INSTITUTO POLITÉCNICO NACIONAL

UNIDAD PROFESIONAL INTERDISCIPLINARIA DE INGENIERÍA Y CIENCIAS SOCIALES Y ADMINISTRATIVAS

SECCIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

RECONOCEDOR DE LENGUAJES CON BASE EN GRAMÁTICAS FORMALES

T E S I S

QUE PARAOBTENER EL GRADO DE MAESTRO
EN CIENCIAS EN INFORMÁTICA
P R E S E N T A

HORACIO ALBERTO GARCÍA SALAS

México D.F. 2007

SIP-14

INSTITUTO POLITECNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISION DE TESIS

En la Ciudad de México, D. F., siendo las 12:00 horas del día 20 Del mes de Septiembre del 2006 se reunieron los miembros de la Comisión Revisora de Tesis designada Por el Colegio de Profesores de Estudios de Posgrado e Investigación de U P I I C S A
Para examinar la tesis de grado titulada:
"RECONOCEDOR DE LENGUAJES CON BASE EN GRAMÁTICAS FORMALES"

Presentada por el alumno:

GARCÍA	SALAS	HORACIO ALBERTO
Paterno	materno	nombre(s)

Con registro:

A0	1	0	3	9	9

Aspirante al grado de:
MAESTRO EN CIENCIAS EN INFORMÁTICA

Después de intercambiar opiniones los miembros de la Comisión manifestaron ***SU APROBACION DE LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISION REVISORA

Director de Tesis

M. en C. EDUARDO RENÉ RODRÍGUEZ AVILA

M. en C. EMILIA ABAD RUIZ

DR. MAURICIO JORGE PROCEL MORENO

DR. ERIC MANUEL ROSALES PEÑA ALFARO

M. en C. EDUARDO VEGA ALVARADO

EL PRESIDENTE DEL COLEGIO

DR. MAURICIO JORGE PROCEL MORENO

S. E. P.
I. P. N.
U. P. I. I. C. S. A
SECCIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

CARTA CESION DE DERECHOS

En la Ciudad de México, D.F. el día 28 del mes de noviembre del año 2006, el que suscribe **Horacio Alberto García Salas,** alumno del Programa de **Maestría en Ciencias con la especialización en Informática** con número de registro **A010399** adscrito la **Sección de Estudios de Postgrado de la UPIICSA-IPN**, manifiesta que es la autora intelectual del presente trabajo de Tesis baja la dirección del **M. en C. Eduardo René Rodríguez Ávila** y cede los derechos del trabajo intitulado **"Reconocedor de Lenguajes con base en Gramáticas Formales"** al Instituto Politécnico Nacional para su difusión con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección itztlii@gmail.com Si el permiso se otorga, el usuario deberá dar el seguimiento correspondiente y citar la fuente del mismo.

Lic. Horacio Alberto García Salas

AGRADECIMIENTOS

«Todo lo que hacemos, todo lo que somos descansa en nuestro poder personal»
«Don Juan»

Todo lo que soy en esta vida Padre te lo debo a ti y no creo que exista modo alguno con el cual pagarte. Quiero expresarte con todo mi corazón, mi más profundo agradecimiento, este trabajo está dedicado a tu venerable persona.

Tu que has sido una persona íntegra, honesta y con un gran corazón, contar con tu ejemplo, con tus palabras de aliento, con tu cariño hacia mi expresado de mil maneras, me has dado las bases con las cuales llevar una vida con muchas satisfacciones, siendo la más grande de ellas tener el honor de ser tu hijo.

Hermanos, sin ustedes mi vida nunca se hubiera visto rodeada de la magia que me ha permitido ver lo maravilloso de esta vida. Quiero agradecerles por todo su amor, todas las enseñanzas, aventuras y momentos que he tenido el privilegio de compartir con ustedes y que han marcado el sendero que me ha tocado recorrer.

Quiero agradecerle a mi dualidad, quien me ha acompañado a lo largo de este camino y con su amor me ha enseñado a sentir la magia de la vida de este mundo maravilloso.

Agradezco a todas a aquellas personas que colaboraron directa o indirectamente en la realización de este trabajo. De manera especial, le quiero agradecer al Dr. Alexander Gelbukh quien con su incondicional apoyo ha hecho posible la culminación de esta Tesis.

Muchas Gracias

Lic. Horacio Alberto García Salas

ÍNDICE

RESUMEN

Durante esta investigación se desarrolló un *sistema reconocedor de lenguajes basado en gramáticas formales*. Es una herramienta informática que condensa algunas de las técnicas utilizadas en el desarrollo de metacompiladores y permite que un lenguaje de programación sea reconocido por el sistema con sólo describirlo a través de su gramática. Es un metacompilador descendente recursivo que integra tanto el analizador léxico como el sintáctico en un solo analizador, permitiendo que el código fuente del sistema sea reducido.

Tomando en consideración potenciales programadores de este sistema, lo más importante es que cuenten con una manera útil y sencilla para describir sus lenguajes. Por esta razón, se ha implementado una interfaz que permite leer la gramática desde un simple archivo de texto y que la descripción de la gramática del lenguaje se haga por medio de reglas de producción con una notación clara y sencilla.

Con el fin de explicar el funcionamiento y la construcción de este sistema, a través de las páginas de este trabajo se hablará primeramente, de manera introductoria, de la teoría de los lenguajes en términos computacionales, que a diferencia del enfoque lingüístico, centra su atención en el desarrollo de herramientas computacionales, cuyo principal objetivo es la creación de *software* para el procesamiento automático de los lenguajes, específicamente lenguajes de programación o computacionales. En este sentido, la idea es que los lenguajes de programación sean más amigables y que impongan menos restricciones, de manera que los programadores tengan mayor libertad para expresar su creatividad, utilizando lenguajes de programación cuyas oraciones sean más parecidas a las utilizadas en el *lenguaje natural*.

Se hace una reseña acerca de la historia de los lenguajes computacionales. Empezando con el que es considerado el primer lenguaje de programación, el *Plankalkül*, hasta el lenguaje *Java* que es de los lenguajes más utilizados actualmente. Como toda ciencia, la de la computación basa su desarrollo en la evolución de los descubrimientos, aplicando nuevas técnicas y conocimientos a los descubrimientos previos, con el fin de obtener resultados más eficientes que los anteriores.

Se presenta una descripción de las gramáticas formales, así como su clasificación en gramáticas irrestrictas, sensibles al contexto, libres de contexto y regulares. La mayoría de los lenguajes de programación pueden ser descritos por medio de este tipo de gramáticas, por lo que, una de las áreas que ha obtenido beneficios del empleo de gramáticas es la de desarrollo de compiladores.

Se describen las principales características de los compiladores como herramientas necesarias para la traducción de programas fuente escritos en lenguaje de alto nivel a lenguaje máquina. En este sentido, se hace referencia de acuerdo a su compilador a dos tipos de lenguajes, interpretados y compilados.

Una vez explicada la teoría sobre la que se basa el desarrollo de esta investigación, se describe el funcionamiento y la manera en la que se llevó a cabo la construcción del sistema. Este sistema consiste de cuatro funciones, dos de ellas se utilizan para hacer un preprocesamiento de la gramática, que permite identificar en las reglas de producción los diferentes símbolos que las constituyen, así como almacenarlas en la memoria primaria con el fin de que estén disponibles al momento de la ejecución. Otra función se encarga del código fuente, lo almacena en un arreglo y en caso de ser necesario elimina los separadores. La cuarta es la función principal y es recursiva, se encarga de verificar la estructura de las oraciones y que sean construidas únicamente con símbolos que pertenezcan al alfabeto del lenguaje.

Después de haber explicado el funcionamiento y la construcción del sistema, se presentan algunas pruebas de lenguajes que es posible reconocer. En estos ejemplos, se muestra la sencillez con la que se describen las reglas de producción de la gramática de un lenguaje. Finalmente, se proporciona el código fuente del sistema.

SUMMARY

During this investigation a language-recognizing system has been developed, based on formal grammars. This software is an informatical tool that encompasses some of the techniques used in the development of metacompilers and allows a programming language to be recognized by this system through the mere description of its grammar. It is a descendent recursive metacompiler that performs both lexical and syntactical analysis with only one analyzer which results in a very reduced source code.

A point has been made, having in mind potential programmers of this system, to have a useful and simple way to describe their own languages. For this reason, an interface has been implemented to read grammar from a text file. Language grammar description can be done based on production rules with a really clear and simple notation.

In order to explain the working process and construction of this system through pages of this work, the language theory will be introduced in terms of information. That represents a different point of view from that of linguistics, centering its focus on computational tools development whose main goal is software creation for automatic processing of languages, specifically programming or computational languages. In this sense, the idea is to render friendlier programming languages and to reduce constraints in a way that programmers can express their creativity more freely, using computational languages whose sentences are more similar to those used in natural languages.

A brief account of the history of computational languages is presented as a background. Starting off with "*Plankalkül*", considered the first programming language, and running all the way to "*Java*", one of the most used languages nowadays. As any other scientific field, computation has based its development on discovery evolution, applying new techniques and knowledge to previous discoveries aiming to obtain progressively more efficient results.

A description of formal grammars follows along with its classification as unrestricted, context sensitive, context free and regular grammars. Most programming languages can be described through this kind of grammars. One of the areas where formal grammars have been widely used is in compilers development.

The main features of compilers are described as translation tools for source programs written in high level language into machine language. In this sense, there are two types of languages according to its compiler, interpreted and compiled.

Once theory has been explained on which this research is based, a description of how this language-recognizer system works and its construction ensues. The whole system consists of four functions, two of them to preprocess the grammar in order to identify the different symbols that constitute the rules of the grammar and store them in main memory to be available in execution time. Another function deals with source code, storing it into one array. The fourth is the main function and is recursive, it verifies sentence's structure and that all symbols used to build sentences belong to the language's alphabet.

After explaining system's working process and construction, tests are presented. They are tests of languages that are possible to recognize. These examples show the simplicity for the description of the rule production for the grammar of a language. Finally, the source code is supplied.

INTRODUCCIÓN

En sus orígenes, las computadoras fueron apreciadas como un invaluable laboratorio en el que el lenguaje de las matemáticas podía ser utilizado de forma práctica; donde todas aquellas abstracciones que los hombres de ciencia creaban únicamente en su mente podían ser "materializadas" por medio de programas escritos con lenguajes que implementaban los *algoritmos*. Sin embargo, la programación de los algoritmos se tenía que hacer por medio de matemáticas y en código binario, por lo que sólo aquellas personas que poseían este conocimiento podían "comunicarse" con las computadoras.

Con la aparición de las computadoras el problema de comunicación se ha acrecentado, ya que las investigaciones referentes a los lenguajes no sólo se limitan a aquéllos que utilizan los seres humanos para comunicarse con sus semejantes sino que la comunicación se da en términos de hombre y máquina. Por esta razón, se han desarrollado una gran cantidad de *lenguajes computacionales*[1] que van desde ensambladores hasta lenguajes declarativos, que permiten que la comunicación entre computadoras y usuarios sea más amigable y, como consecuencia, cada vez más personas se ven beneficiadas con el uso de computadoras.

Así como en otras ramas del conocimiento, la investigación en el campo de los lenguajes ha permitido que se vaya construyendo toda una teoría, que se ha fundamentado sobre la lingüística matemática y es conocida como *lenguajes formales*[1]. Con base en dicha teoría se estudian y desarrollan los lenguajes computacionales. En este sentido, se habla de diferentes tipos de lenguajes clasificados de acuerdo a la cercanía que tienen con el *lenguaje natural*, haciendo con esto referencia al lenguaje que utilizan los seres humanos para comunicarse con sus semejantes. De esta manera, hay lenguajes de alto nivel que permiten que las instrucciones se escriban con oraciones parecidas a las que se usan en el lenguaje natural; algunos de estos lenguajes conocidos son Basic, Pascal, Fortran, Cobol, Algol y Lisp, y lenguajes de bajo nivel que son aquéllos que están más cerca del lenguaje de máquina o binario (0's y 1's), entre los que se encuentran los lenguajes ensambladores.

Sin embargo, pese a los avances tecnológicos, las computadoras no han dejado de funcionar en términos de 0's y 1's, por lo que la utilización de lenguajes de alto nivel requiere de un mecanismo que permita que las instrucciones sean traducidas a lenguaje de máquina. A estos sistemas traductores se les conoce como *compiladores*. En general, los compiladores son sistemas que se utilizan para hacer una traducción de un "lenguaje de alto nivel" a lenguaje de máquina, pero bien se pueden utilizar como reconocedores de lenguajes o generadores de ellos.

[1] En este escrito se hará referencia a lenguaje o lenguaje computacional indistintamente.

En la actualidad, el desarrollo de lenguajes computacionales y compiladores es un área de conocimiento que se trabaja y expone en todas las universidades del mundo, por lo que es un área de investigación madura y de amplio desarrollo. El problema principal que se intenta resolver con el desarrollo de compiladores es facilitar la comunicación hombre-máquina, por lo que la idea básica que persiguen los investigadores de esta área es que las computadoras entiendan en cierta proporción el *lenguaje natural*.

A este respecto, existen diversas líneas de investigación. Una de las herramientas que se han utilizado y con las que se han logrado avances significativos en estas áreas informáticas son las *gramáticas*, que han mostrado una gran versatilidad para *modelar* lenguajes, lo que ha permitido que su aplicación se extienda a lo largo de las distintas ramas del conocimiento.

En México, uno de los problemas que se enfrentan, es que sólo algunos investigadores están dedicados a estas áreas de conocimiento, ya que la mayoría de los profesionistas de las tecnologías de la información están dedicados al desarrollo de sistemas administrativos [50]. Como consecuencia, a pesar de que existen desarrollos de compiladores tan robustos como los desarrollados en el extranjero, no hay ningún compilador mexicano disponible a nivel comercial en el mercado mundial.

En este sentido, en la Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas del Instituto Politécnico Nacional, el M. en C. Eduardo René Rodríguez Ávila, se ha interesado por desarrollar con fines académicos un lenguaje computacional denominado Stop (*Stack Operations*) [42]. Como su nombre lo menciona, este lenguaje basa su funcionamiento en operaciones de pila, que es una de las estructuras principales de una computadora y de donde se deriva el poder de este lenguaje, ya que modela el funcionamiento “natural” de un sistema computacional [41].

El lenguaje Stop ha servido como motivación para el desarrollo de este trabajo, de hecho la idea original surge con el fin de desarrollar un sistema generador del compilador de Stop, al cual sólo se le proporcionaría la gramática del lenguaje Stop y de manera automática se obtendría su compilador. Sin embargo, con la evolución de la investigación se vislumbró la posibilidad de extender las aplicaciones de este sistema, siendo el objetivo general de este proyecto desarrollar una herramienta que permita *generar de manera automática el reconocedor de un lenguaje que sea posible describir a través de una gramática libre del contexto*. La gramática no deberá contener recursividad izquierda y deberá estar factorizada, lo que significa que ninguna de las diferentes opciones que se presentan en una regla de producción deberá empezar con el mismo símbolo.

Algunos de los objetivos que se pretenden alcanzar con este proyecto son:

1) Fines didácticos. Esto es, que se utilice como apoyo en la creación y la enseñanza de lenguajes de programación y lenguajes formales.
2) La aplicación de esta herramienta en la creación de lenguajes de programación. Se espera que el sistema diseñado a partir de este trabajo se utilice como apoyo en la creación y el desarrollo de nuevos lenguajes de programación.
3) Llevar a cabo una mejora de lenguajes existentes. Con este sistema se pueden modificar compiladores de manera rápida; basta con hacer modificaciones sobre la gramática y ejecutar el sistema para verificar los efectos causados por los cambios. De esta forma, prácticamente en tiempo real, se puede mejorar un lenguaje existente.
4) Desarrollo de lenguajes basado en gramáticas libres de contexto. Sólo los lenguajes que puedan ser descritos mediante gramáticas libres de contexto serán reconocidos por este sistema. Esto excluye, por ejemplo, a aquéllos lenguajes que se describen por medio de gramáticas irrestrictas o sensibles al contexto.

CAPÍTULO I

LENGUAJES, GRAMÁTICAS Y COMPILADORES

1.1 LENGUAJES

1.1.1 Teoría de Lenguajes Formales

La ciencia de los lenguajes formales se ocupa de estudiar los lenguajes matemáticamente. Así como en la geometría no existe una definición formal de lo que es un punto, tratándose de lenguajes existe una entidad que no tiene una definición formal, el *símbolo*. Las letras y los dígitos son ejemplos de símbolos [63]. Los símbolos se utilizan para construir "cadenas" o palabras, que se pueden concatenar con otras para formar cadenas más grandes que se denominan oraciones. Una cadena es una secuencia finita de símbolos [2].

En general, un lenguaje es un conjunto de oraciones. Éstas se constituyen de palabras que a su vez se construyen con los símbolos del alfabeto. Las diferencias que existen entre los lenguajes se pueden ver reflejadas en cualquiera de estos tres niveles. 1) Los lenguajes pueden diferir en la manera en la que se forman sus oraciones, es decir, en su *estructura*. 2) En sus *palabras*, en donde los símbolos que las constituyen y la secuencia con la que se disponen permiten diferenciar una palabra de otra. Y 3) en los *símbolos*, significa que el alfabeto de un lenguaje puede ser diferente del de otro, cualquier símbolo puede ser parte del alfabeto de un lenguaje.

Un lenguaje es un conjunto de oraciones que poseen cierta estructura. Estas oraciones están constituidas por palabras o *tokens*, entiéndase *token* como la unidad léxica mínima, A su vez las palabras se forman a partir de un conjunto finito de símbolos, conocidos como el alfabeto del lenguaje. Debido a que las palabras también tienen una estructura, pues cada palabra está formada por una secuencia específica de letras, también pueden ser vistas como oraciones en donde los tokens son las letras o símbolos del alfabeto. Con esto se puede ver que los lenguajes están formados de estructuras que a su vez están formadas de estructuras similares.

Un ejemplo se puede observar en un lenguaje cuya expresión de operaciones aritméticas sea en notación infija. Una expresión válida es: 8*4-3; en este lenguaje se introducen los paréntesis para modificar el sentido de la expresión, ya que existe un orden de ejecución según la posición de los operadores (conocida como prioridad de ejecución de acuerdo al tipo de operador) y con el fin de que la estructura de la operación sea interpretada correctamente. Sin paréntesis el significado de esta oración sería 29, sin embargo 8*(4-3) tiene otro significado semántico: 8. Además, puede haber diferentes oraciones que reflejen mismos significados semánticos, (8+16)/3 que también significa 8. De esta forma, es evidente que los tokens deben respetar la estructura o sintaxis del lenguaje, de lo contrario pueden resultar errores semánticos y por consecuencia una mala comunicación.

Generalmente, el alfabeto del lenguaje se denota por Σ y representa el conjunto de todos los símbolos que pertenecen al lenguaje. El lenguaje que se puede generar utilizando estos símbolos se denota por Σ^* (cerradura de Kleen sobre Σ). Ejemplo, supóngase $\Sigma = \{a,b\}$ a partir de estos símbolos terminales se puede formar el conjunto de todas las oraciones pertenecientes al lenguaje Σ^*, que incluye a la oración vacía $\Sigma^* = \{ \varepsilon ,a,b,aa,ab,ba,bb,aaa,aab,abb,bbb,...\}$. Algunas propiedades de este lenguaje son que es un conjunto infinito, formado por todas las posibles combinaciones que se pueden hacer con los símbolos “a” y “b”. Interesante es el hecho de que todos los lenguajes que se definan con respecto a los símbolos “a” y “b”, están contenidos en el lenguaje Σ^*, de manera que el lenguaje L1={todas las oraciones que tengan más a’s que b’s} o el lenguaje L2={las oraciones de sólo a’s} son subconjuntos del lenguaje Σ^*, ya que Σ^* contiene todas las posibles oraciones sobre el alfabeto $\Sigma = \{a,b\}$, incluyendo la oración vacía, que contiene cero a’s y cero b’s.

1.1.2 Lenguajes Computacionales

El término *lenguaje* engloba una cantidad inmensa de conocimiento, ya que el lenguaje ha acompañado al ser humano desde el momento que tuvo necesidad de comunicarse con sus semejantes. Cada vuelta en la espiral de la evolución ha provocado que los lenguajes tomen senderos diferentes. En el siglo XX, la historia de los lenguajes abrió una nueva página titulada *Lenguajes Computacionales.*

Los lenguajes computacionales nacen al mismo tiempo que las computadoras, de manera particular con la computadora Z1 desarrollada por el alemán Konrad Zuse entre 1936 y 1939. Entre 1945 y 1946 creó el *Plankalkül* (Plan de Cálculos), el primer lenguaje de programación de la historia y predecesor de los lenguajes modernos de programación algorítmica [47].

Los lenguajes computacionales se desarrollaron con el fin de satisfacer la necesidad de comunicación entre las personas y las computadoras. Así como el lenguaje que se utiliza entre los seres humanos de todo el mundo, al que se hace referencia como *lenguaje natural*, que fue evolucionando por caminos diferentes de acuerdo a los diferentes lugares y la gran diversidad cultural que ofrece este planeta, así también, los lenguajes computacionales han evolucionado vertiginosamente y se han desarrollado utilizando diversas técnicas, dando como resultado una extensa variedad de lenguajes de programación.

1.1.3 Breve Historia de los Lenguajes Computacionales

El desarrollo de lenguajes de programación o computacionales ha evolucionado con el trabajo de diferentes grupos de investigación. No es hasta la aparición de las computadoras y áreas como la lingüística matemática, que se han especializado las técnicas para el desarrollo de lenguajes de programación y que permiten entender mejor el lenguaje natural. En la década de 1940 (aunque esta fecha puede diferir entre las diferentes versiones que existen acerca de la historia de la computación), fue eminente la creación de lenguajes que hicieran posible la comunicación entre seres humanos y computadoras. Durante los tiempos de guerra que prevalecieron entonces y en donde los cálculos eran fundamentales, surgen los lenguajes numéricos. Entre éstos se encuentran el lenguaje A-0 desarrollado en la Univac por un grupo cuyo líder era Grace Hopper y, por otro lado, John Bakus desarrolla Speedcoding para la IBM701[45]. Estos lenguajes hacían una traducción de expresiones aritméticas a código de máquina, pero lo más importante es que se empezaron a desarrollar las notaciones simbólicas.

