<<<<<<                    >>>>>>>>>>>>>>>>>>>>>>>>
Archivo encontrados - ruta: operacion
0 Operacion.txt
1 suma
       ======================================================================
              ============    ANÁLISIS CLASE: Operacion    ===========
       ======================================================================
------------  Código externo analizado  ------------
paquete operacion ;
publico abstracto clase Operacion {
}
-----------  Fin Código externo analizado -----------

-----------  División por segmentos del código   ----
------------  Paquete   -----------
operacion
-----------  Fin Paquete -----------

-----------   Importa   ------------
-----------  Fin Importa ------------

-----------  Firma Clase ------------
publico abstracto clase Operacion
---------  Fin Firma Clase -----------

-------------  Atributos -------------
------------  Fin Atributos -------------

======     Métodos constructores    =====
=======  FIN  Métodos constructores   =======

==========    Métodos encontrados   =========
Método : publico abstracto entero realizarOperacion ( entero param1 ,
entero param2   )
-----------  Cuerpo Método  -----------
----------  FIN Cuerpo Método -----------
Método : publico abstracto doble realizarOperacion ( doble param1 ,
doble param2   )
-----------  Cuerpo Método  -----------
----------  FIN Cuerpo Método -----------
========    FIN Métodos encontrados   ========
-------- FIN  División por segmentos del código   ----
```

Figura 6-34 Resumen reportado por la primera pasada para el código presentado en la figura 6-28

Se puede apreciar en la figura 6-34, que el objetivo principal es la separación de cada una de las partes que describen la *clase* en estudio. El proceso deja pendiente para la siguiente *pasada* la revisión sintáctica de los métodos y constructores de dicha *clase*.

6.8.3.5 Resumen de la segunda pasada

Considere el ejemplo código mostrado en la figura 6-32.

```
      ============    ANÁLISIS CLASE: Suma    ================
...
==================  Análisis Constructor  ======================
==============  FIN  Análisis Constructor ======================
============================  Análisis Método  ========================
Método : publico entero realizarOperacion  ( entero param1  , entero param2  )
--------  Variables locales ----------
entero resultado
-----------  Cuerpo Método -----------
**********    EXPRESIÓN     *****************
*********  ASIGNACIÓN SIMPLE   **************
Asignación: resultado = param1 + param2
Tipo de dato leído:       param1 - IDENTIFICADOR,   param2 - IDENTIFICADOR
********   FIN DE EXPRESIÓN     *************
**********    EXPRESIÓN     *****************
********  RETORNO DEL MÉTODO  **************
      Valor devuelto: resultado - Tipo leído: IDENTIFICADOR
********   FIN DE EXPRESIÓN     *************
-----------  FIN Cuerpo Método  ------------
Método : publico doble realizarOperacion  ( doble param1  , doble param2   )
--------  Variables locales ----------
Texto miTexto, boleano variable_boleana, entero resultado
-----------  Cuerpo Método -----------
**********    EXPRESIÓN     *****************
*********  ASIGNACIÓN SIMPLE   **************
Asignación: miTexto = "mi texto"
Tipo de dato leído:       "mi texto" - TEXTO_SIMPLE
********   FIN DE EXPRESIÓN     *************
**********    EXPRESIÓN     *****************
*********  ASIGNACIÓN SIMPLE   **************
Asignación: variable_boleana = miTexto
Tipo de dato leído:       miTexto - IDENTIFICADOR
********   FIN DE EXPRESIÓN     *************
**********    EXPRESIÓN     *****************
*********  ASIGNACIÓN SIMPLE   **************
Asignación: resultado = param1 + param2
Tipo de dato leído:       param1 - IDENTIFICADOR,   param2 - IDENTIFICADOR
********   FIN DE EXPRESIÓN     *************
**********    EXPRESIÓN     *****************
********  RETORNO DEL MÉTODO  **************
      Valor devuelto: variable_boleana - Tipo leído: IDENTIFICADOR
********   FIN DE EXPRESIÓN     *************
-----------  FIN Cuerpo Método  ------------
===============  FIN  Análisis Método ======================
```

Figura 6-35 Resumen de la segunda pasada del código de la figura 6-32

Aunque la figura 6-35 no muestra la primera parte del resumen obtenido del proceso de compilación en la segunda *pasada*, ésta no se traduce en perdida de información, ya que se refiere sólo a parte del resumen presentado en la primera *pasada*; es decir, se presenta la información correspondiente a la declaración de paquete, *clases* importadas y de *clase*. No hay métodos constructores, de ahí que se encuentre vacío el apartado destinado a presentar su resumen. Sin embargo, se puede ver que los dos métodos desarrollados se encargan de presentar el tipo de *expresión* encontrado; de acuerdo al tipo de *expresión* encontrada, es el proceso de validación sintáctica aplicado.

6.8.3.6 Resumen de la tercera pasada

Considere el ejemplo código mostrado en la figura 6-32.