Los primeros lenguajes en aparecer son los llamados *ensambladores*, con los que se dio un gran avance al tener la oportunidad de programar las computadoras a través de *mnemónicos* o claves (y no "cableando" los equipos, ya que las primeras computadoras que funcionaron exitosamente fueron computadoras analógicas, las cuales efectuaban cálculos matemáticos a través de complejos circuitos electrónicos analógicos) para obtener la configuración requerida por medio de códigos numéricos, código de máquina. El empleo de *mnemónicos* para la elaboración de programas fue de gran utilidad, aunque la traducción a código máquina se seguía haciendo de manera manual. No pasó mucho tiempo, cuando se dio el paso decisivo de que fueran las mismas computadoras las encargadas de hacer esta traducción, lo que fue conocido como "ensamblar el programa" y que dio nombre a este tipo de lenguajes. La serie de mnemónicos que utilizan los lenguajes ensambladores, tienen una correspondencia, prácticamente uno a uno con el lenguaje de máquina, lo que permite una programación detallada.

Pese a que los lenguajes ensambladores representan un gran avance, los programadores aún debían ser "verdaderos expertos". Con el paso del tiempo se encontró que los programadores recurrían a ciertos patrones de instrucciones que formaban bloques y tenían que ejecutarse de manera periódica al llevar a cabo una acción en los programas desarrollados, lo que originó la búsqueda de lenguajes que englobaran dichas instrucciones y se presentaran con un lenguaje más amigable para el usuario, más cercano al lenguaje natural.

En 1950, John Bakus dirige una investigación que culmina en el desarrollo del conocido lenguaje Fortran (Formula Translator). Este lenguaje ha sido considerado el primer lenguaje de alto nivel y fue utilizado en sus inicios por computadoras IBM. Con su aparición surge el término *"compilador" como un sistema encargado de hacer la traducción entre un lenguaje de alto nivel y lenguaje de máquina* [7].

En Europa, otra corriente basaba sus investigaciones en la creación de la definición de un lenguaje independiente de la máquina implementando la teoría de gramáticas de Noam Chomsky[45], en particular las gramáticas libres de contexto. Con estas ideas un grupo de la Universidad de Münich, definió el "International Algebraic Language", publicado en Zurich en 1958. La implementación de este lenguaje dio como resultado el "Lenguaje Algol58 (Algoritmic Language)" [7]. Con el lenguaje Algol evolucionaron y se crearon los nuevos conceptos de los lenguajes como: formato libre, declaración explícita de identificadores, estructuras iterativas más generales, recursividad, paso de parámetros por valor y por nombre y estructura de bloques. Tiempo después aparecería Algol60 que sería un lenguaje considerado como expresivo para describir algoritmos y que se convertiría en un estándar que influenciaría a lenguajes como Pascal, C y Ada [44].

Entre 1959 y 1960 un equipo cuyo líder era Grace Hopper del departamento de defensa de EEUU desarrolló Cobol, lenguaje que se utilizó ampliamente por las instituciones para crear sus bases de datos y sus aplicaciones financieras. Debido a los escasos recursos con los que contaban las computadoras cuando este lenguaje fue creado, se decidió por ahorrar memoria en la definición de los formatos de las fechas, utilizando sólo dos caracteres para el año. Esto ocasionarían en el año 2000 pérdidas multimillonarias, en un "fenómeno" que se denominó Y2K (year 2000) y al que se hace referencia en español como "el problema del año 2000" [48][49].

Al final de la década de 1950 fue desarrollado Lisp en el MIT (Massachusetts Institute of Technology) por John McCarthy, siendo desarrollado sobre la base de estructuras generales de listas. Éste ha sido un lenguaje utilizado en inteligencia artificial y han existido múltiples variantes de él, entre las que se encuentran: MacLisp, FrazLisp, CommonLisp, Scheme.

Aproximadamente en los mismos años se diseñó otro lenguaje conocido como APL (A Programming Language), que fue diseñado por K. Iverson en la Universidad Harvard. Fue creado como un lenguaje orientado a resolver cálculos matemáticos, particularmente aquéllos con arreglos y matrices. APL es un lenguaje funcional y el poder de sus operaciones lo ha convertido en parte importante en el desarrollo de lenguajes computacionales. Durante la década de 1960 sería utilizado como la base para el desarrollo de uno de los primeros sistemas de tiempo compartido, que funcionaba sobre una IBM 360.

La experiencia que se adquirió con el desarrollo de los lenguajes computacionales en la década de 1950, provocó en la siguiente una proliferación en este campo. Muchos de los lenguajes desarrollados fueron de propósito específico. En la búsqueda de un lenguaje "universal" un grupo de investigadores desarrolló un proyecto para la IBM, que produjo el lenguaje PL/I. Éste intentó rescatar las mejores características de los lenguajes Fortran, Cobol y Algol60. Fue utilizado con la familia de las computadoras 360 de IBM.

En la década de 1960 se desarrolló Algol68, lenguaje que pretende adicionar características a la versión previa Algol60, aunque, a pesar de ser un lenguaje en el que fue creada una estructura más expresiva y completamente consistente, fue poco utilizado. El esfuerzo que se invirtió en el desarrollo de los lenguajes PL/I y Algol68 fue grande y no dieron los resultados que se esperaban de ellos. No es el caso del Snobol (String Oriented Symbolic Language) que fue desarrollado por R. Griswold en los Laboratorios Bell, siendo uno de los primeros lenguajes procesadores de cadenas.

De esta misma década es Simula67, desarrollado por Kristen Nygaard y Ole-Johan Dahl en Noruega. Este lenguaje fue diseñado para simulaciones y su contribución principal es el entendimiento de un concepto fundamental en los lenguajes orientados a objetos: las *clases*. Simula67 es considerado como el primer lenguaje orientado a objetos.

Un lenguaje no menos importante que los hasta ahora mencionados es Iswin, que fue desarrollado por Peter Landin. Su influencia principal es aún mayor que la del mismo Lisp en el desarrollo de lenguajes funcionales como el ML y el Haskell. Posiblemente, es el primer lenguaje basado en formalismos matemáticos y su comportamiento fue descrito con precisión.

El lenguaje Basic (*Beginners All Purpose Symbolic Instruction Code*) que también fue parte de los desarrollos de esta década de 1960, fue creado por John Kemeny y Thomas Kurtz en el colegio Dartmouth, con la idea de hacer un lenguaje simple para el desarrollo de aplicaciones en los sistemas de tiempo compartido. Su amplio uso en la PC ha permitido que sea un lenguaje que no ha dejado de utilizarse, tanto en escuelas para la enseñanza, en los negocios para el desarrollo de aplicaciones financieras o en la casa.

La década de 1970 fue testigo del crecimiento de la industria de las computadoras y, por consecuencia, del *software*. Fue en estos tiempos en los que Nicklaus Wirth junto con C.A.R. Hore, que eran parte de quienes estaban en contra de abandonar el lenguaje Algol, publicaron Algol-w y en 1971 Wirth desarrolló Pascal, un lenguaje estructurado, con propósitos académicos, basado en las ideas de Algol. El éxito del lenguaje Pascal fue asombroso y, debido a su aceptación en los ámbitos académicos, se desarrollaron aplicaciones de las más variadas, hasta que la creciente necesidad de nuevas aplicaciones sobrepasó los límites para los que fue diseñado Pascal.

En 1972 fue desarrollado el lenguaje C por Dennis Ritchie en los Laboratorios Bell. C es un lenguaje que se basa en los lenguajes B y BCPL [54], y una de sus características principales es que permite el acceso, prácticamente directo, al *hardware,* por lo que las personas dedicadas al desarrollo de *software* encontraron en C un lenguaje de alto nivel que permite desarrollar funciones de un ensamblador.

El lenguaje C tuvo un gran éxito y sería incluido como parte del sistema operativo UNIX y con ello empezaría la competencia por dominar el mercado de los sistemas operativos, otra vertiente en la historia del desarrollo de software. Pese a que ni C ni Pascal aportaron nuevos conceptos al desarrollo de lenguajes, han tenido una presencia sólida alrededor del mundo, por lo que una gran cantidad de aplicaciones se han desarrollado utilizando estos lenguajes.

Otros lenguajes fueron implementados en la década de los 1970, entre los que se encuentra Clu, desarrollado en el MIT por un equipo cuyo líder era Barbara Liskov. El mecanismo de manejo de excepciones que se implantó en Clu se utilizó como base en el desarrollo de un mecanismo similar de un lenguaje posterior conocido como Ada.

Euclid es un lenguaje que se desarrolló en la Universidad de Toronto entre 1976 y 1977. Este lenguaje es una extensión del lenguaje Pascal, incorporando tipos de datos abstractos y evitando características no deseables en la programación.

Xerox desarrolló entre 1976 y 1979, en el centro de investigación en Palo Alto, el lenguaje llamado Mesa. Es un lenguaje experimental parecido al Pascal e incorpora manejo de excepciones y mecanismos de concurrencia o programación en paralelo, lo que permite que este lenguaje sea utilizado para desarrollar sistemas operativos.

La década de los 1980 comienza con la aparición de dos lenguajes: Ada y Modula-2. Ambos lenguajes tienen relación con Pascal, por lo que pueden ser considerados como parte de la misma familia de lenguajes.

Ada es un lenguaje diseñado por el departamento de defensa de los EEUU con la finalidad de satisfacer la necesidad de desarrollo de aplicaciones militares. Después de un concurso de 17 proyectos, un grupo, cuyo líder era J. Ichbiah, fue quien desarrolló Ada. Su diseño comenzó a partir del lenguaje Pascal, sin embargo, el resultado final fue un lenguaje diferente con características tales como un mecanismo para el tipo de datos abstractos (el paquete, “package”), un mecanismo de concurrencia (la tarea, “the task”) lo que permite la programación en paralelo y manejo de excepciones. Ada es un lenguaje que se utiliza en la actualidad, sin embargo, pese a lo robusto del lenguaje, no tuvo el éxito que se esperaba, ya que al ser concebido para cuestiones militares se establecieron múltiples controles con el fin de evitar su uso indebido.

El Modula-2 es un lenguaje de programación que fue desarrollado por Niklaus Wirth como sucesor de Pascal y fue publicado en 1982. Uno de los objetivos de Modula-2 era que fuera un lenguaje simple, lo que permitió que fuera utilizado en computadoras pequeñas. Este lenguaje cuenta con un mecanismo conocido como el “módulo”, que sería el equivalente al “paquete” de Ada, aunque con algunas restricciones. También es posible implementar concurrencia con este lenguaje, tiene un soporte limitado para tipos de datos abstractos y no tiene un mecanismo para el manejo de excepciones, lo que provocó que no fuera útil para el desarrollo de aplicaciones grandes.

Si bien es cierto que Ada y Modula-2 fueron lenguajes que no tuvieron el éxito esperado por algunas de sus características, también es cierto que hubo otros factores que influyeron para que esto sucediera, entre lo que destaca el crecimiento de la programación orientada a objetos. En 1980 aparece en el mercado la última versión de Smalltalk y tiene lugar el acelerado crecimiento de la utilización de C++. El lenguaje C++ es una extensión de C que incorpora el paradigma orientado a objetos. Tanto el Smalltalk como el C++ son lenguajes representativos del paradigma orientado a objetos y se basaron en el concepto de clases del lenguaje Simula67.

El Smalltalk se desarrolló entre 1972 y 1980 por Alan Kay y Dan Ingalls, que formaban parte de un grupo del Centro de Investigación Palo Alto de la corporación Xerox. Este lenguaje se distingue por adoptar completamente el paradigma orientado a objetos y proporcionar una interfaz gráfica, con la cual el usuario podía interactuar de manera más amigable con el sistema. Para que esto fuera posible, Smalltalk tenía que funcionar sobre un sistema operativo y *hardware* en particular, lo que implicó en aquella época, que este sistema no se difundiera masivamente, ya que no todas las personas contaban con dicho sistema.

Sin embargo, aunque Smalltalk no conquistó el mercado mundial, sí mostró algunas virtudes del paradigma orientado a objetos y dio impulso al uso del lenguaje C++. El lenguaje C++ fue desarrollado por Bjarne Stroustrup en los Bell Labs en 1980. Se basa en el lenguaje C, que ya para entonces tenía popularidad, lo que le dio transportabilidad y permitió que este lenguaje se pudiera ejecutar, prácticamente, sobre cualquier plataforma.

El C++ incorpora del Simula67 el concepto de clases, lo que al principio le valió el nombre de “C con Clases” y la construcción de una gran cantidad de bibliotecas provocó que este lenguaje fuera utilizado para el desarrollo de una extensa variedad de aplicaciones, a pesar de ser un lenguaje complejo para su entendimiento en comparación con algunos otros lenguajes. De esta forma, creció la programación orientada a objetos propiciando la creación de diferentes lenguajes bajo este modelo como son: Objetive C, Object Pascal, Modula-3, Oberon e Eiffel.

Sin embargo, pese al rápido crecimiento de la programación orientada a objetos en la década de 1980, también surgió el interés por el *paradigma funcional o aplicativo*. Bajo este esquema, los lenguajes adoptan el modelo matemático de función con el fin de representar la manera en que ejecuta las instrucciones una computadora; es decir, los programas. Matemáticamente, el concepto de función asocia a una variable con otra y los valores que resultan de efectuar una función son los que se utilizan para el cálculo de la siguiente función, por lo que dichos valores no tienen que ser almacenados y, de esta manera, los lenguajes funcionales se olvidan del concepto de localidad de memoria y, por consecuencia, del concepto de variable como localidad de memoria, así como de la operación de asignación y de los ciclos por medio de funciones iterativas, para ello se utiliza la recursión como una elegante forma de realizar cualquier cantidad de ciclos.

Scheme es un lenguaje funcional que fue desarrollado entre 1975 y 1978 por Gerard J. Sussman y Guy L. Steel en el MIT, con base en el lenguaje Lisp. Otro lenguaje funcional que registró un creciente uso fue ML (*Metalanguage*), que fue desarrollado por Robin Milner en la Universidad de Edimburgo. ML se diferencia de otros lenguajes funcionales en su sintaxis, que es más parecida al lenguaje imperativo Pascal. Un lenguaje que tiene relación directa con ML es Miranda, que fue desarrollado por David Turner en la Universidad de Manchester entre 1985 y 1986.

En términos generales, los lenguajes funcionales comenzaron a sustituir a los lenguajes imperativos y John Bakus desarrolló un lenguaje funcional llamado Fp (Functional Programming). Por otros caminos se hacían intentos por desarrollar lenguajes de acuerdo a la lógica matemática. En este sentido, el lenguaje más sobresaliente es Prolog; en particular, el desarrollo del Prolog de Edimburgo marcó una gran diferencia en lo que a publicidad se refiere, al ser escogido como el lenguaje a utilizar en el proyecto de Japón conocido como la 5ta generación. Este proyecto no llegaría a cumplirse, sin embargo el lenguaje Prolog ha mantenido su aceptación en el desarrollo de aplicaciones de *inteligencia artificial* y ámbitos académicos.

Con todos estos avances en el área de los lenguajes de computación, el crecimiento de los lenguajes orientados a objetos, los sistemas operativos con interfaces gráficas y las fascinantes oportunidades que ofrecían los nuevos mercados, comenzaron las competencias a través del mercadeo (*marketing*). Y sin embargo, lo más impresionante estaría por suceder: la expansión de la Internet.

En la década de 1990 aparece en el mercado Mosaic, un navegador de la Internet que inmediatamente fue aceptado y vendría a ser parte de lo que ahora es todo un “mundo virtual”. En la misma primera mitad de esa década, C++ parecía ser el lenguaje seleccionado por los desarrolladores, ya que nadie esperaba la introducción de Java.

Java fue desarrollado por James Gosling de la empresa Sun Microsystems. Originalmente fue desarrollado para aplicaciones en sistemas *incrustados* (*embedded systems*) y su nombre era Oak. El acelerado crecimiento de la Internet fue aprovechado por Sun Microsystems para introducir en el mercado a Java como un lenguaje orientado al desarrollo de redes y de aplicaciones de la Internet.

Java es un lenguaje orientado a objetos, interpretado, con la posibilidad de ser transportado entre los diferentes sistemas operativos, con una buena cantidad de bibliotecas disponibles que le permiten hacer aplicaciones gráficas de manera sencilla, permitiendo desarrollos de redes y con utilidades de concurrencia. Estas cualidades y las ventajas que ofrece el mercadeo hicieron de Java el lenguaje por excelencia en el desarrollo de aplicaciones ordinarias, de red y de la Internet, de sistemas incrustados y para le enseñanza en las Universidades.

A finales de la década de 1990 fue desarrollado Haskell, nombrado en honor del matemático Haskell B. Curry. Con un esfuerzo internacional, su investigación y su desarrollo fueron apoyados por las Universidades Yale y de Glasgow. Haskell es un lenguaje funcional y considerado como la punta en la evolución de los lenguajes por las personas que utilizan este lenguaje para sus desarrollos, principalmente matemáticos que encuentran en este lenguaje la herramienta ideal para plasmar sus ideas en las computadoras de manera directa. Haskell es parecido al ML y al Miranda, cuenta con un creciente número de bibliotecas, capacidades de interfaz y características más sofisticadas. Gufer y Hugo son otros lenguajes relacionados con Haskell.

También de esta década es Ada95, que es una versión mejorada de Ada83, que adiciona el paradigma orientado a objetos y la programación en paralelo. Con estas características se espera que Ada95 siga siendo un lenguaje actual.

Una diferencia importante entre los lenguajes de esa década y los de las décadas anteriores es el uso de las *bibliotecas*. En Java, por ejemplo, el uso de las API (*Aplication Program Interface)* es lo que le da su fuerza; mientras que en los lenguajes como C o Pascal, las bibliotecas simplemente constituían un elemento no relevante. De esta misma forma, lenguajes como Haskell, Ada95 y Java contienen conjuntos crecientes de bibliotecas que permiten una gran versatilidad en el desarrollo de aplicaciones.

Con el creciente uso de bibliotecas se han hecho presentes de manera significativa lenguajes *script* (*scripting languages*), que fueron desarrollados con el fin de "pegar" aplicaciones [55]. Algunos de estos lenguajes son: AWK, Perl, Tcl, Javascript, Rexx y Python. Algunas de las características que distinguen a estos lenguajes son: permiten hacer programas pequeños, dinámicos en su comportamiento, pueden ser construidos a partir de estructuras de datos de alto nivel, son de sintaxis que se puede entender y no tienen control al respecto del tipo de datos. Jcl (*Job Control Language*) es un lenguaje *scripting* que funcionaba en computadoras de IBM en la década de 1960. Otros lenguajes de este tipo son los "*shell*" de UNIX y el lenguaje de comandos del DOS (Disk Operative System), sin embargo, éstos tenían operación y alcance limitados. En el caso del AWK, Perl y Tcl fueron desarrollados para suplir dichas limitaciones del *shell* de UNIX. El Javascript es un lenguaje que se utiliza de manera incrustada (*embedded*) y que permite el manejo dinámico de la información en las páginas web.

1.2 GRAMÁTICAS

El uso de las diversas tecnologías de la información ha mostrado la utilidad de modelar la realidad. La creatividad del ser humano ha dado múltiples giros a través de la historia creando y mejorando herramientas que permiten que los modelos sean cada vez más precisos y parecidos a la realidad. Una de estas herramientas, que ha servido para obtener modelos en los que la línea que divide a la ciencia del arte desaparece, ha sido desarrollada por los lingüistas con el fin de describir los lenguajes, las *gramáticas*.

Desde el punto de vista lingüístico, las gramáticas son una herramienta que se desarrolló con el fin de describir lenguajes, enfocándose primordialmente a los lenguajes utilizados por los seres humanos para comunicarse entre sí, lo que se conoce como lenguaje natural y una de sus corrientes de estudio es el "estructuralismo"[23][34][35]. Cuando un lenguaje es correctamente empleado desde el punto de vista de su estructura, se dice que es sintácticamente correcto. La sintaxis se encarga de verificar que las oraciones de un lenguaje estén correctamente estructuradas, por ejemplo, la sintaxis del español indica que una oración se forma de sujeto y predicado, a su vez el sujeto puede estar formado por artículo, sustantivo, adjetivo y el predicado por verbo, adverbio, etc. Por otro lado, la gramática contiene las reglas de cómo se forman las palabras; por ejemplo en español para formar el plural, a las palabras se les agrega una *s* al final (pluma-pluma**s**), aunque existen excepciones a la regla. Así que un lenguaje definido contiene una serie de reglas sintácticas en las que se describen las distintas formas en las que se pueden hacer las oraciones y una serie de reglas gramaticales que definen la manera en la que se deben formar las palabras.

A partir de 1950, el área de la ciencia dedicada al estudio de la lingüística sufre un cambio radical con la introducción de la teoría de Noam Chomsky [43] referente a la g*ramática universal*. La evolución que ha sufrido la teoría de Chomsky y la aplicación que se le ha dado en el desarrollo de lenguajes de programación, ha afectado de manera significativa el desarrollo de las gramáticas, ya que se encontró una manera de representar las gramáticas de forma tal que pudieran ser manipuladas matemáticamente, dando la base para toda una rama de conocimiento conocida como *lingüística matemática*.

Un área en la que se aplicaron directamente las ideas de la teoría de Chomsky fue en la del desarrollo de lenguajes de programación, en donde las *gramáticas formales* son empleadas en la construcción de compiladores y diversas herramientas capaces de reconocer y generar lenguajes; a este respecto se ha hecho amplio uso de las *gramáticas formales* como sistemas generadores de lenguajes [10][12]. Una gramática formal es *una "receta" exacta, de tamaño definido para construir las oraciones de un lenguaje* [2]; asumiendo que un lenguaje es un conjunto infinito de oraciones que se forman a partir de un número finito de símbolos, conocido como el alfabeto. Un lenguaje puede ser visto como una cadena infinita de bits y a partir de las reglas de su gramática debe ser posible formar cada uno de los subconjuntos de bits o cadenas parte del lenguaje [1].

Uno de los retos que enfrentan los investigadores dedicados al estudio de los lenguajes, es encontrar lenguajes que permitan describir a los objetos de la realidad. Posteriormente, encontrar las gramáticas que describen a dichos lenguajes y crear modelos, de manera que se puedan desarrollar sistemas basados en estas gramáticas. De acuerdo a las características inherentes a los objetos de la realidad y por consecuencia a los distintos lenguajes que se utilizan para describir dichos objetos, se han desarrollado distintos tipos de gramáticas, como son las sensibles al contexto, las libres de contexto o las regulares .

En estos términos, hablar de gramáticas representa una interminable cantidad de conocimiento. Una manera de entender las gramáticas es a partir de la clasificación conocida como "jerarquía de Chomsky", en donde las gramáticas son clasificadas en tipo 0 (irrestrictas), tipo 1 (sensibles al contexto), tipo 2 (libres de contexto) y tipo 3 (regulares) [10][11][12][16].

Estas gramáticas son frecuentemente catalogadas como más poderosas o menos poderosas; lo cierto es que el tipo de gramática a utilizar depende del lenguaje que se quiere describir. En términos generales, de acuerdo a las características del objeto a modelar, es el tipo de gramática a utilizar. Por esta razón, el hecho de que sean más poderosas unas que otras responde, finalmente, a las restricciones que se les imponen con el fin de modelar cierto tipo de lenguajes, aunque las gramáticas libres de contexto son ampliamente utilizadas y han sufrido una evolución que ha resultado en múltiples variantes de este tipo de gramáticas [10][11][12][16].

1.2.1 Teoría de Gramáticas formales

Con los estudios de la lingüística matemática se ha elaborado toda una teoría sobre la que se sustentan los trabajos referentes a lenguajes y gramáticas, se hace referencia a este conocimiento como *lenguajes formales*. *La descripción formal de un lenguaje es conocida como gramática formal*.

Un lenguaje es un conjunto de oraciones y una oración es un conjunto de símbolos. La gramática es un conjunto de reglas que describen dicho lenguaje y una *gramática formal* es aquella que permite generar todas las oraciones del lenguaje [2]. En la descripción de lenguajes de programación las gramáticas formales son de gran utilidad, ya que permiten generar o reconocer matemáticamente un lenguaje. En este escrito se hará referencia a gramática o gramática formal indistintamente.

Un lenguaje puede ser visto como una secuencia de símbolos o bits que mantienen cierta estructura. Por medio de la gramática es posible reconocer los símbolos y describir dicha estructura. Los símbolos que aparecen en esta secuencia representan el alfabeto del lenguaje y en la gramática forman el conjunto de los símbolos terminales (Vt).

En una gramática se utiliza otro conjunto de símbolos, conocido como el conjunto de los símbolos no-terminales (Vn). Como su nombre lo indica, cuando uno de estos símbolos aparece en la gramática indica que la cadena no termina, sino que debe ser sustituido por otros símbolos que pueden ser terminales o no-terminales.

Una *gramática* "G" es una cuádrupla (Vn,Vt, S, R), en donde Vn y Vt son conjuntos finitos de símbolos. Vn es el conjunto de símbolos no-terminales y Vt es el conjunto de los símbolos terminales o alfabeto del lenguaje. R es un conjunto de reglas de "producción", "transición" o "reescritura", razón por la que las gramáticas formales también son conocidas como sistemas de reescritura. S es la oración inicial y debe pertenecer al conjunto de los símbolos no-terminales, a partir de este símbolo se va a generar el lenguaje en *n* transiciones, de acuerdo a las reglas de reescritura.

Convención: En este escrito los símbolos terminales de una gramática serán representados por letras o palabras en minúsculas y los no-terminales por letras mayúsculas o palabras que empiecen con letra mayúscula.

Por ejemplo, sea el lenguaje L={aa, ab, baaa, baab, bbb}. Una gramática que genera este lenguaje es G = { Vn, Vt, S, R }, en donde

$$Vn = \{S, A, B\}$$

$$Vt = \{a,b\}$$

Y cuyas reglas de producción o reescritura R son:

$$1)\ S \rightarrow aA \mid bB$$

$$2)\ A \rightarrow a \mid b$$

$$3)\ B \rightarrow aaA \mid bb$$

El símbolo "$\rightarrow$" debe leerse como "produce a" y el símbolo "|" significa "o". Con lo cual, las reglas anteriores se leen: 1) *S* "produce a" *aA* "o" *bB*; 2) *A* "produce a" *a* "o" *b*; y 3) *B* "produce a" *aaA* "o" *bb*.

El lenguaje *L* se genera a partir del símbolo inicial *S* sustituyendo los símbolos no-terminales por terminales o más no-terminales de acuerdo a las reglas de transición, hasta que se obtenga una cadena que contenga solamente símbolos terminales. Siguiendo las reglas: *S* produce a *aA* o *bB*. Sustituyendo *S* por la primera opción se obtiene la cadena *aA*. Ahora se deberá sustituir el símbolo *A* por alguna de las alternativas de la segunda regla de producción, es decir *a* o *b*. Con la primera opción se genera la cadena *aa* y dado que esta cadena está formada únicamente por símbolos terminales, entonces ya se habrá terminado el proceso y se podrá comprobar que la cadena *aa* pertenece al lenguaje *L*. Con la segunda opción se habrá generado la cadena *ab*, que también pertenece al lenguaje *L*.

Regresando a la primera regla de producción y en caso de que se elija sustituir el símbolo inicial *S* por la segunda opción *bB*, las cadenas que se generan al sustituir el símbolo *B* de acuerdo a la tercera regla de producción son: *baaA* o *bbb*. Nuevamente se tendrá que elegir alguna de las estas dos cadenas. En el primer caso *baaA* se tendrá que sustituir el símbolo *A* por alguna de las opciones de la segunda regla de reescritura, obteniéndose *baaa* o *baab*, que son cadenas que contienen únicamente símbolos terminales y pertenecen al lenguaje *L*. Finalmente, si se elige la segunda opción de la tercera regla de transición se obtendrá la cadena *bbb* que pertenece al lenguaje *L*. De esta manera, se ha mostrado que por medio de la gramática *G* se han generado todas las cadenas que pertenecen al lenguaje *L*, por lo que se dice que la gramática *G* genera al lenguaje *L*.

1.2.1.1 Gramáticas Irrestrictas Tipo 0

Las gramáticas irrestrictas, tienen reglas de producción de la forma $\alpha \rightarrow \beta$ donde α y β son cadenas arbitrarias de símbolos, siendo $\alpha \neq \lambda$ [63], en donde λ representa la cadena vacía, la cadena que no contiene ningún símbolo. Las gramáticas irrestrictas son aquéllas cuyas reglas de producción no tienen restricciones, lo que implica que pueden contener reglas con cualquier cantidad de símbolos terminales y no-terminales, tanto del lado derecho como del izquierdo, lo que les da la posibilidad de transformar cualquier cantidad de símbolos en una menor cantidad de ellos o incluso ninguno. Por ejemplo, la regla $abAB \rightarrow aC$ que produce dos símbolos a partir de cuatro.

Debido a que los otros tipos de gramáticas de la jerarquía de Chomsky resultan de aplicar restricciones a la manera de expresar las reglas de producción, cualquier lenguaje que pueda ser descrito mediante una gramática regular, libre de contexto o sensible al contexto, también puede ser descrito mediante una gramática irrestricta. En otras palabras, las gramáticas regulares, libres de contexto y sensibles al contexto están contenidas en las gramáticas irrestrictas [63][65]. Sin embargo, la falta de restricciones hace que su manejo resulte complicado en ámbitos computacionales.

1) $S \rightarrow ACaB$	5) $aD \rightarrow Da$
2) $Ca \rightarrow aaC$	6) $AD \rightarrow AC$
3) $CB \rightarrow DB$	7) $aE \rightarrow Ea$
4) $CB \rightarrow E$	8) $AE \rightarrow \lambda$

Fig. 2.1 Gramática Irrestricta que genera el conjunto $\{a^i \mid i$ es potencia positiva de $2\}$.

1.2.1.2 Gramáticas Sensibles al Contexto Tipo 1

Las gramáticas sensibles al contexto resultan de restringir a las gramáticas irrestrictas, pues la longitud de las cadenas del lado derecho de las reglas de producción, tiene que ser mayor o igual que la del lado izquierdo. Una gramática $G = (Vn, Vt, S, R)$ es sensible al contexto si sus reglas de producción son de la forma: $\alpha A\beta \rightarrow \alpha\delta\beta$, en donde los símbolos α y β representan cadenas de símbolos sobre el alfabeto $V = Vt \cup Vn$, $A \in Vn$ y δ es una cadena no vacía sobre V, con la posible excepción de la producción $S \rightarrow \lambda$ (cadena vacía), cuya ocurrencia en R implica que S no ocurre en el lado derecho de ninguna de las reglas de producción en R [1]. A las cadenas α y β se les denomina el contexto del símbolo no-terminal A, ya que sólo puede ser sustituido por la cadena δ, cuando se encuentra presente dicho contexto [63].

En las gramáticas sensibles al contexto, los símbolos no-terminales que se encuentran en el lado izquierdo de las reglas de producción, dependen de las relaciones que tienen con los símbolos que los rodean. Por ejemplo, en la regla de producción: aAb → acdeb, el símbolo no-terminal *A* produce la cadena *cde*. Los terminales *a* y *b*, en este caso representan el contexto. En la fig. 2.2 se presenta una gramática sensible al contexto que genera el lenguaje $\{a^n b^n c^n \mid n >= 1\}$.

1) S → abc | aAbc
2) Ab → bA
3) Ac → Bbcc
4) bB → Bb
5) aB → aaA | aa

Fig. 2.2 Gramática Sensible al Contexto que genera el lenguaje $\{a^n b^n c^n \mid n >= 1\}$.

1.2.1.3 Gramáticas Libres de Contexto Tipo 2

Las gramáticas libres de contexto tienen más restricciones y por lo tanto son menos poderosas que las gramáticas sensibles al contexto, lo que significa que hay lenguajes tipo 1 y tipo 0 que no se pueden describir con gramáticas libres de contexto. Sin embargo, son lo suficientemente poderosas como para ser utilizadas para describir lenguajes computacionales. Su aplicación en cuestiones de programación es directa y más sencilla que los otros tipos de gramáticas, razón por la que han sido las gramáticas con mayor desarrollo y sus aplicaciones son variadas.

A diferencia de las gramáticas sensibles al contexto, y como su nombre lo indica, las gramáticas tipo 2 se distinguen por la ausencia de contexto, es decir, lo que produce cada símbolo no-terminal es independiente de lo que puedan producir los demás símbolos de las reglas de producción. Una consecuencia de esto, que caracteriza a las gramáticas libres de contexto, es que sólo pueden contener un símbolo no-terminal del lado izquierdo de las reglas de producción.

Una gramática libre de contexto es un conjunto de variables denominadas categorías sintácticas, cada una de las cuales representa un lenguaje [63]. El desarrollo de las gramáticas libres de contexto surgió por la necesidad de describir los lenguajes naturales, sin embargo, su amplio rango de aplicaciones les permitió a los científicos de la computación utilizarlas para describir lenguajes de programación, lo que se tradujo en un avance significativo en la construcción de compiladores.

Una gramática $G = (Vn, Vt, S, R)$ es libre de contexto si cada producción en R es de la forma $X \rightarrow \alpha$, donde $X \in Vn$ y $\alpha \in (Vn \cup Vt)$ [1]. Los lenguajes generados por medio de gramáticas libres de contexto incluyen a los lenguajes que pueden ser generados mediante gramáticas regulares, esto quiere decir que cualquier gramática regular es libre de contexto y como consecuencia las gramáticas libres de contexto son más poderosas que las regulares [64]. En la fig. 2.3 se muestra una gramática libre contexto que genera el lenguaje $\{a^n b^n \mid n>=1\}$.

1) $S \rightarrow aSb$
2) $S \rightarrow \lambda$

Fig. 2.3 Gramática Libre de Contexto que genera el lenguaje $\{a^n b^n \mid n >= 1\}$.

1.2.1.4 Gramáticas Regulares Tipo 3

Si todas las reglas de producción de una gramática libre de contexto son de la forma $A \rightarrow \alpha B$ o $A \rightarrow \alpha$, donde A y B son variables y α es una cadena de terminales (posiblemente vacía), se dice que la gramática es lineal-derecha. Si todas las reglas de producción son de la forma $A \rightarrow B\alpha$ o $A \rightarrow \alpha$ se trata de una gramática lineal-izquierda. A cualquiera de estos dos tipos de gramáticas (lineal izquierda o derecha) se les conoce como gramáticas regulares [63]. Las gramáticas regulares, también son conocidas como gramáticas de estados finitos [2]. Por ejemplo, el lenguaje 0(10)* se puede generar mediante la gramática lineal derecha: S$\rightarrow$ 0A, A $\rightarrow$ 10A | λ, o por medio de la gramática lineal izquierda: S $\rightarrow$ S10 | 0

De esta manera, en la jerarquía de Chomsky las gramáticas regulares son las más restringidas. Afortunadamente, muchos lenguajes pueden ser descritos mediante gramáticas regulares, por lo que en la práctica son de gran utilidad. En la fig. 2.4 se muestra una gramática regular que genera copias de la cadena *ab*.

1) S $\rightarrow$ aA
2) A $\rightarrow$ bB
3) B $\rightarrow$ aA | λ

Fig. 2.4 Gramática Regular que genera una o más copias del patrón *ab*.

1.3 COMPILADORES

1.3.1 Compiladores e Intérpretes

Los compiladores son herramientas que hicieron su aparición hace 50 años y han formado toda un área de estudio de la ciencia en la computación. Incluso, se puede hablar de la creación de múltiples empresas que deben su existencia al desarrollo de compiladores.

Un compilador es un traductor que comprende la sintaxis y gramática de un lenguaje de programación de alto nivel conocido como programa fuente y produce un texto en lenguaje de máquina conocido como programa objeto (fig. 3.1), con el fin de que se puedan desarrollar programas utilizando un lenguaje más claro y amigable para el programador y que otros programadores entiendan los programas desarrollados, estos programas contienen una secuencia de instrucciones y generalmente constituyen lo que se conoce como el *software* de la computadora. Sin embargo, las computadoras sólo entienden el lenguaje binario pues están formadas únicamente de circuitos a lo que generalmente se le denomina el *hardware* de la computadora. Así que el desarrollo de un traductor o compilador involucra diversas áreas de conocimiento como son: la teoría de los lenguajes de programación, las gramáticas, la arquitectura de computadoras y el desarrollo de algoritmos.

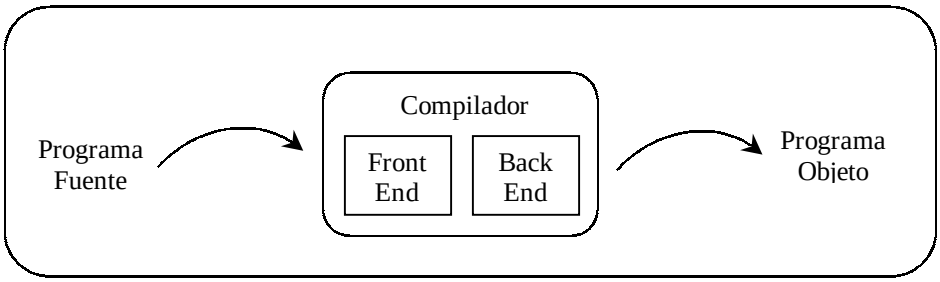

Fig. 3.1 Diagrama General de un Compilador

Debido a la multitud de sistemas computacionales electrónicos desarrollados comercialmente, se presenta un problema importante a resolver: que el compilador sea independiente de la arquitectura de la computadora sobre la cual es desarrollado, es decir, que pueda ser ejecutado sobre equipos computacionales distintos. Una solución que se ha adoptado es desarrollar el compilador en dos partes denominadas “front end”, que analiza el programa fuente y “back end”, que se encarga de generar código para la máquina objeto (máquina sobre la que se va a ejecutar el producto del compilador) fig. 3.2. De esta manera sólo basta modificar el *back end* para transportar el compilador a otra máquina objeto.

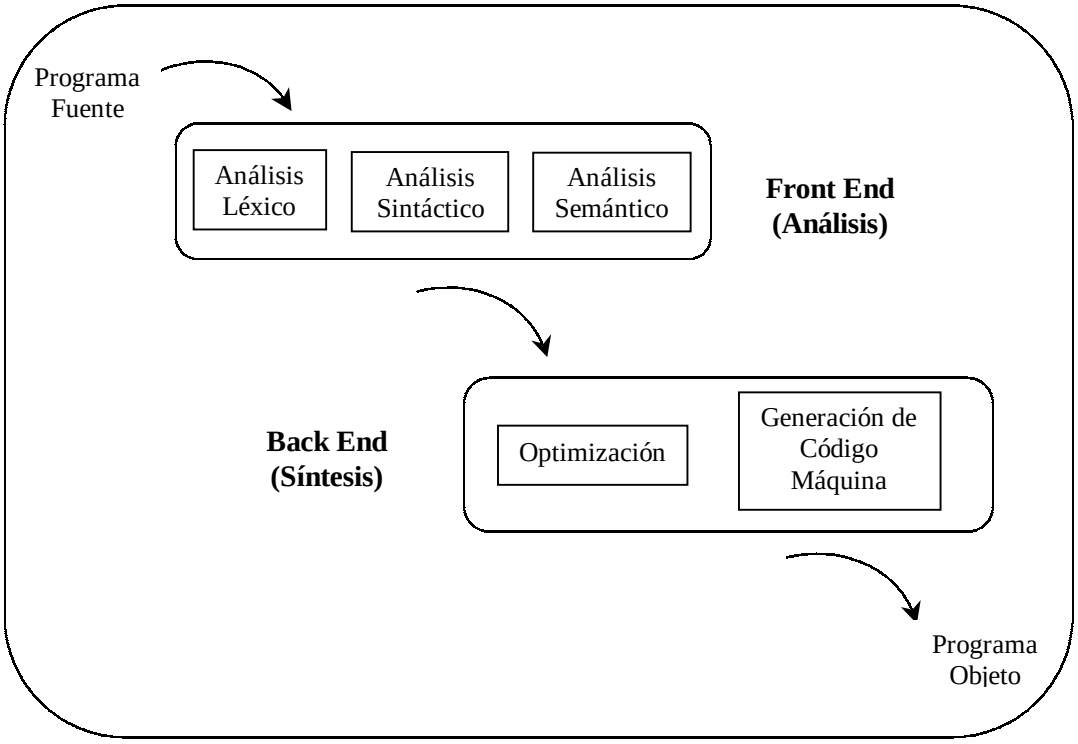

Fig. 3.2 Fases de un Compilador

Otra forma en la que la evolución de las técnicas de compilación ha resuelto este problema de transportabilidad, es desarrollando *intérpretes*. Los intérpretes son traductores que en lugar de generar código (lenguaje máquina) para una arquitectura en particular, simplemente ejecutan las instrucciones sobre una "máquina virtual". De manera similar al *front end*, la máquina virtual genera toda una estructura de datos mediante *software* independiente de la arquitectura física (*hardware*), permitiendo con esto que el lenguaje interpretado pueda ser ejecutado sobre diversos equipos. Los intérpretes generalmente son sistemas más pequeños que los compiladores y son más sencillos de desarrollar, ya que al no generar código máquina se evita todo el desarrollo de síntesis (back end). Por otro lado, cuando se desarrolla un intérprete es necesario pensar sólo en un lenguaje, el lenguaje fuente, mientras que en el desarrollo de compiladores es necesario pensar en el lenguaje fuente y en el lenguaje objeto [52].

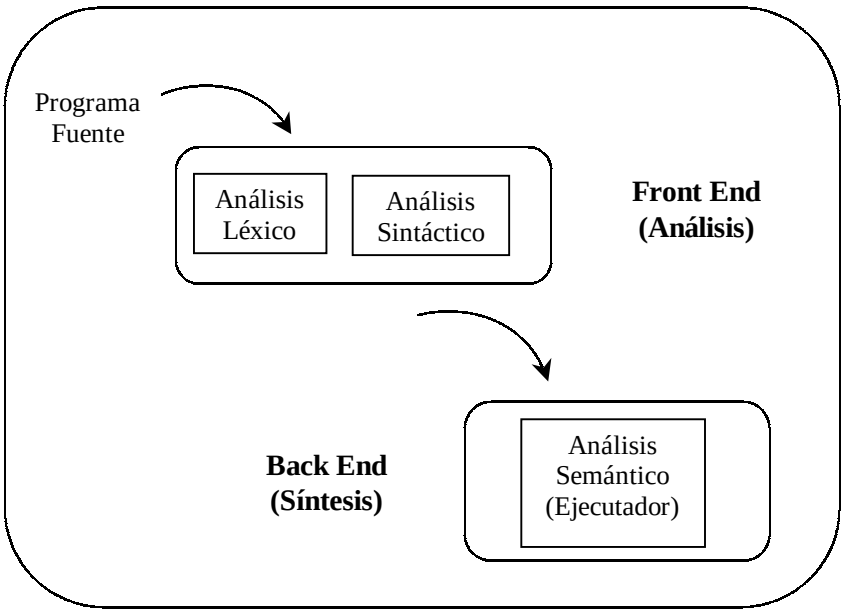

Fig. 3.2 Fases de un Intérprete

Los compiladores necesitan traducir un lenguaje fuente en un lenguaje objeto y generar un programa ejecutable. Para poder ver el funcionamiento del programa generado hace falta ejecutarlo. En cambio, un intérprete no tiene que generar el programa ejecutable y simplemente ejecuta las instrucciones directamente. Esto permite a los intérpretes ser más versátiles, pues el código generado por los compiladores depende del hardware [53]. Otra razón por la que un compilador plantea algunos problemas es la cuestión del manejo de errores. Algunos errores, como dividir un número entre cero, se pueden detectar sólo en tiempo de ejecución. En el caso de un compilador esto significa que si existe un error de esta naturaleza hay que solucionar el problema y volver a compilar el programa, es decir, llevar a cabo todo el proceso de generación de código y nuevamente ejecutar el programa para corroborar que el error fue corregido. Por esta razón, este tipo de errores es más fácil solucionarlos en el caso de los intérpretes.

En tiempos pasados se prefería el desarrollo de lenguajes compilados sobre los lenguajes interpretados ya que los intérpretes son más lentos, lo cual es una buena razón pues un lenguaje compilado es de 10 a 100 veces más rápido que uno interpretado [53]. Sin embargo, con el avance de las tecnologías de la información los tiempos de ejecución de los procesadores se miden en nanosegundos, por lo que en la vida cotidiana estas diferencias no afectan de manera significativa. Por lo tanto, no es de extrañar que Java, un lenguaje interpretado sea el estándar que marca la pauta, actualmente, en cuanto a desarrollo de software se refiere.