```
...
=================================================================
      ======  Análisis semántico de la Clase: Suma  ======
=================================================================

===============  Análisis de sus Atributos   ================
===============  Análisis de sus Constructores   ============

-----------     publico  Suma()     -------------
-----   -----    -----     -----
-----   -----    -----     -----

===============  Análisis de sus Métodos   ==================
---  publico  entero realizarOperacion(entero param1,entero param2) ---
-----   -----    -----     -----
Sentencia: resultado = param1 + param2
LI: entero - LD: ENTERO Válido: true
Expresión de retorno encontrada
      Valor devuelto: resultado - Tipo leído: IDENTIFICADOR
Tipo declarado: entero - Tipo regresado: entero --> Válido: true
-----   -----    -----     -----
--- publico  doble realizarOperacion(doble param1,doble param2)     ----
-----   -----    -----     -----
Sentencia: miTexto = "mi texto"
LI: Texto - LD: TEXTO_SIMPLE  Válido: true
Sentecia: variable_boleana = miTexto
LI: boleano - LD: Texto Válido: false
Sentencia: resultado = param1 + param2
LI: entero - LD: DOBLE  Válido: false
Expresión de retorno encontrada
      Valor devuelto: variable_boleana - Tipo leído: IDENTIFICADOR
Tipo declarado: doble - Tipo regresado: boleano --> Válido: false
-----   -----    -----     -----
```

Figura 6-36 Resumen de la tercera pasada del código de la figura 6-32

Se puede observar de la figura 6-36 al revisar semánticamente, el cuerpo de los métodos de la *clase* Suma, que se presenta el resultado de la validación según el tipo de *expresión* encontrada. Se señala el tipo de dato de la variable a asignar (LI – lado izquierdo de la expresión de asignación) y el tipo asignado (LD – lado derecho de la expresión de asignación).

6.8.3.7 Resumen de la cuarta pasada

Considere el ejemplo código mostrado en la figura 6-28. El resumen presentado como producto de la cuarta pasada es bastante simple, de hecho es el mostrado en la figura 6-30. Como recordará en dicha figura se presenta el código fuente en *lenguaje Java*, que es obtenido luego de una compilación exitosa, razón por la cual no se extenderá en el tema.

7 Conclusiones

Una vez concluido el presente documento y su correspondiente sistema, es tiempo de volver la vista sobre los pasos recorridos y verificar simplemente que los objetivos trazados al inicio del presente trabajo hayan sido cumplidos.

Como pudo observarse, las primeras dos partes del presente material; es decir, los primeros cuatro capítulos:

- Teoría de lenguajes formales y autómatas
- Compiladores
- Orientación a objetos
- Java

Presentan el material con el que se cumple la meta de reunir información referente a: "Teoría de Lenguajes", "Teoría de compiladores" y "Paradigmas de programación". Se cumple así el primer objetivo. Ese mismo material proporciona la teoría sobre la cual se construyen los capítulos 5 y 6 que se citan a continuación:

- Lenguaje Java en Español
- Compilador en Español

En el capítulo 5, "*Lenguaje Java en Español*", se realiza la definición y delimitación de un nuevo lenguaje de programación que cumple con el requisito inicial de que sus palabras reservadas fueran en el idioma Español. Está basado en el paradigma de programación de la

Orientación a Objetos y toma como modelo al popular lenguaje de programación Java. Queda aclarado, con esta breve explicación, el origen del nombre del lenguaje de programación desarrollado. Queda claro que se siguen cumpliendo los objetivos trazados.

Quizá de poco hubiera servido la creación de un nuevo lenguaje informático de programación sin alguna herramienta que permita hacer uso de él. Así pues, durante el capítulo 6, "*Compilador en Español*", se presenta el material encaminado a diseñar y desarrollar un compilador. Herramienta con la cual se pretende obtener un programa escrito en *lenguaje Java* a partir del *lenguaje Java en Español* definido en el capítulo 5.

Adicionalmente, se presenta un software desarrollado a partir de las líneas plasmadas en los capítulos señalados. Dicho software tiene funcionalidad muy limitada, sin embargo, podría ser ampliado y mejorado sin dificultad aunque si con una considerable inversión de tiempo, razón por la cual no se ha realizado desde éste momento. El elaboración y presentación del diseño y desarrollo del *compilador en Español*, indica que se siguen cumpliendo los objetivos iniciales del presente texto.