Con el uso extendido de la Internet y las diferentes plataformas existentes, el empleo del intérprete de Java resolvió muchos de los problemas de transportabilidad entre dichas plataformas. De esta manera, un programa desarrollado en Java puede ser ejecutado sobre Windows, MacOSX o UNIX, lo que hace no muchos años se vislumbraba como algo imposible.

El desarrollo de un compilador o un intérprete no es un trabajo fácil, pero puede ser realizado de manera modular, ya que generalmente está estructurado en varias partes: En resumen, en el funcionamiento de un compilador o intérprete, se llevan a cabo tres procesos básicos: 1) Comprobar que el símbolo leído pertenezca al vocabulario del lenguaje, en la construcción tradicional de lenguajes esta tarea se le asigna al analizador léxico. 2) Se debe cumplir que dicho símbolo esté en la posición adecuada es decir, debe asegurarse que se cumpla la estructura definida para el lenguaje, tradicionalmente esto lo realiza el analizador sintáctico. 3) Finalmente, se debe llevar a cabo la acción asociada a la instrucción, comando, palabra o símbolo, cuestión que desempeña el analizador semántico. La manera en la que estas funciones de un compilador o intérprete se lleven a cabo varía, como ya se mencionó, tradicionalmente los compiladores se han construido sobre la base de estos tres analizadores (léxico, sintáctico y semántico) sin embargo, con el avance del área de construcción de compiladores no siempre se sigue este modelo.

Este trabajo consiste en el desarrollo de un intérprete que permite reconocer un lenguaje a partir de la definición de su gramática. Para ello se han implementado los analizadores léxico y sintáctico formando las partes fundamentales de este intérprete. Debido a que ésta es una herramienta general con la que es posible reconocer cualquier lenguaje que pueda ser descrito por medio de una gramática libre de contexto, no es la intención implementar el analizador semántico, pues cada lenguaje tiene funciones semánticas diferentes y pretender interpretar las funciones semánticas de cualquier lenguaje sería demasiado ambicioso y se encuentra fuera del alcance de este proyecto.

1.3.2 METACOMPILADORES

Los metacompiladores o compiladores de compiladores son sistemas que generan de manera automática el compilador de un lenguaje y se han desarrollado con el fin de facilitar el desarrollo de compiladores. La manera en la que funcionan es, generalmente, utilizar la descripción de un lenguaje a través de su gramática para construir su compilador. Se basan en el hecho de que por medio de la gramática se describen las reglas que deben cumplir los lenguajes, es decir, el alfabeto que comprenden y la estructura con la que se deben construir las oraciones del lenguaje.

La mayoría de los metacompiladores disponibles a nivel comercial sólo generan analizadores léxicos o analizadores sintácticos, dejando a los programadores la tarea de desarrollar las demás fases del desarrollo de un compilador. Para la generación automática de analizadores léxicos utilizan gramáticas regulares y para la generación de analizadores sintácticos utilizan gramáticas libres de contexto.

Los analizadores semánticos deben construirse de manera particular para cada lenguaje, ya que las acciones semánticas a ejecutarse son completamente diferentes entre un lenguaje y otro. Por ejemplo, no es lo mismo desarrollar acciones semánticas para un lenguaje que efectúa operaciones en notación algebraica (a+b/c) que en otro que las lleva a cabo en notación polaca inversa (Reverse Polish Notation, RPN) ab+c/. De esta manera, no es tan sencillo determinar los significados que cada autor le da a las oraciones de su lenguaje y por lo tanto la mayoría de los metacompiladores sólo ayudan en las fases de análisis léxico y sintáctico del desarrollo de compiladores. A continuación se describen algunos de los compiladores de compiladores disponibles.

Lex (A Lexical Analyzer Generator) es un sistema que ayuda a escribir programas cuyo control de flujo es dirigido por instancias de expresiones regulares. Es idóneo para la transformación de tipos de edición *script* y para segmentar la entrada de cadenas en la preparación para el análisis sintáctico. El código fuente de Lex es una tabla de expresiones regulares y fragmentos de programas correspondientes. La tabla es traducida al programa que lee la entrada, particionándola en cadenas que concuerdan con las expresiones regulares. Cada vez que es reconocida una cadena se ejecuta el programa asociado. El reconocimiento de las expresiones es desempeñado por un autómata finito determinístico generado por Lex. Los fragmentos de programas son escritos por el usuario y ejecutados en el orden en que ocurren las expresiones regulares en el lenguaje [60].

Yacc (Yet Another Compiler-Compiler) provee una herramienta general para describir la estructura de un programa de computadora. El usuario especifica las estructuras de su entrada, junto con el código para ser invocado cuando cada estructura sea reconocida. Yacc genera una función para controlar el proceso de entrada de acuerdo a la especificación. Yacc está desarrollado en el lenguaje C, así como las acciones las subrutinas de salida y la mayoría de las convenciones sintácticas. El análizador sintáctico implementa el método LALR(1) [60].

Accent es un compilador de compiladores que permite utilizar cualquier tipo de gramática. Soporta la definición de gramáticas en la forma Backus Naur extendida y permite la especificación de atributos sintetizados y heredados.

Aflex y Ayacc son herramientas similares al Lex y Yacc desarrollados para UNIX, con la diferencia que fueron diseñados para generar código en el lenguaje Ada. Fueron desarrollados en proyecto Arcadia en la Universidad de California. Ambos sistemas son de distribución libre.

ALE (Attribute-Logic Engine) es un sistema de los denominados freeware y fue desarrollado por Bob Carpenter y Gerald Penn utilizando el lenguaje Prolog. Integra el análisis sintáctico basado en estructuras de frases, la generación es dirigida semánticamente y emplea la programación lógica con restricciones, con las estructuras como términos.

AnaGram es un sistema para ejecutarse en plataformas Win32. Este metacompilador genera código en el lenguaje C o C++ de acuerdo al estándar ANSI compatible independiente de plataforma.

Cocktail es un conjunto de generadores de programas o herramientas para la construcción de compiladores que se puede utilizar en todas las fases de construcción de un compilador

COCOM también es conocido como el armamento ruso y es un conjunto de herramientas para ayudar a los desarrolladores de compiladores. Incluye el generador de analizadores sintácticos MSTA, que genera analizadores LR(k) y LALR(k).

Coco/R es un metacompilador que genera analizadores sintácticos recursivos descendentes y los respectivos analizadores léxicos utilizando gramáticas con atributos.

Coco/R para Java es una versión de COCO/R que produce analizadores léxicos y sintácticos en lenguaje Java.

Cogencee es un generador de compiladores para Delphi que fue desarrollado como una variante de Coco.

Cosy es un sistema para el desarrollo de compiladores flexible, ya que permite la generación en código C y Java. Permite a los desarrolladores de compiladores generar y configurar compiladores rápidamente.

Delphi Compiler Generator fue desarrollado para generar clases para el lenguaje Borland Delphi de analizadores léxicos y semánticos. Delphi Compiler Generator es parte de TPLex y TPYacc (the Pascal translations of Lex and Yacc) desarrollado por Albert Graef, con el fin de permitir la edición de archivos de plantillas, compilar las clases y asociarlas instantáneamente a los proyectos de Delphi.

DMS Toolkit (Automated Software Analysis and Modification) Tecnología de compiladores generalizada es utilizada para el análisis automático y la modificación en gran escala de sistemas en diversos lenguajes como: C, C++, Cobol, Java, Fortran, SQL, XML, etc. Permite el manejo de millones de líneas, miles de archivos y mezcla de lenguajes. Provee de analizador sintáctico, impresión con buena calidad, evaluación de atributos y traducción de programas fuentes entre diferentes lenguajes. Es útil en análisis de código, la documentación la generación de código y la migración de sistemas entre diferentes equipos.

EAG es un compilador de compiladores basado en las gramáticas Affix extendidas (Extended Affix Grammars).

Eli es un sistema que ofrece soluciones para la mayoría de las tareas que involucra la implementación de un lenguaje. Desde análisis estructuras utilizando herramientas similares a Lex y Yacc hasta el análisis de nombres, tipos y valores, el almacenamiento y traslado de estructuras y la producción de texto.

Gentle cubre todo el espectro en la construcción de compiladores. Soporta el reconocimiento de lenguajes, la definición de árboles sintácticos, la construcción de analizadores de árboles basada en el reconocimiento de patrones, la traducción entre programas fuente de diferentes lenguajes y la selección óptima de código para microprocesadores.

LISA es un sistema que genera tablas para analizadores léxicos LL(1) y analizadores sintácticos de expresiones regulares en forma Backus Naur.

Urgen es un generador de analizador sintáctico LALR, un generador léxico, constructor de tablas de símbolos, constructor de árboles sintácticos y generador de código intermedio, es decir es un generador compiladores de la fase *front-end*.

MyLang (MKS Lex y Yacc) es un generador léxico compatible con Lex y generador sintáctico compatible con Yacc para PC's.

PCCTS (The Purdue Compiler Construction Tool Set) es un proyecto que se desarrolló en la escuela de Ingeniería Eléctrica en la Universidad de Purdue. Es un sistema que consiste en un analizador sintáctico (ANTLR *Another Tool for Language Recognition*) LL(k) con un analizador léxico integrado (DLG DFA-based Lexical analyzer Generator) y un traductor de código fuente a fuente (SORCERER).

PRECC (A PREttier Compiler-Compiler) es un compilador de compiladores de búsqueda adelantada para gramáticas sensibles al contexto. Dichas gramáticas se especifican en la forma Backus Naur extendida y permite atributos sintéticos y heredados.

Ragel compila máquinas de estados finitos de lenguajes regulares en código ejecutable C. Permite la inserción de llamadas de funciones en cualquier punto para controlar el no-determinismo de las máquinas resultantes. Las máquinas de estados finitos de Ragel son cerradas bajo sus operadores, propiedad que permite que se puedan describir lenguajes regulares.

S/SL es un lenguaje de programación para construir compiladores. Incorpora secuencias, repeticiones y selección de acciones; entradas, comparaciones y salidas de tokens; salidas de señales de error; subprogramas y la invocación de operaciones semáticas.

ScanGen, LLGen, LARLGen son generadores de analizadores léxicos y analizadores sintácticos utilizando el método LL(1) y LALR(1). Introducidos en «*Crafting a Compiler*'' por Fischer & LeBlanc.

Visual Parse++ es una herramienta que provee interfaz visual que permite a cualquier programador aprender y utilizar tecnología de analizadores léxicos y semánticos interactivamente.

Yacc++ y la Librería del Lenguaje de Objetos es un re-escritor de Lex y Yacc. Algunas de sus características incluyen la herencia de clases de gramáticas, la integración de expresiones regulares en un analizador sintáctico LR y soluciones para incluir archivos, subcadenas de palabras clave, comentarios anidados, etc.

YOOC es un compilador de compiladores escrito enteramente utilizando el lenguaje Eifel.

Esta es una lista incompleta de metacompiladores y de hecho seguirá creciendo constantemente ya que aun no se han desarrollado generadores de compiladores que permitan obtener un compilador completo [57]. En general, los compiladores de compiladores son difíciles de utilizar y el lenguaje que utilizan, conocido como metalenguaje, es complicado para aprender y otro problema que presentan es la carencia de análisis de errores [61]. De esta manera, pese a que existe una gran cantidad de este tipo de herramientas disponibles, aún hay bastantes cambios que se pueden realizar para obtener una mejor eficiencia en el desempeño de estas herramientas.

CAPÍTULO II

FUNCIONAMIENTO Y CONSTRUCCIÓN DEL SISTEMA

2.1 ANÁLISIS

La función principal del sistema que se ha desarrollado a partir de esta investigación es la de un reconocedor de lenguajes, esto quiere decir que sólo basta con describir un lenguaje a través de su gramática para obtener de manera automática su reconocedor. Como ya se mencionó a este tipo de sistemas se les conoce como compiladores de compiladores o metacompiladores.

Debido a las diferencias que existen entre los lenguajes de programación, implementar las funciones semánticas en un compilador de compiladores debe hacerse de manera particular para cada lenguaje. Por esta razón, la mayoría de los metacompiladores se limitan a desempeñar únicamente el análisis léxico y sintáctico de los lenguajes. Esto es de gran ayuda cuando se desarrolla un lenguaje de programación y su compilador, ya que es una tarea compleja que hasta hace no muchos años involucraba a una buena cantidad de personas y se necesitaba un tiempo considerable para su desarrollo. Con la utilización de herramientas como ésta, el programador economiza tanto tiempo como esfuerzo y sólo le resta implementar las funciones semánticas, con la certeza de que su lenguaje está correctamente diseñado en cuanto a su estructura y léxico se refiere.

Convención: En la fig. 4.1 se muestra el significado de la simbología utilizada en los diagramas de flujo de datos que se presentan en este capítulo.

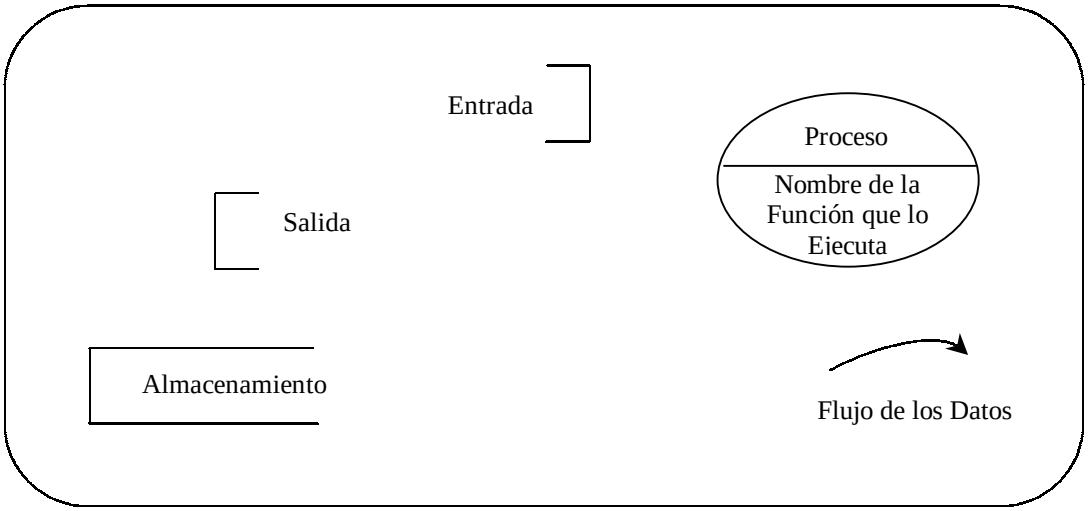

Fig. 4.1 Simbología utilizada en los Diagramas de Flujo de Datos.

El diagrama general del sistema se presenta en la fig 4.2, en donde se muestra que se tienen dos entradas al sistema, que representan dos archivos de texto. En uno de ellos se define la gramática del lenguaje, utilizando para ello la definición de las reglas de transición de gramáticas libres de contexto y en el otro, se escribe el programa fuente. Con la información contenida en estos archivos, el sistema se ejecuta verificando que el programa fuente se haya escrito correctamente, lo que significa que las oraciones están escritas utilizando únicamente símbolos que pertenecen al alfabeto del lenguaje y además se respeta la estructura del lenguaje, es decir, que se cumple con las reglas definidas por la gramática. Como salida se obtiene una aceptación del programa fuente, que implica que el programa está correctamente escrito o un rechazo en caso contrario.

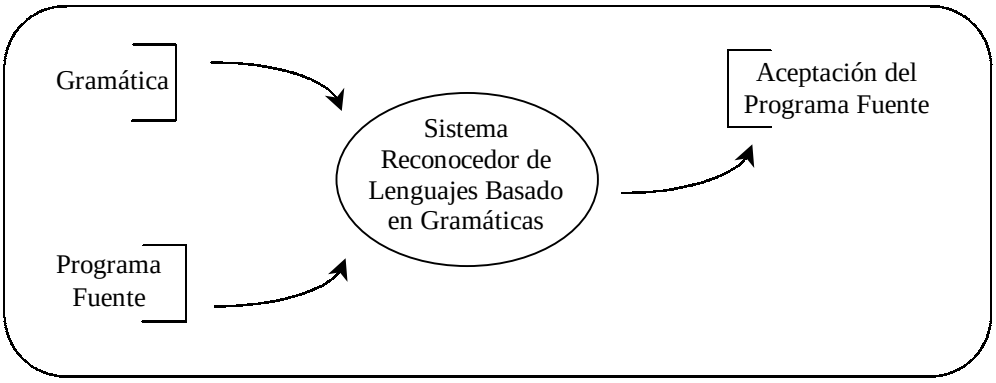

Fig. 4.2 Diagrama General del Sistema.

Para definir una gramática libre de contexto, generalmente, es necesario crear una lista en donde se establece cuales son los símbolos terminales y no-terminales del lenguaje. En el caso de este sistema no es necesario especificar cada uno de estos conjuntos de símbolos, es suficiente con definir las reglas de reescritura y el sistema se encargará de determinar cuáles es el conjunto de símbolos terminales (Vt) y el conjunto de símbolos no-terminales (Vn). Se reconoce a la *S* como el símbolo de inicio, así que en cualquier parte del archivo donde se encuentre la regla de producción S → ..., se tomará como el inicio de la gramática. Por ejemplo, la gramática que se presenta en la fig. 4.3 produce cadenas como: ad, abd, abbd, abbbd,... cysa, cxysa, cxxysa,...

1) S → aB | cCa

2) B → bB | d

3) C → ys | xC

Fig 4.3. Gramática Libre de Contexto

El archivo de texto que contiene el programa fuente escrito en el lenguaje objeto, debe cumplir con las reglas establecidas por la gramática. En forma paralela se va procesando tanto la gramática como el programa fuente, comprobando que las cadenas que forman las instrucciones del programa fuente cumplan con las reglas de producción de la gramática. El sistema se encargará de leer el programa fuente y marcará una bandera de error en caso de que el lenguaje utilizado en la elaboración del programa no pertenezca al lenguaje descrito en la gramática. En la gramática de la fig. 4.3, algunos programas fuente correctos serían las cadenas *ad, abbbd, cysa, cxxysa* y otros incorrectos serían *ac, aabbbd, caysa*.

A diferencia del funcionamiento tradicional de los compiladores, en donde primero se ejecuta el analizador léxico y después el sintáctico, en este sistema se llevan a cabo los dos análisis de acuerdo a la manera en la que aparecen los diferentes símbolos en la gramática. Primeramente se identifican los símbolos no-terminales y se marcan en las reglas de producción de la gramática. Al tratarse de gramáticas libres de contexto, este conjunto se encuentra representado por los símbolos que aparecen en el lado izquierdo de las reglas de producción.

Dado que cada símbolo no-terminal está asociado a cada una de las reglas de producción, es posible referirse a cada símbolo o regla de acuerdo a un índice, que obedece al orden en el que están escritas las reglas de producción. Por ejemplo, la gramática de la fig.4.3 se marcaría de la siguiente manera: La regla 1) $S \rightarrow aB \mid cCa$ contiene en su lado derecho dos símbolos no-terminales (B y C) y dado que estos corresponden a las reglas 2 y 3, respectivamente, entonces esta regla quedaría marcada como $S \rightarrow a<2> \mid c<3>a$. En la fig. 4.4 se muestra como quedaría marcada la gramática de la fig. 4.3.

1) S → a<2> | c<3>a

2) B → b<2> | d

3) C → ys | x<3>

Fig 4.4. Marcas que se agregan a la gramática de la fig. 4.3

Para comprobar que los programas fuente están correctamente escritos se utiliza la técnica de análisis recursivo descendente. Cada vez que en la gramática aparece un símbolo terminal, éste debe coincidir con el símbolo leído del programa fuente y cada vez que se encuentra un no-terminal se ejecuta de forma recursiva la función que permite evaluar la regla de producción asociada al símbolo no-terminal. De esta manera, se va comprobando que los símbolos leídos pertenezcan al alfabeto del lenguaje y además que se encuentren en la posición correcta, cumpliendo con la estructura de las oraciones del lenguaje.

2.2 DISEÑO

El funcionamiento de este sistema es llevado a cabo por 4 procesos, 3 de ellos son procesos auxiliares que se utilizan para hacer un preprocesamiento y almacenamiento tanto de la gramática, así como del programa fuente en arreglos de la memoria principal. El preprocesamiento que se realiza es con el fin de facilitar la ejecución del proceso principal y consiste en tareas tales como el borrado de blancos en el caso del programa fuente y la inserción de algunas marcas en la gramática, que sirven para identificar los diferentes símbolos de los que está constituida. Estos procesos son ejecutados respectivamente por las funciones: ProgramaFuente(), CargaGramDim2(), IndicaVn() y LeeGram().

Nota explicatorio: Los nombres de las funciones son de acuerdo a la construcción de nombres de funciones en C. Es decir, están formados por varias palabras que inician con mayúscula y sin espacio entre ellas, formando así una sola palabra y con paréntesis, para indicar que se está hablando de la función que utiliza el sistema para procesar la información.

2.2.1 Función ProgramaFuente()

El programa fuente o programa objeto es un archivo de texto en donde están contenidas las instrucciones que en su momento serán ejecutadas por la computadora. Estas instrucciones se deberán construir con oraciones que respeten las reglas del lenguaje y dichas reglas estarán definidas por medio de la gramática en otro archivo de texto.

La función ProgramaFuente() se encarga de almacenar en un arreglo llamado ProGram[] que se ubica en la memoria principal, el contenido del archivo de texto que representa el programa fuente escrito por el usuario. En la fig. 4.5 se muestra del diagrama de flujo de datos de esta función.

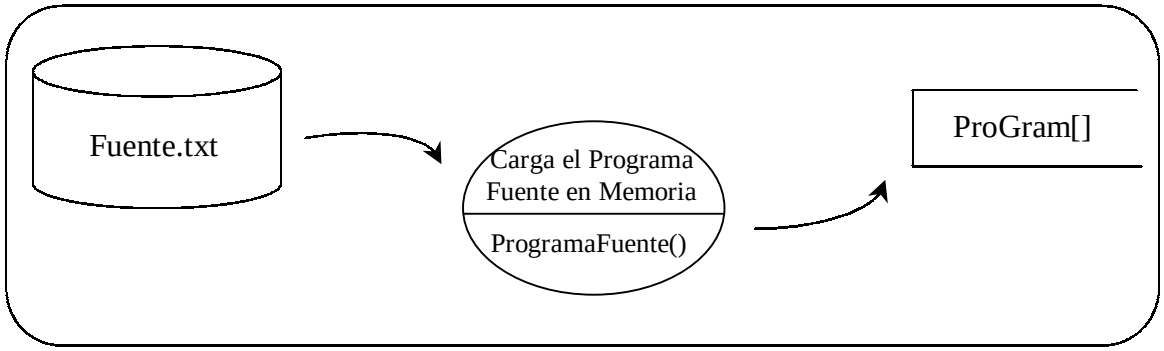

Fig. 4.5 Diagrama de Flujo de Datos de la función ProgramaFuente().

Después de la ejecución de la función ProgramaFuente(), el arreglo ProGram[] contendrá una larga cadena de caracteres, un proceso importante que se lleva a cabo mediante esta función es borrar los separadores. En la fig. 4.6 se muestra lo que sucede cuando se ejecuta la función ProgramaFuente(): el cuadro del lado izquierdo representa la memoria secundaria que puede ser cualquier medio de almacenamiento como disco duro, diskettes, etc. en el ejemplo se trata del archivo "*fuente.txt*" que contiene un programa escrito en lenguaje C para imprimir en la pantalla "hola". Después de la ejecución de la función, el contenido del archivo se almacena en la memoria primaria ya sin separadores en el arreglo denominado ProGram[], que se muestra en el cuadro de la derecha.

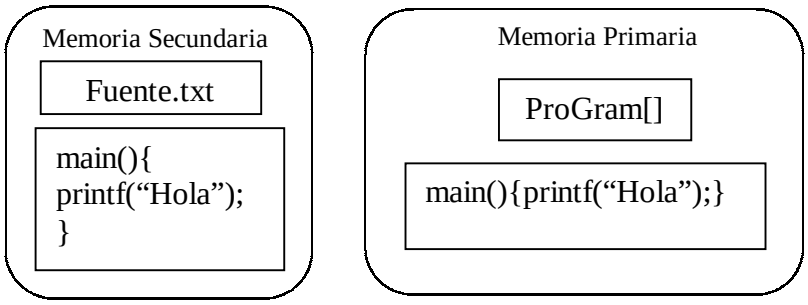

Fig. 4.6 Representación del programa fuente en la Memoria

Como se puede observar, quitar los separadores implica borrar algunos caracteres, por ejemplo los blancos o los saltos de línea y depende de la definición del lenguaje ya que algunos lenguajes como "C" se borran absolutamente todos los separadores, pero no es el caso de otros lenguajes. Por esta razón hay que tener cuidado cuando se define la gramática de un lenguaje, al respecto de la importancia de los separadores para cada lenguaje en particular. A continuación en la fig. 4.7 se muestra el pseudocódigo de la función ProgramaFuente().

```
ProgramaFuente()
{
  Abre el archivo que se desea compilar;
  Mientras ( No fin del Programa Fuente )
  {
    Lee Caracter
    Si( Carácter no es blanco o retorno de línea )
      ProGram[xxpr++]=Caracter;
  }
}
```

Fig 4.7 Pseudocódigo de la función ProgramaFuente(), que almacena el código fuente en el arreglo ProGram[].

2.2.2 Función CargaGramDim2()

La función CargaGramDim2() tiene por objetivo almacenar las reglas de producción de la gramática en la memoria primaria. Para ello se abre el archivo *gram.txt* y se vacía su contenido en dos arreglos, uno de ellos denominado Vn[] en donde se almacena el lado izquierdo de las reglas de producción y el otro denominado Gram[] en donde se guarda el lado derecho de las reglas de producción.

En las gramáticas libres de contexto el lado izquierdo de las reglas de producción equivale al conjunto de los símbolos no-terminales de la gramática. De esta manera, una vez que se ejecuta la función CargaGramDim2() el arreglo Vn[] contiene una lista de todos los símbolos no-terminales, a los que se puede hacer referencia de acuerdo a la posición en la que estén almacenados dentro del arreglo.

El lado derecho de las reglas de producción representa el conjunto de símbolos que está formado tanto por símbolos terminales así como no-terminales, además, la forma en la que están dispuestos estos símbolos indica la sintaxis del lenguaje descrito por la gramática. La función CargaGramDim2() se encarga de almacenar este conjunto en el arreglo Gram[], de manera tal que en cada registro del arreglo quede almacenado el lado derecho de sólo una regla de producción. Debido a que el lado izquierdo y derecho de una regla de producción están directamente relacionados, se debe asegurar que la posición del lado derecho dentro del arreglo Gram[] corresponda con la posición dentro del arreglo Vn[] de su símbolo no-terminal asociado.

En la fig. 4.8 se muestra el diagrama de flujo de datos de la función CargaGramDim2(). En este diagrama se puede apreciar como se divide el contenido del archivo *gram.txt* en dos arreglos: Gram[] y Vn[].

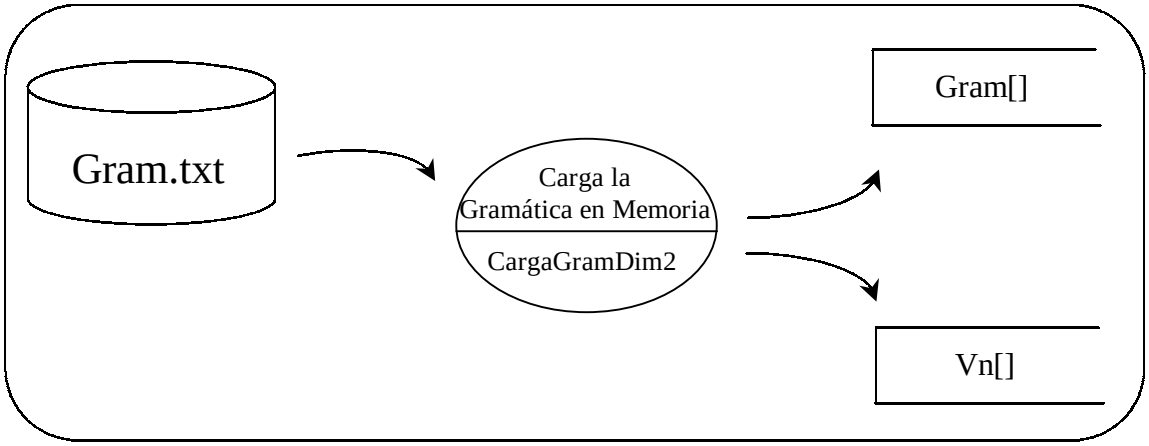

Fig. 4.8 Diagrama de Flujo de Datos de la función CargaGramDim2().

En la fig. 4.9 se representa lo que sucede a nivel de memoria cuando se ejecuta la función CargaGramDim2(). En el cuadro del lado izquierdo se muestra el contenido del archivo *gram.txt*, que básicamente es la gramática que se presentó como ejemplo en la fig. 4.3. En el cuadro derecho se muestran los dos arreglos Vn[] y Gram[] que contienen después de haber sido ejecutada la función CargaGramDim2() el lado izquierdo y derecho, respectivamente, de las reglas de producción de la gramática.

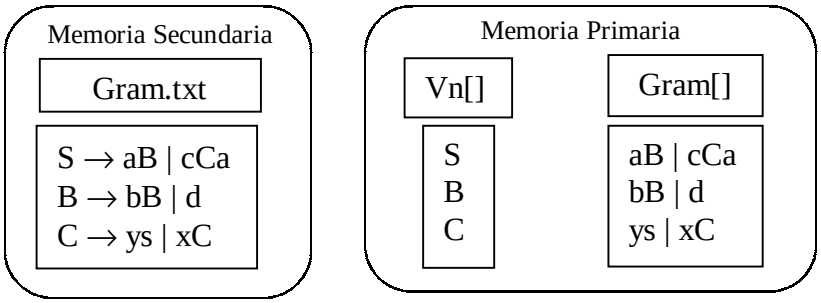

Fig. 4.9 Representación de la gramática en la Memoria

En la fig. 4.10 se muestra el pseudocódigo que lleva a cabo el proceso de la función CargaGramDim2(). Éste consiste en seleccionar el tipo de símbolo al que se hace referencia cuando se leen desde el archivo *gram.txt,* los símbolos que se encuentran al lado derecho del símbolo "→" de las reglas de producción de la gramática se almacenan en el arreglo Gram[] y los que se encuentran en el lado izquierdo se almacenan en el arreglo Vn[].

```
CargaGramDim2 ()
{
  Abre archivo Gram.txt
  Mientras ( No Fin de Gram.txt )
  {
    Lee Caracter
    Si ( Carácter == pertenece al lado derecho )
      Gram[yy][xx++] = Caracter
    else
      Si ( Carácter == pertenece al lado izquierdo )
        Vn[yyvn][xxvn++] = Caracter
  }
}
```

Fig. 4.10 Pseudocódigo de la función CargaGramDim2(), que almacena la gramática en dos arreglos: Gram[] y Vn[].

2.2.3 Función IndicaVn()

Por medio de esta función se indican en la gramática la ubicación de los símbolos no-terminales. La función CargaGramDim2() previamente almacena en el arreglo Vn[] la lista de todos los símbolos no-terminales. Cada símbolo no-terminal tiene asociada una y sólo una regla de producción (razón por la que es un símbolo no-terminal), de manera que cuando en el cuerpo (lado derecho) de las reglas de producción se hace referencia a un símbolo no-terminal, significa que hay que ejecutar la regla de producción que tiene asociada dicho símbolo.

Para que esto se pueda llevar a cabo adecuadamente, el sistema lee las reglas de producción por medio de la función principal LeeGram() y debe poder distinguir si se trata de un símbolo terminal o no-terminal. Los compiladores resuelven este problema de diferentes maneras algunos obligan a que el usuario asigne letras mayúsculas a los símbolos terminales y minúsculas a los no-terminales. Sin embargo, en el desarrollo de este sistema se pretende imponer la menor cantidad de restricciones posible y respetar la libertad del usuario de usar los símbolos que guste con letras mayúsculas o minúsculas, como consecuencia tiene que ser el sistema el que determine de qué tipo de símbolo se trata.

La función IndicaVn() es la encargada de llevar a cabo este proceso de identificación y se hace de la siguiente manera: Dado que el arreglo Vn[] contiene una lista secuencial de los símbolos no-terminales, sólo basta identificar dichos símbolos en el arreglo Gram[]. Así que, con base en la lista contenida en el arreglo Vn[], se inicia una búsqueda de cada símbolo no-terminal a través de toda la gramática y cada vez que es identificado un símbolo no-terminal, se marca con el índice que le corresponde en el arreglo Vn[].

En la fig. 4.11 se presenta un ejemplo de como se lleva a cabo este proceso a nivel de memoria, en el cuadro de la izquierda se muestra el contenido de los arreglos Vn[] y Gram[], al lado del contenido del arreglo Vn[] se ha agregado una numeración que indica la posición de los símbolos no-terminales dentro del arreglo Vn[]. En el cuadro de la derecha se puede ver el contenido del arreglo Gram[] una vez que se ha ejecutado la función IndicaVn(). Es posible observar que las marcas que se introducen corresponden a la posición de los símbolos no-terminales en el arreglo Vn[]. Desde luego el proceso que lleva a cabo esta función es transparente para el usuario, lo que significa que no se entera de todo lo que se tiene que realizar para que funcione adecuadamente el sistema. En este sentido, el usuario sólo se debe preocupar por desarrollar y escribir adecuadamente la gramática del lenguaje.

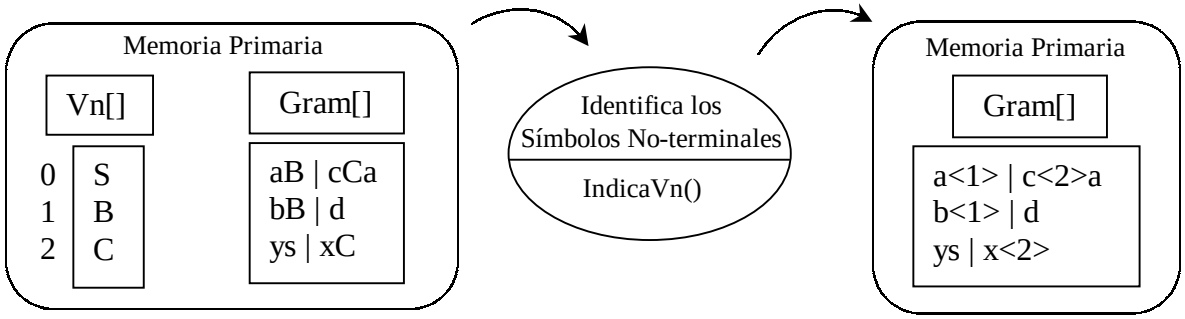

Fig. 4.11 Identificación de los símbolos no-terminales en la gramática

De esta manera, la gramática queda marcada y lista para ser utilizada por la función principal del sistema. Este preprocesamiento se lleva a cabo con el fin de facilitar al usuario la definición de la gramática, de lo contrario sería él quien tendría que definir explícitamente cada uno de los símbolos que utilice en la gramática. En la fig. 4.12 se presenta el pseudocódigo de la función IndicaVn().

```
IndicaVn()
{
 Mientras( Lista de Símbolos no-terminales no fin )
 {
  Mientras( No fin Gram[] )
  {
    Si ( Símbolo en Gram[] == Símbolo en Vn[] )
      Pon Marca en Gram[]
  }
 }
}
```

Fig.4.12 Pseudocódigo de la función IndicaVn(), que identifica los símbolos no terminarles en el arreglo Gram[].

2.2.4 Función LeeGram()

LeeGram() es la función que representa el corazón de este sistema reconocedor, implementa tanto el analizador léxico, así como el sintáctico. Es la encargada de verificar que el programa fuente cumpla con las reglas establecidas por la gramática, de manera que sea aceptado o rechazado como perteneciente al lenguaje descrito por la gramática. Esta función ha sido implementada mediante un algoritmo descendente recursivo, de ahí que este sistema sea considerado dentro de la clasificación de los analizadores (*parsers*) descendentes recursivos.

En la fig. 4.13 se muestra el diagrama de flujo de datos de esta función. Como se puede apreciar, las entradas que se utilizan para llevar a cabo el proceso de la función LeeGram() consisten en los datos contenidos en los arreglos Gram[], Vn[] y ProGram[], que son la consecuencia del preprocesamiento que efectúan las funciones CargaGramDim2(), IndicaVn() y ProgramaFuente(). Al ejecutar la función LeeGram() se activa el mecanismo que permite evaluar si el programa fuente que se encuentra almacenado en el arreglo ProGram[], cumple con las reglas establecidas por la gramática, que se encuentran almacenadas en el arreglo Gram[] o una bandera de error si el código del programa fuente está escrito con símbolos que no pertenecen al alfabeto del lenguaje o no cumple con la estructura del mismo.

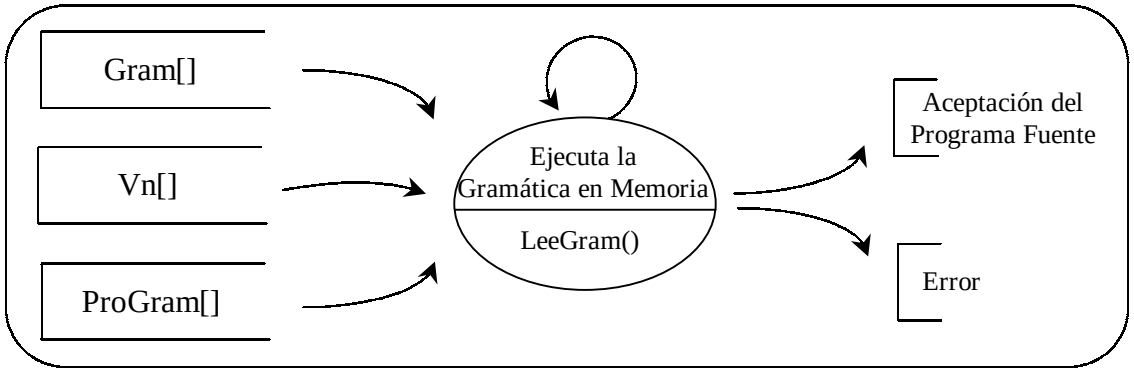

Fig. 4.13 Diagrama de Flujo de Datos de la Función LeeGram().

Para que el sistema reconozca que el código del programa fuente corresponde a las características definidas en la gramática, se debe cumplir que: Los símbolos utilizados para formar las oraciones o instrucciones del programa fuente deben pertenecer al alfabeto del lenguaje y por otro lado, una característica fundamental es que las oraciones cumplan con la estructura o sintaxis del lenguaje, lo que quiere decir que las palabras o símbolos con los que se forman las oraciones deben estar dispuestos en un orden específico.

Por ejemplo, la gramática de la fig. 4.14 representa a través de sus reglas de producción el lenguaje constituido por las oraciones {ac, ae, bdc, bde} y ninguna otra cadena pertenece a este lenguaje, a excepción de la cadena vacía. De esta manera, a partir de la regla de inicio "S" es posible generar o reconocer dicho lenguaje, siguiendo los diferentes caminos que ofrecen las reglas de producción de la gramática. Así que si la oración comienza con el símbolo "a" sólo pueden seguir el símbolo "c" o el "e", en ese orden, con esa estructura, de no ser así implicaría que existe un error en el programa fuente. Y si empieza con la "b" debe seguir la "d" y después la "c" o la "e", cumpliéndose de esta forma con la estructura del lenguaje.

1) $S \rightarrow aA \mid bB$

2) $A \rightarrow c \mid e$

3) $B \rightarrow dA$

Fig 4.14. Gramática Generativa Libre de Contexto

En este punto es importante notar que las reglas de producción de la gramática indican la estructura en la que se deben formar las oraciones, es decir, el orden en el que deben ser escritos los símbolos terminales. Los símbolos no-terminales no existen en el lenguaje, se introducen en la gramática con el fin de representar que después de un símbolo terminal sigue una de diferentes opciones de símbolos terminales, en el ejemplo de la fig. 4.14 después de una “a” puede seguir una “c” o una “e”, que son las opciones que ofrece la segunda regla de producción etiquetada con el símbolo no-terminal “A”. Finalmente, un lenguaje puede ser visto como una cadena o secuencia de símbolos terminales, esta conceptualización de la realidad permite ver a los lenguajes como enormes cadenas de bits y de esta forma, manipularlos mediante algoritmos computacionales.

Por definición de gramática, el alfabeto del lenguaje lo constituye el conjunto de símbolos terminales (Vt) y forman parte del lado derecho de las reglas de producción. Para verificar que las oraciones del programa fuente están formadas con símbolos que pertenecen al alfabeto, basta con comprobar que están formadas únicamente por los símbolos terminales definidos por la gramática. En este sentido, la función LeeGram() dispone de un mecanismo que le permite identificar cuando se trata de un símbolo terminal, de acuerdo a las reglas de producción de la gramática y con base en las marcas que introduce la función IndicaVn()

La función LeeGram() lee los símbolos almacenados en el arreglo Gram[], que corresponden a las reglas de producción de la gramática y de manera simultanea va leyendo los símbolos almacenados en el arreglo ProGram[], que pertenecen a los símbolos del programa fuente, de manera que la estructura del lenguaje se va comprobando de acuerdo al orden en el que están dispuestos los diferentes conjuntos de símbolos en las reglas de la gramática. En caso de que el símbolo leído en la gramática sea un símbolo terminal, implica que el símbolo leído del programa fuente debe coincidir con ese símbolo, así pues, el sistema compara los símbolos leídos de la gramática y del programa fuente y estos símbolos deben ser iguales, de otra forma existe algún error. Por ejemplo, supóngase la gramática de la fig. 4.15.

$$S \rightarrow main()\{\}$$

Fig. 4.15 Gramática que genera sólo símbolos terminales

Está formada por una sola regla de producción que genera los símbolos terminales: m, a, i, n, (,), {, }. Y sea el programa fuente de la fig. 4.16.

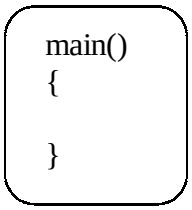

Fig. 4.16 Programa Fuente que cumple con las reglas de la gramática de la fig. 4.15

Para comprobar que este programa cumple con las reglas de la gramática, el sistema hace lo siguiente: Primeramente, la función CargaGramDim2() carga la gramática en dos arreglos Vn[] y Gram[], fig. 4.17.

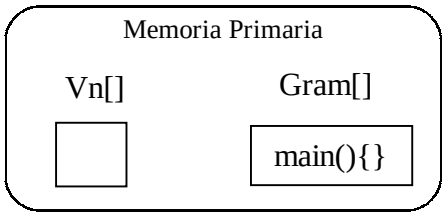

Fig. 4.17 Representación en la Memoria de los arreglos Vn[] y Gram[]

Posteriormente, se ejecuta la función IndicaVn() que es la encargada de marcar en las reglas de la gramática, es decir en el arreglo Gram[], los símbolos no-terminales. En este ejemplo, la regla de la gramática contiene únicamente símbolos terminales, así que este arreglo queda igual. Después se carga el programa fuente en el arreglo ProGram[], fig. 4.18.

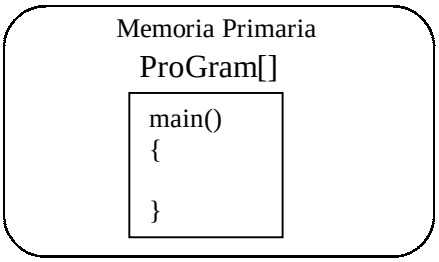

Fig. 4.18 Representación en la Memoria del arreglo ProGram[]

Finalmente, se ejecuta la función LeeGram(). Esta función ignora todos los separadores, que incluyen los espacios en blanco, los retornos de carro y los saltos de línea que sean parte del programa fuente. En las reglas de producción sólo puede haber 4 tipos de símbolos: terminales, no-terminales, “O” (|) y el símbolo de la cadena vacía “Ø”, ya que en esta regla no aparece ni el “O”, ni el “Ø” y dado que no existen marcas en la gramática que indiquen la presencia de símbolos no-terminales, entonces la función LeeGram() debe verificar que se trata únicamente de símbolos terminales. Para ello comienza comparando uno a uno los símbolos del programa fuente y de la gramática y como se puede ver en la fig. 4.19, los símbolos del programa fuente coinciden con los descritos por la gramática y por lo tanto, el programa si pertenece al lenguaje.