En resumen, de acuerdo a la información asentada en los párrafos anteriores se considera cubierto el propósito planteado al inicio del presente proyecto de tesis. Un hueco queda aún, y es realizar una invitación a las personas del área informática a que se sumen en favor del desarrollo en México, estoy convencido que existe mucho talento en cada individuo, sabemos que no sólo en el área de desarrollo de software se pueden diseñar y desarrollar cosas nuevas e interesantes. Hay mucho, mucho por hacer.

Finalmente, antes de dar por concluido el presente texto, quisiera plasmar y trasmitir mis experiencias. En lo personal, el desarrollo tanto del presente documento como del sistema, me deja un grato y un amargo sabor de boca. Muchos fueron los conocimientos que adquirí al paso de mi investigación acerca de los diferentes tópicos; sin embargo, eso me ha servido únicamente para darme cuenta que hay un mar de información y que he tomado apenas un grano de arena.

Me parece importante citar que aún cuando la idea original al comenzar ésta odisea, que representó el desarrollo del tema elegido, en principio buscaba que el *código objeto* obtenido por el compilador fuera un archivo con extensión *.class* que pudiera ser directamente interpretado por la *máquina virtual de Java*, ello implicaba sumergirse en un mar aún mayor de conocimientos e inversión de tiempo en el desarrollo, así que muy a mi pesar y con una inversión de tiempo ya realizada decidí acortar los alcances hasta quedar en el actual material presentado. Sin embargo considero que no todo está perdido ya que, si algo aprendí de la realización del presente trabajo es que, a partir del *código intermedio* generado tengo una y mil posibilidades para aterrizar un *código objeto* final; de tal forma que el presente trabajo, aún con todas sus limitaciones y defectos, tiene un enorme potencial de crecimiento.

No puedo negar que me sorprendí y emocioné cuando logré ver funcionar cada parte del compilador que desarrollaba; sin embargo entendí, aunque quizá no de la mejor forma, que aunque me haya preciado de ser un buen programador, el tiempo y las actividades de nuestra vida diaria pueden causar estragos en las decisiones que tomamos antes, durante y después de iniciar y/o desarrollar un proyecto, de tal modo que algunos fragmentos de código que he desarrollado no reflejan las mejores practicas de programación. Aún con todas las desventuras y momentos difíciles vividos a largo del desarrollo del presente texto y del compilador, me siento muy contento y espero sinceramente que las líneas encontradas en el presente documento puedan ayudar en algo a quien en ellas busque apoyo.

Nunca viviré lo suficiente para vivir

Jesús López

8 Bibliografía

1. APRENDIENDO JAVA 2, Lemay Laura, Cadenhead Rogers, edit. Prentice Hall, primera edición, 1999.

2. APRENDIENDO PROGRAMACIÓN ORIENTADA A OBJETOS, Sintes Anthony, edit. Prentice Hall, primera edición, 2002.

3. AUTÓMATAS Y LENGUAJES, Brena Ramón, Tec de Monterrey, 2003.

4. COMPILADORES, PRINCIPIOS, TÉCNICAS Y HERRAMIENTAS. Aho Alfred V., Seti Revi, Ullman Jeffrey D., edit. Addison Wesley, primera edición, 1998.

5. CONSTRUCCIÓN DE COMPILADORES: PRINCIPIOS Y PRÁCTICA, Kenneth C. Louden, edit Paraninfo, primera edición, 2004.

6. FUNDAMENTOS DE PROGRAMACIÓN. Peñaloza R. Ernesto, edita UNAM, tercera edición, 2001.

7. INTRODUCCIÓN A LA TEORÍA DE AUTÓMATAS, LENGUAJES Y COMPUTACIÓN, Hopcroft John E., Ullman Jeffrey D., edit. CECSA, primera edición, 2000.

8. JAVA, CÓMO PROGRAMAR, Deitel Harvey M., Deitel Paul J., edit. Prentice Hall, quinta edición, 2004.

9. JAVA EN POCAS PALABRAS, Flanagan David, edit. Mc Graw Hill, primera edición, 1999.

10. LEARNING JAVA, Knudsen Jonathan, Niemeyer Patrick, Mc Graw Hill, tercera edición, 2005.

11. PROGRAMACIÓN UML 2, Arlow Jim, Neustadt Ila, edit. Anaya, 2006.

12. THE JAVA LENGUAGE SPACIFICATION, Gosling James, Joy Hill, Steele Guy, Bracha Gilad, Addison Wesley, tercera edición, 2005.

13. THE JAVA VIRTUAL MACHINE SPECIFICATION, Leindholm Tim, Yellin Frank, Addison Wesley, segunda edición, 1999.