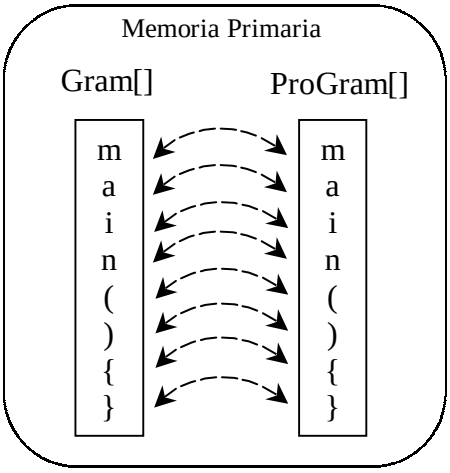

Fig. 4.19 Comparación que lleva a cabo LeeGram()
entre los arreglos Gram[] y ProGram[]

Esta es la manera en la que se procesan los símbolos terminales que representa el análisis léxico del lenguaje. Una característica de la función LeeGram() que le permite procesar las reglas de producción de la gramática y con ello reconocer los lenguajes, está representada mediante el arco en la fig. 4.13 que le da la posibilidad a esta función de llamarse a sí misma de manera recursiva. En caso de que el símbolo leído desde el arreglo Gram[] esté marcado como símbolo no-terminal, entonces se ejecutará la función LeeGram() de manera recursiva, procesando la regla de producción asociada a dicho símbolo no-terminal.

Cuando una función se ejecuta de forma recursiva hay que tomar en cuenta varios aspectos, ya que las bondades de las funciones recursivas pueden fácilmente convertirse en ciclos infinitos que podrían llevar al sistema al estado de "*down*". Existen ciertos datos como el número de regla que se está procesando y la posición o caracter que está siendo examinado, que con cada llamado recursivo tienen que ser almacenados. Para ello, la función LeeGram() hace uso de una pila denominada PilaRec[], que de forma teórica debería ser infinita, pero en la práctica está limitada a los recursos del sistema computacional. Si el llamado recursivo se lleva a cabo de manera errónea, la capacidad de almacenamiento de esta pila puede ser excedida, traduciéndose en problemas que podrían incluso dañar, incluso, procesos esenciales del equipo de cómputo.

En la fig. 4.20 se muestra un ejemplo de una gramática que contiene en dos reglas de producción y por lo tanto, dos símbolos no-terminales, "S" y "Digito". Esta gramática es el resultado de agregar a la gramática de la fig. 4.15 la posibilidad de producir o reconocer un dígito.

S → main(){Digito}
Digito → 0|1|2|3|4|5|6|7|8|9

Fig. 4.20 Gramática que genera un dígito.

En la fig. 4.21 se presenta un programa fuente que cumple con las reglas de esta gramática. El número "5" se eligió al azar, pero de acuerdo a las reglas de producción de la gramática puede ser cualquiera de los diez dígitos.

```
main()
{
   5
}
```

Fig. 4.21 Programa Fuente que cumple con
las reglas de la gramática de la fig. 4.20

El estado de la memoria principal se muestra en la fig. 4.22. Este es el resultado de llevar a cabo el preprocesamiento por parte de las funciones CargaGramDim2, IndicaVn() y ProgramaFuente(). El arreglo GramCpy[] es una copia del arreglo Gram[] y sólo se utiliza para hacer un paso intermedio para incrustar las marcas de los símbolos no-terminales.

Como se puede observar en la fig. 4.22, en el arreglo Gram[] en la primera línea aparecen los símbolos “Å1Æ” en sustitución del símbolo “Dígito”. Estos indican que se trata de un símbolo no-terminal y además, el número que aparece entre ellos tiene que ver con el número de la regla de producción asociada al símbolo no-terminal “Digito”, que en este ejemplo le corresponde a la segunda regla de producción.

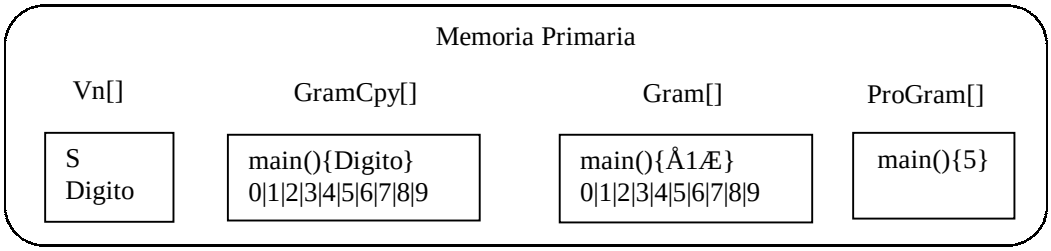

Fig. 4.22 Representación en la Memoria de los arreglos Vn[], GramCpy[], Gram[] y ProGram[].

Una vez que la memoria se encuentra en el estado mostrado en la fig. 4.22 se ejecuta la función LeeGram(), que empieza a comparar los símbolos de la primera regla de producción con los símbolos del programa fuente de igual manera que como se muestra en la fig. 4.19, es decir, lee el primer símbolo de la primera regla “m” y lo compara con el primer símbolo del programa fuente “m”, ya que son iguales, lee los siguientes símbolos, el “a” de la gramática y el “a” del programa fuente. Hasta que lee el símbolo “Å” y el símbolo “5” en el programa fuente que son símbolos diferentes. Los símbolos “Å” y “Æ” de la gramática indican que se trata de un símbolo no-terminal y por lo tanto, se llama a sí misma la función LeeGram() y comienza a procesar la segunda regla de producción.

Cuando compara el primer símbolo de la segunda regla de producción “0” con el símbolo “5” del programa fuente, encuentra que no coinciden por lo que hace una búsqueda en esta regla de producción por el símbolo “|”, que significa que hay otra opción. De esta forma va comparando el “5” del programa fuente con todos los símbolos de la segunda regla de producción, hasta que finalmente llega a uno donde si coincide, es decir, cuando lo compara con la sexta opción de esta regla de producción. En caso de que no existiera ningún símbolo que corresponda con el leído desde el programa fuente, entonces se marcaría un error.

La gramática de la fig. 4.20 obliga a que haya en el programa fuente un dígito entre las llaves, esto quiere decir que si se dejara vacío el espacio entre las llaves habría un error. Para ilustrar como se utiliza la cadena vacía, en la fig. 4.23 se ha agregado a la gramática de la fig. 4.20 la posibilidad de que entre las llaves haya cero o un dígito. Para ello, se agrega al final de la segunda regla de producción el símbolo “Ø”, que identifica la cadena vacía.

S → main(){Digito}
Digito → 0|1|2|3|4|5|6|7|8|9|Ø

Fig. 4.23 Gramática que genera un dígito.

Al permitir que la cadena vacía sea parte del lenguaje, se abre la posibilidad de que el programa fuente de la fig. 4.24 sea reconocido como válido. El proceso de reconocimiento se lleva exactamente igual, sin embargo, al terminar de comparar con los diez dígitos de la segunda regla de la gramática y encontrar que el espacio en blanco del programa fuente, efectivamente, no coincide con ninguno, se lee la siguiente opción de la regla que es el símbolo “Ø” que significa que aunque no venga nada es correcto. Es decir, entre las llaves puede haber uno o ningún dígito.

```
main()
{

}
```

Fig. 4.24 Programa Fuente que cumple con
las reglas de la gramática de la fig. 4.23

Este proceso se ejecuta hasta que se lea el último símbolo del programa fuente o hasta que aparezca un error. En la fig. 4.25 se presenta el pseudocódigo de la función LeeGram().

```
LeeGram()
{
 Mientras( ProGram[x] != Fin  &  Error !=1  &  Gram[x] != Fin  &  Gram[] != "|"  &  vacia == 0 )
 {
   Ignora Separadores;
   Si(Gram[x] == ProGram[x]                    // Si( Símbolo == Terminal )
     x++;                                      // Lee Siguientes Símbolos
   else

     Si( Gram[x] == "Å" )                      // Si( Símbolo == No-terminal )
       LeeGram(siguiente regla)                // Analiza Siguiente Regla
     else

       Si( Gram[x] == "Ø"                      // Si es el símbolo de la cadena vacía
         vacia = 1;                            // Sale del ciclo sin error
       else

         Error

   Si( Error )
     Busca "|"                                 // Busca la siguiente opción
 }
 vacia = 0;
}
```

Fig. 4.25 Pseudocódigo de la función LeeGram(), que verifica la estructura del lenguaje.

CAPÍTULO III

ESPECIFICACIONES Y VALIDACIÓN DEL SISTEMA

3.1 ESPECIFICACIONES DEL SISTEMA

La interfaz de este sistema reconocedor de lenguajes consiste en tres ventanas de edición de texto, cada una de ellas independiente de la otra. Una de ellas es utilizada para editar la gramática libre de contexto del lenguaje que se quiere modelar, otra para editar el programa fuente y en la tercera donde se informa si el programa fuente pertenece al lenguaje descrito por la gramática (Fig. 5.1).

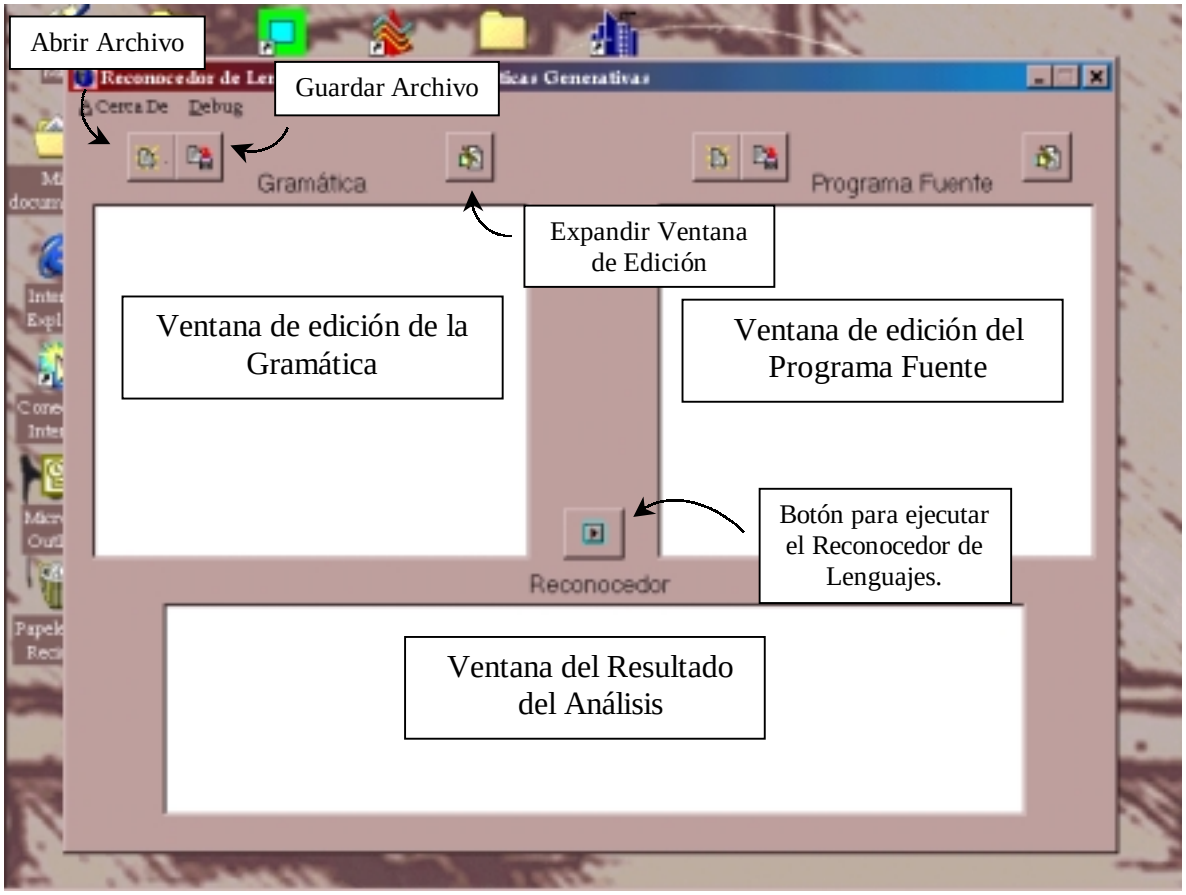

Fig. 5.1 Interfaz del Sistema

Las reglas que se escriben en esta ventana de texto obedecen a la siguiente notación: Empiezan con un símbolo no-terminal seguido del símbolo que identifica el operador de producción (→). Debido a que la ventana de la edición de la gramática es un editor de texto, este operador se forma con uno o más guiones y "un mayor que" (>). Finalmente, el lado derecho de las reglas de producción que se conforma por símbolos terminales, no-terminales, el símbolo "|" que significa "o" y que permite identificar las diferentes opciones que ofrece la regla de producción o el símbolo "Ø" que identifica la cadena vacía.

Los símbolos no-terminales, que se encuentran del lado izquierdo de las reglas de producción, pueden ser cualquier cadena de caracteres. Deben ser seguidos de cero o más espacios en blanco, uno o cualquier cantidad de guiones y un símbolo de "mayor que" ">". Este último símbolo identifica la punta de flecha del operador de producción (→) de las reglas y es obligatorio, este símbolo indica el límite entre el lado izquierdo y el lado derecho de las reglas de producción. En la fig. 5.2 se muestran algunas reglas de producción.

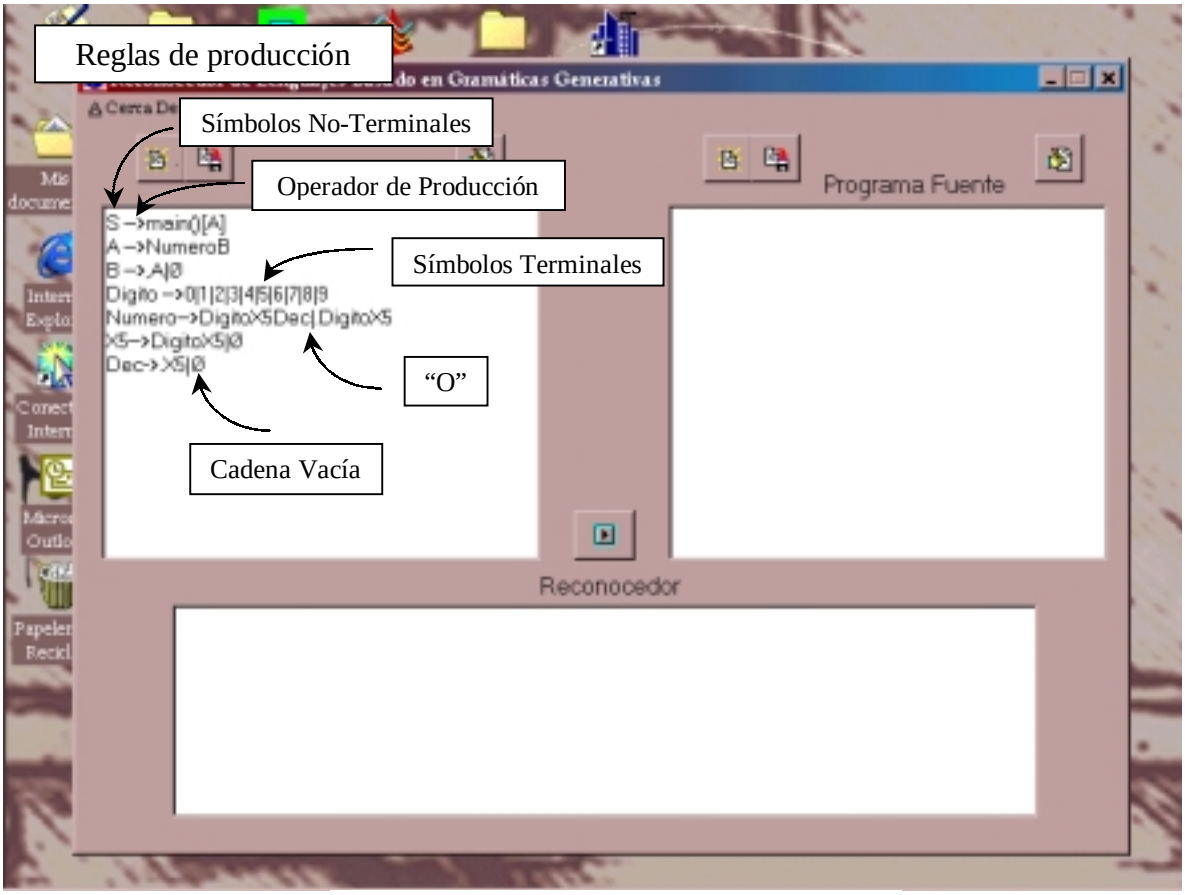

Fig. 5.2 Reglas de Producción y Símbolos de la Gramática

El único símbolo no-terminal que se ha reservado es el carácter “S”, con el fin de que la regla de inicio pueda ser escrita en cualquier parte del archivo. De esta manera, todas las gramáticas deberán contener una regla de producción etiquetada con el símbolo no-terminal “S”. En cualquier línea en la que aparezca la regla S→... se tomará como el inicio de la gramática y ninguna otra regla de la gramática deberá ser definida con el símbolo no-terminal “S”, que es una regla general, pues dos reglas de producción no pueden tener lados izquierdos iguales.

Con respecto a la ventana de edición del programa fuente, fue diseñada con el objetivo de proporcionar una herramienta con la cual pueda ser abierto, guardado o editado un archivo de texto que contenga el programa fuente. Tanto la ventana de edición de la gramática, así como la del programa fuente, han sido colocadas de manera tal que se pueda verificar fácilmente el contenido de ambas, con el fin de poder comprobar que el programa fuente contiene los símbolos de la gramática. Sin embargo, también es posible expandir estas ventanas, de manera que sea más amplio el campo de visión que se tiene, además de contar con las herramientas como copiar, pegar, cortar, etc. de un editor de textos.

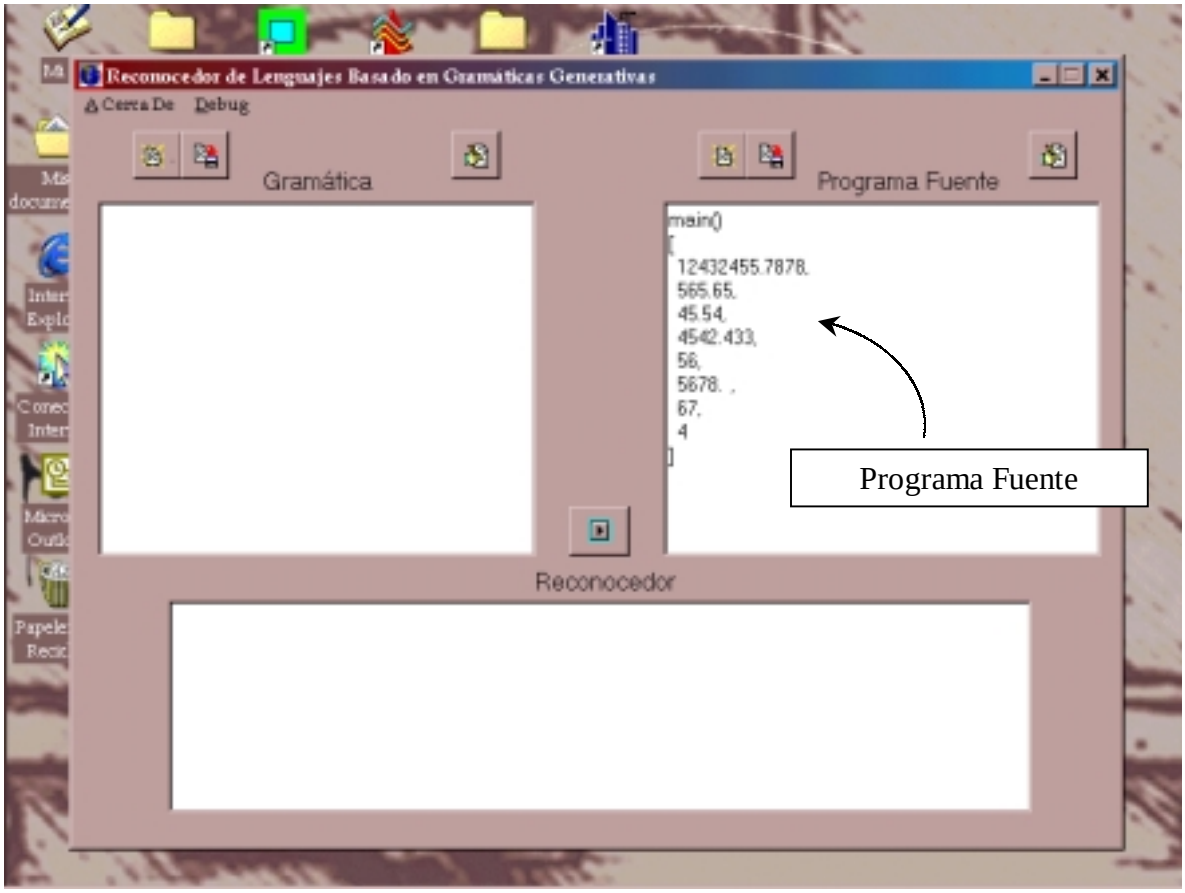

Fig. 5.3 Ventana de edición del Programa Fuente

La tercera ventana que aparece en la pantalla principal del sistema, es utilizada para informar el resultado de analizar el programa fuente de acuerdo a las reglas de producción de la gramática. A través de esta ventana el sistema informará si el programa fuente pertenece o no al lenguaje descrito por la gramática. En caso de que efectivamente las oraciones del programa fuente cumplan con las reglas de producción, aparecerá en esta ventana la frase afirmativa: “Felicidades, este programa Si pertenece al lenguaje”. En caso contrario, será la frase negativa: “Programa que No pertenece al lenguaje”.

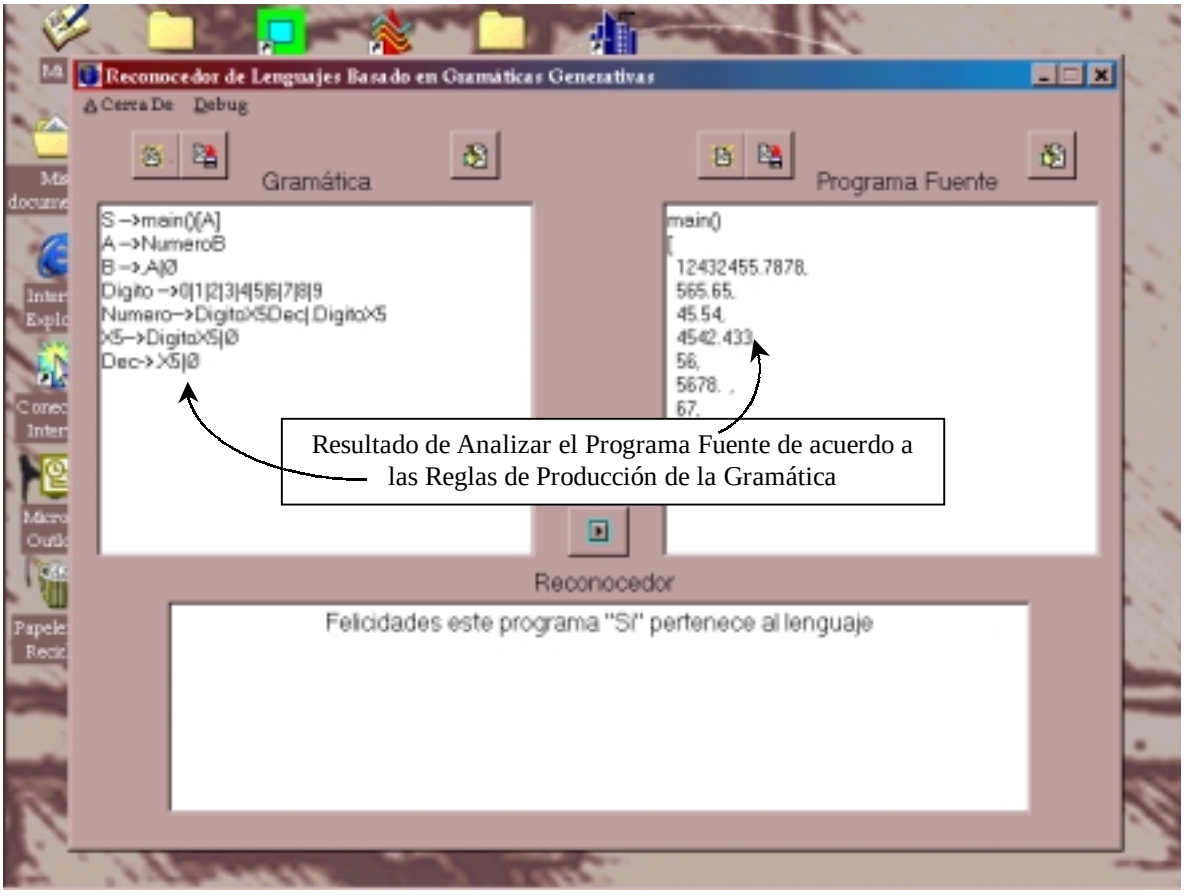

Fig. 5.4 Resultado de ejecutar el Reconocedor de Lenguajes

3.2 VALIDACIÓN DEL SISTEMA

A continuación se mostrarán algunos ejemplos de lenguajes así como su gramática que pueden ser reconocidos por este sistema. El primer ejemplo es de una gramática que genera un solo símbolo terminal, en particular este lenguaje está constituido por una sola cadena: {a}. Fig. 5.5.

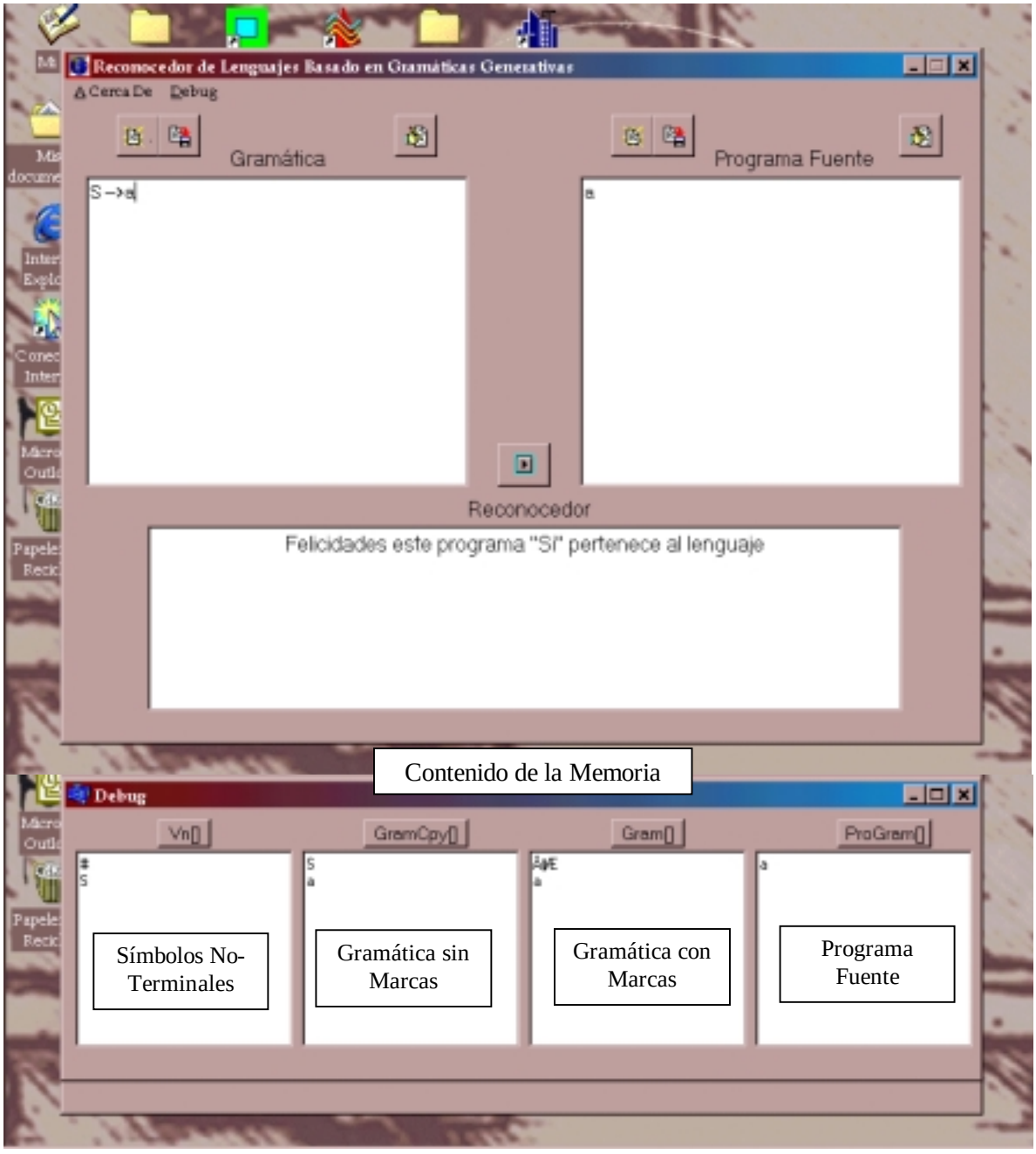

Fig. 5.5 Ejemplo 1

En el ejemplo de la fig. 5.5 se puede apreciar como el programa fuente, que contiene la oración “a”, si cumple con las reglas de la gramática, es decir es un programa que está correctamente escrito. También es posible ver el contenido de la memoria principal representada por 4 arreglos. El arreglo Vn[] contiene dos símbolos (#, S), el “#” no existe en la gramática, lo agrega el sistema con el fin de que sea posible escribir la regla de inicio (S→…) en cualquier parte del archivo. La siguiente ventana muestra el contenido del arreglo GramCpy[] que es una copia de la gramática, la cual no contiene ninguna marca. Después se muestra el contenido del arreglo Gram[], en donde la gramática ya ha sido marcada por la función IndicaVn(). De esta manera, se puede apreciar en la primera línea de la ventana de Gram[] los símbolos “Å_Æ” que representan que se trata de un símbolo no-terminal y entre estos símbolos, lo que aparece como una raya representa la posición en la que se encuentra la regla de producción, en este caso, asociada al símbolo no-terminal “S”. Esto es debido a que en este tipo de ventanas se muestra “texto plano” y Gram[] es un arreglo tipo “char”, por lo que se muestra el caracter con valor “1”, así que aunque en la pantalla se ve un caracter extraño, internamente representa el número de línea en el que se encuentra la regla que hay que procesar. Y en la última ventana se muestra el contenido del arreglo ProGram[], es decir el programa fuente.

En la fig. 5.6 se muestra otro ejemplo que contiene únicamente símbolos terminales. A partir de este ejemplo se va a construir una gramática que permita reconocer números.

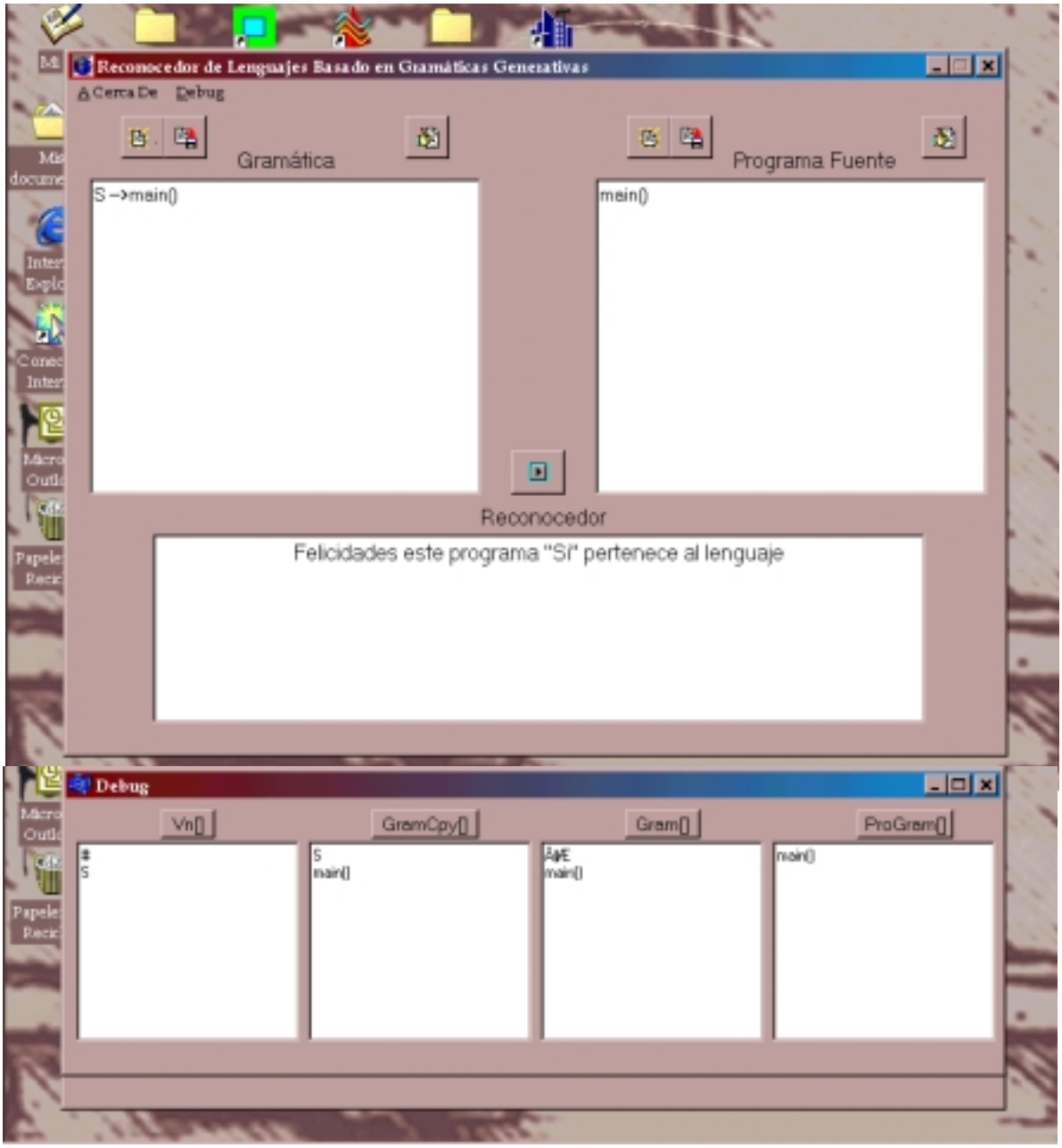

Fig. 5.6 Ejemplo 2. Contenido de la Memoria

En la fig. 5.7 se agrega la posibilidad de reconocer un dígito. Para reconocer uno de los 10 dígitos, en la fig. 5.7 se agregó en la segunda línea la regla de producción etiquetada con el símbolo no-terminal “A”, que hace referencia a la regla de producción “Dígito”, que se encuentra en la tercera línea y que produce los 10 dígitos. En la fig. 5.8 se muestra una gramática a la que se le ha agregado la regla de producción “B”, que permite reconocer varios dígitos seguidos de una “,”. En esta regla de producción se utiliza el símbolo de la cadena vacía, lo que quiere decir que puede seguir una “,” seguida de otro dígito o nada.

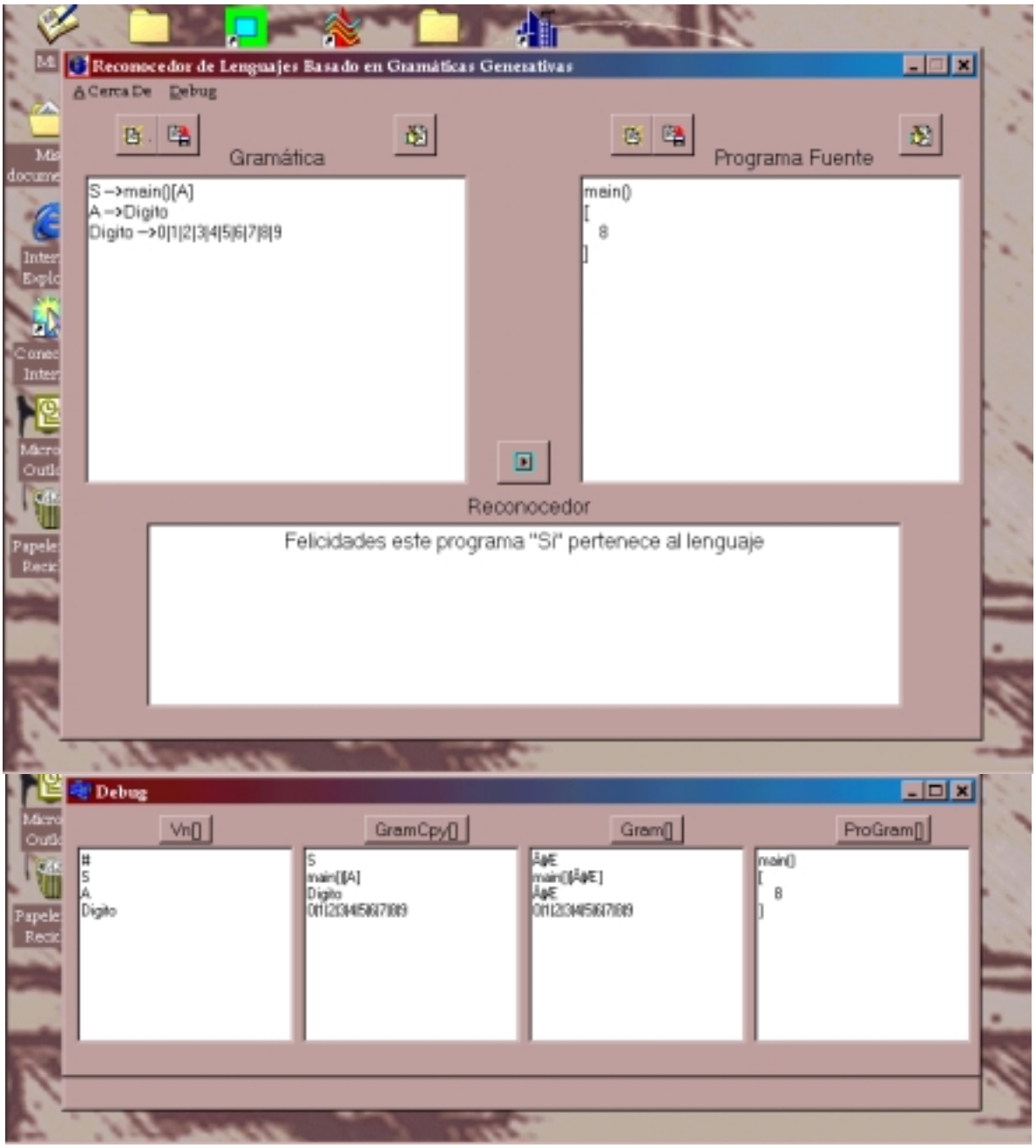

Fig. 5.7 Ejemplo 3

En la fig. 5.8 se muestra una gramática a la que se le ha agregado la regla de producción “B”, que permite reconocer varios dígitos seguidos de una “,”. En esta regla de producción se utiliza el símbolo de la cadena vacía, lo que quiere decir que puede seguir una “,” seguida de otro dígito o nada.

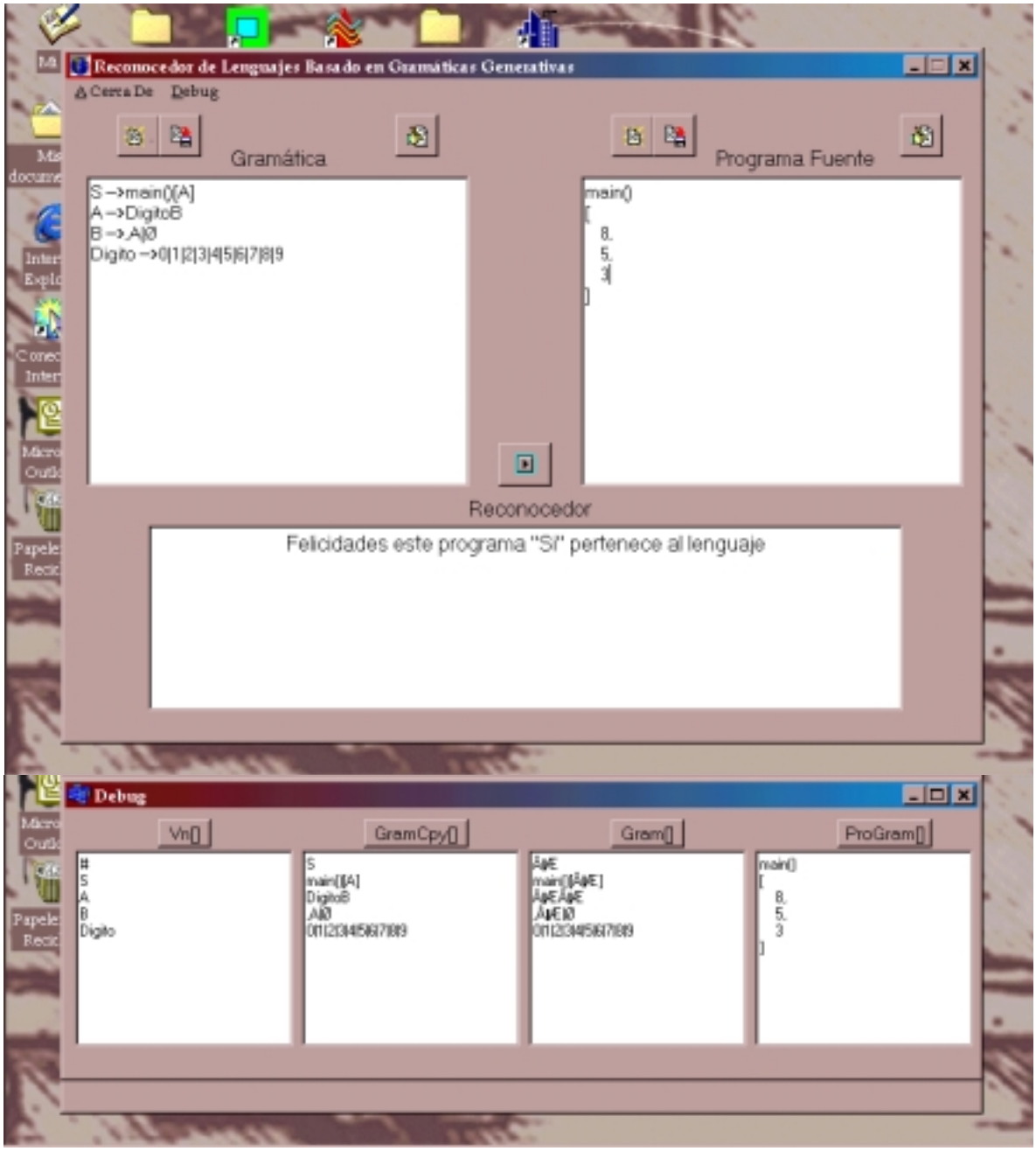

Fig. 5.8 Ejemplo 4

En el ejemplo 5 de la fig. 5.9 se muestra una gramática a la que se le ha agregado la regla de producción etiquetada con el símbolo no-terminal “Numero”, que permite formar números de más de un dígito. Para ello utiliza la regla recursiva “X5” que produce un solo dígito, pero la posibilidad de llamarse recursivamente, le permite formar números de más de un dígito.

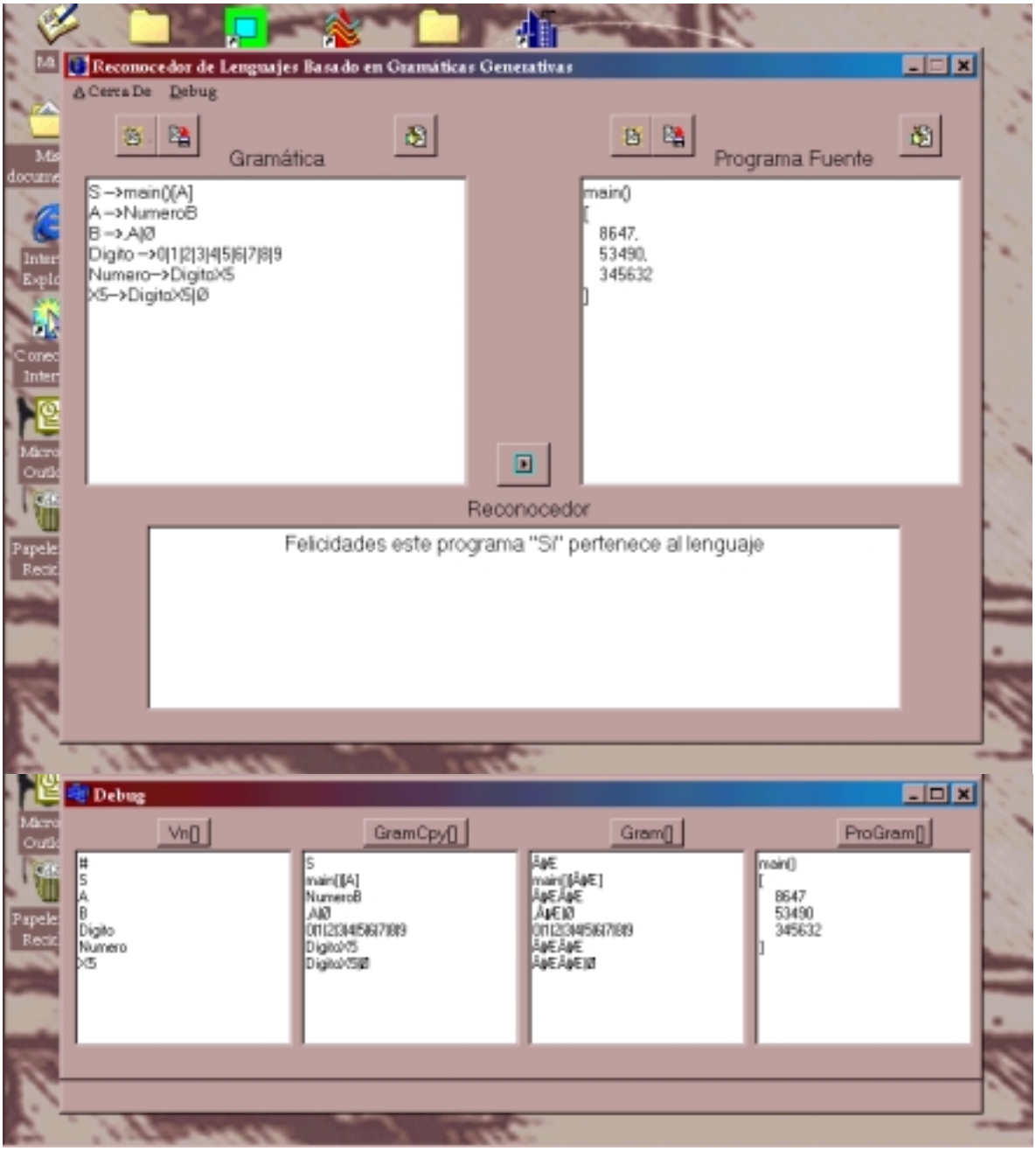

Fig. 5.9 Ejemplo 5

En la gramática del ejemplo 6 fig. 5.10, se ha agregado la regla de producción “Dec”, que permite incluir en los números el punto decimal. Con esta gramática es posible reconocer números separados por “,”, entre corchetes “[]” y que tienen la cadena “main()” como cabecera.

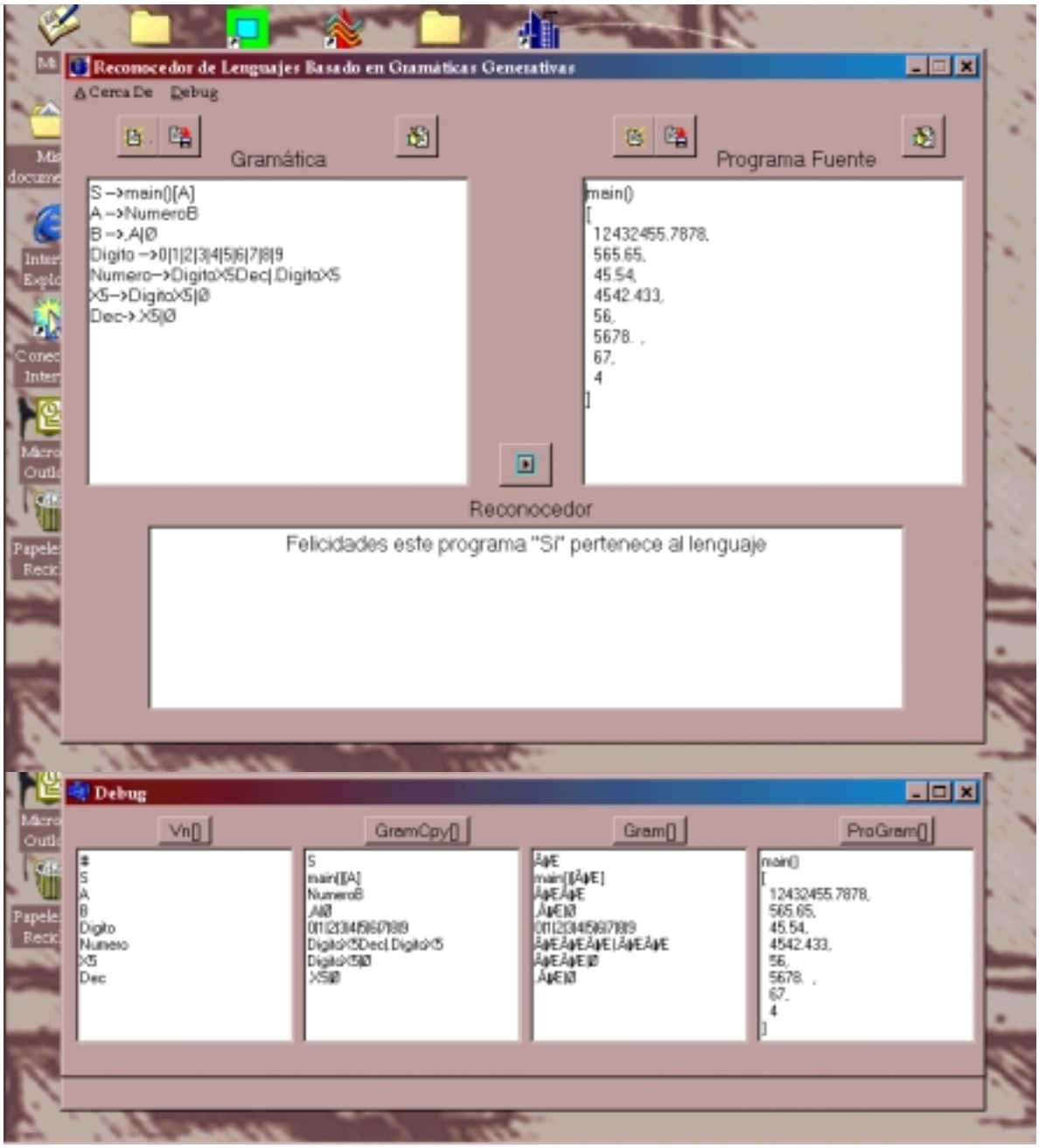

Fig. 5.10 Ejemplo 6

A partir del ejemplo 2 y hasta el 6 se ha mostrado como se puede construir una gramática que permita reconocer números, desde aquéllos formados por un solo dígito, hasta aquéllos que utilizan punto decimal. En la siguiente gramática, 5.11, se muestra el tipo de reglas de producción que se utilizan para reconocer una estructura para mandar a imprimir en pantalla una cadena de caracteres, utilizada en el lenguaje C++. Este es probablemente, el primer programa que aprende todo programador.

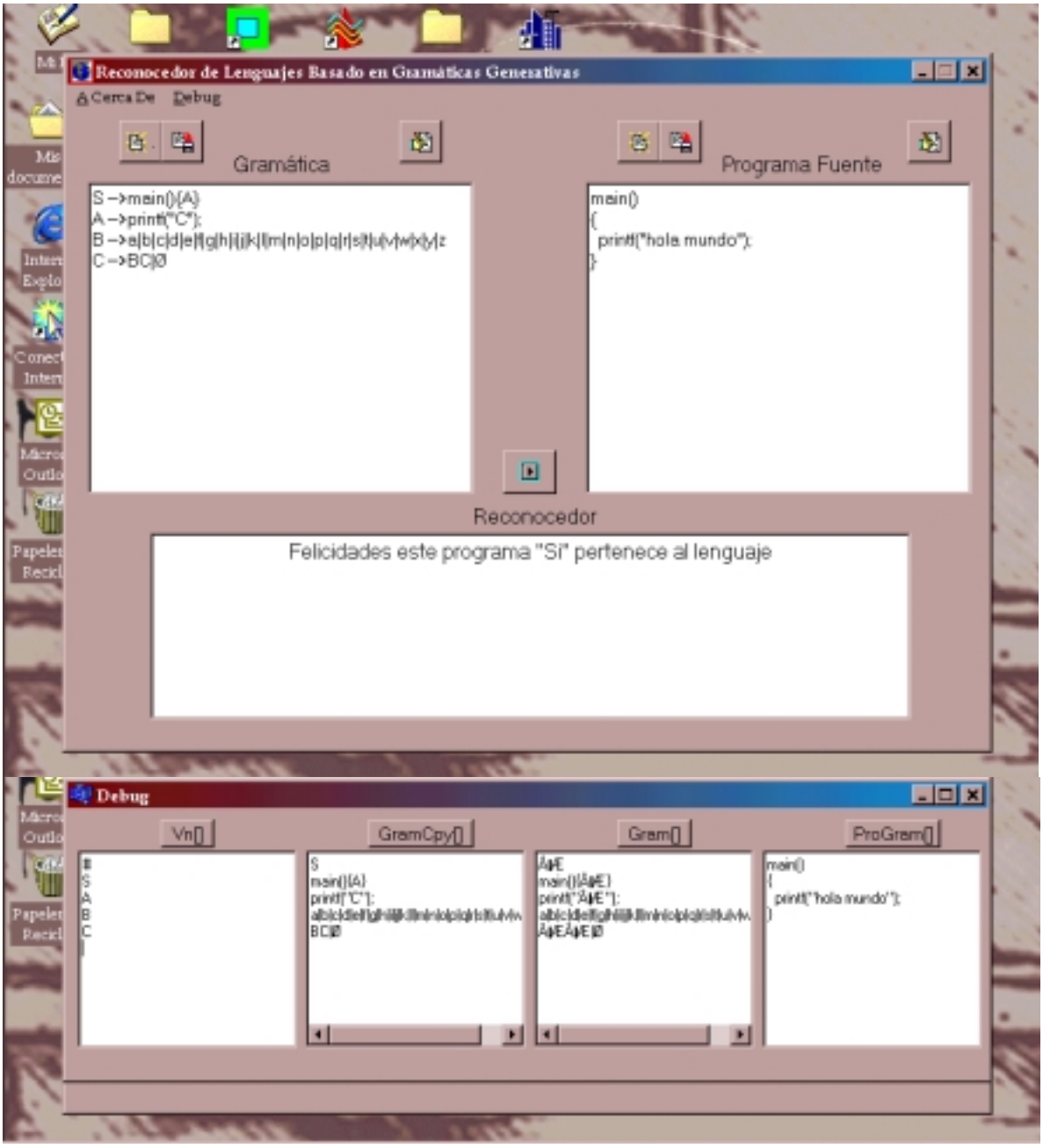

Fig. 5.11 Ejemplo 7

En la fig. 5.12 se muestra la construcción de un lenguaje que se ha denominado CStop, ya que está basado en la gramática del lenguaje Stop, pero se han agregado algunos símbolos separadores con base en la gramática del lenguaje C++. Por esta razón, no es Stop puro, pero tampoco es C.

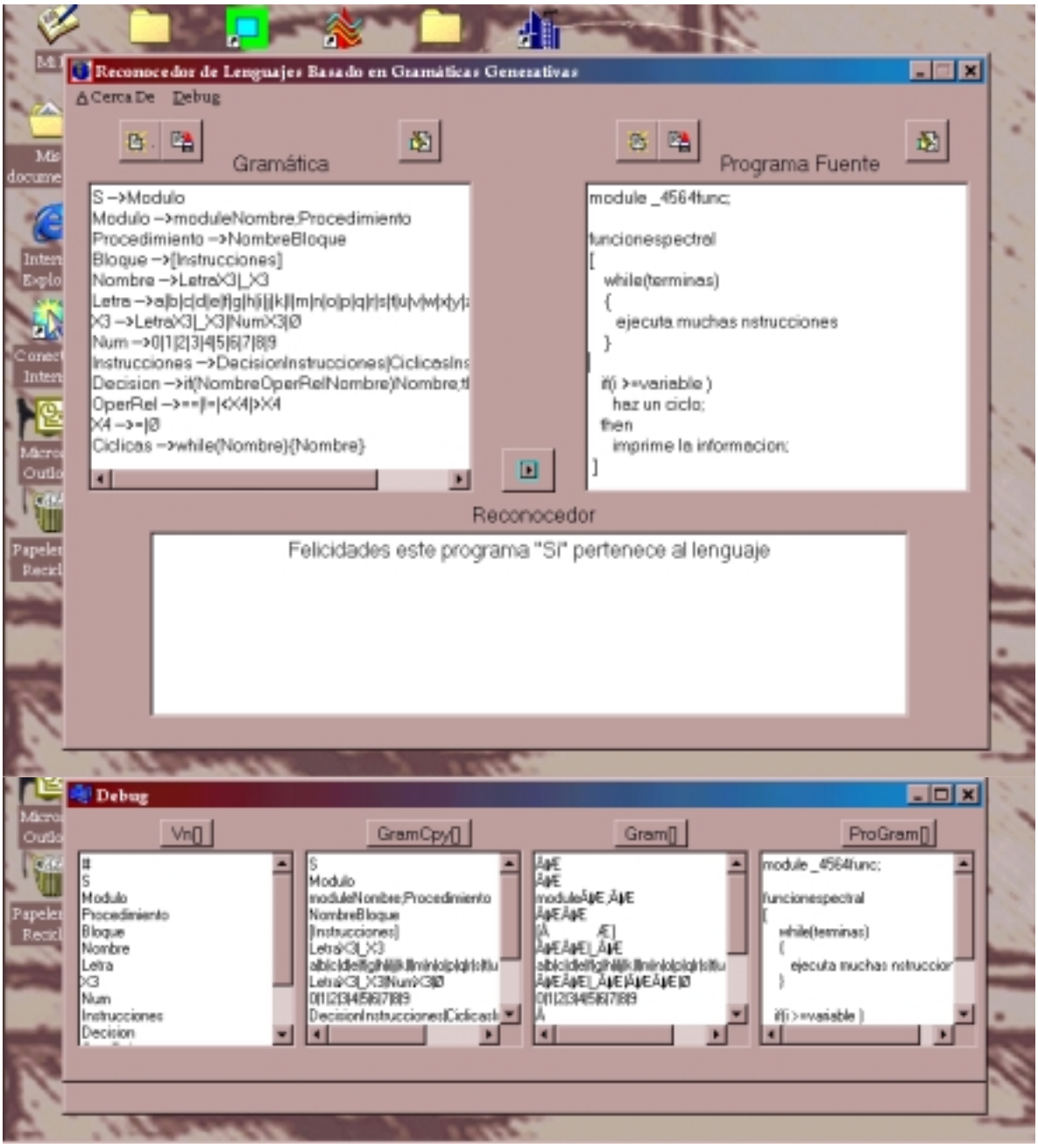

Fig. 5.12 Ejemplo 8

Las reglas de producción de la gramática del lenguaje CStop se interpretan de la siguiente manera: Lo primero que se genera es un “Modulo”, que consiste en la palabra “module”, seguida de un “Nombre”, un “Procedimiento” y un caracter terminal “;”. El “Nombre” puede empezar con una “Letra” un “_” y debe ser seguido por cualquier cantidad de “Letras”, “_” o “Numeros”. Los números se forma de la manera que se mostró en los ejemplos anteriores.

El “Procedimiento” se compone de un “Nombre” y un “Bloque”. El “Bloque” está constituido por un corchete que abre “[”, “Instrucciones” y un corchete que cierra “]”. Las “Instrucciones” pueden ser de dos tipos, de “Decisión” o “Cíclicas”.

Las de “Decisión” consisten en una oración que tiene: la cadena “if”, un paréntesis que abre “(“, un “Nombre”, un “OperRel” (Operador Relacional), un “Nombre”, un paréntesis que cierra “)”. Después debe venir un “Nombre” seguido de un terminal “;” después la cadena “then” y finalmente un “Nombre” seguido de un “;”. Nótese que la regla de producción etiquetada con el símbolo no-terminal “Instrucciones” es una regla recursiva, lo que permite que se puedan escribir entre los corchetes del bloque cualquier cantidad de “Instrucciones”, ya sea “Cíclicas” o de “Decisión”.

CAPÍTULO IV

CONCLUSIONES

El desarrollo de compiladores es una de las ramas de investigación más estudiadas de la informática. Las actualizaciones de los lenguajes computacionales existentes se hacen cada vez con más frecuencia y la creación de nuevos lenguajes más poderosos, más amistosos y más sencillos es una tarea primordial para todo país que quiera formar parte de este vertiginoso avance de la ciencia.

Mediante la aplicación de la teoría de los lenguajes formales se ha desarrollado este proyecto, que es un sistema que permite generar de manera automática el reconocedor de cualquier lenguaje que sea posible describir mediante una gramática libre de contexto. Con esto se resuelve el problema de la codificación detallada que se tiene que llevar a cabo cuando se desarrolla un reconocedor manualmente y que en muchas ocasiones se convierte en una tarea complicada.

De esta manera, aquellas personas interesadas en el desarrollo de lenguajes computacionales, tienen la libertad de centrar su atención en el diseño de la gramática del lenguaje, que equivale a pensar en su estructura, su léxico, su alfabeto, etc. y olvidarse de los detalles de cómo programar dicho conocimiento. En otras palabras, enfocarse a las tareas ejecutivas y dejar que este sistema se ocupe de las tareas operativas del desarrollo de un reconocedor. Esto permite ahorrar tiempo de desarrollo lo cual es valioso si se piensa en el tiempo de un investigador y con ello puede dirigir su esfuerzo a su investigación más que a la programación.

La descripción de los fenómenos lingüísticos es una tarea que requiere muchos recursos, principalmente investigadores que logren que México se posicione globalmente en esta área. Mucho falta por hacer, y más aún en un país como México en donde se olvidan temas tan importantes como éste, lo que provoca una dependencia tecnológica con los países desarrollados, quienes involucran a una buena cantidad de científicos en el desarrollo de herramientas computacionales con el fin de mantenerse en la competencia global.

Las gramáticas se han constituido como una herramienta ideal para la descripción de los lenguajes que se utilizan para modelar la realidad, quizá puedan ser utilizadas para describir cualquier fenómeno que acontezca en el Universo, esto sólo se podrá comprobar hasta que se haga un uso exhaustivo de ellas. Algunos campos en los que se han utilizado de manera experimental y en donde hay pocos investigadores alrededor del mundo involucrados, es en el desarrollo de modelos de las artes cómo la música o la pintura; pero también se necesitan desarrollar más aplicaciones de las gramáticas en las ciencias del cosmos, en la creación de poesía, etc. La idea es extender la aplicación de la lingüística matemática a todos los campos de la actividad humana.

El éxito de este proyecto será mayor si logra motivar a que más personas se involucren en el desarrollo de herramientas informáticas que hagan más sencilla la vida humana en la *era del conocimiento y la información*.

ANEXO A
CÓDIGO DEL SISTEMA

CÓDIGO PROGRAMA GS.CPP

Desarrollado por Horacio Alberto García Salas

```
void CargaGramDim2()
{
int xbuf=0;
  GramCpy[yy++][xx]='S';
  Vn[yyvn++][xxvn]='#';
  while( Buffer[xbuf] != '\x0' )
  {

   if( Buffer[xbuf]!='\r' )
   {
    if( Buffer[xbuf]=='\n' )
    {
        yy++;
        xx=0;
        izq=1;
    }
    else
    {
        if( izq == 0 )
          GramCpy[yy][xx++]=Buffer[xbuf];
        else
         if( Buffer[xbuf]!='-' && Buffer[xbuf]!='>' && Buffer[xbuf]!=' ' )
           Vn[yyvn][xxvn++]=Buffer[xbuf];
        if( Buffer[xbuf] == '>' && izq == 1 )
        {
          izq = 0;
          yyvn++;
          xxvn=0;
        }
    }
   }
   xbuf++;
  }
  xbuf=0;
}
```

```
void IndicaVn()
{
char *ptr;
char *ptr2;
//char *ptr3;
//int tam;

  xx=0;
  yy=0;
  while( GramCpy[yy][xx] != '\x0' )
  {
    ptr = Gram[yy];
    ptr2 = GramCpy[yy];
    while( *ptr2 != '\x0' )
    {
      yyvn = xxvn = 0;
      while( Vn[yyvn][xxvn] != '\x0' && !terminal )
      {
           if( !strncmp( ptr2,Vn[yyvn],strlen(Vn[yyvn]) ) )
           {
             ptr2 += strlen(Vn[yyvn]);
             *ptr++ = 'Å'; //Alt 29
             *ptr++ = yyvn;
             *ptr++ = 'Æ'; //Alt 30
             terminal++;
           }
           yyvn++;
      }
      if( !terminal )
      {
      *ptr = *ptr2;
      ptr++;
      ptr2++;
      }
      terminal = 0;
    }
    yy++;
  }

}
```

```
int BuscaOr(int reglacpy)
{
  while( Gram[reglacpy][xx] != '|' && Gram[reglacpy][xx] != '\x0' )
    xx++;
  if( Gram[reglacpy][xx] == '|' )
    return(1);
  else
    return(0);
}
```

```
void Generador()
{
  xx=0;
  yy=0;
  xxvn=0;
  yyvn=0;
  LeeGram(0);
  if( !Err  && ProGram[xxpr] == '\x0' )
    Form1->RichEdit2->Lines->Add("Felicidades este programa \"Si\" pertenece al lenguaje");
  else
  {
    Form1->RichEdit2->Lines->Add("Programa que \"No\" pertenece al lenguaje");
    Form1->RichEdit2->Lines->Append("El símbolo");
    Form1->RichEdit2->Lines->Append(ProGram[xxpr]);
    Form1->RichEdit2->Lines->Append("No pertenece al lenguaje o está en posición equivocada");
    Form1->RichEdit2->Lines->Append("En la ventana izquierda, el símbolo \"¶\" indica la posición
en donde ha ocurrido el error");
    ProGram[xxpr] = '¶'; //Alt 244
    Form1->RichEdit3->Clear();
    Form1->RichEdit3->Lines->SetText(ProGram);
    Form1->RichEdit3->DefAttributes->Color = clRed;
    Form1->RichEdit3->Visible=true;
  }
}
```

```
int LeeGram(int regla)
{
  Err = 0;
  xx = 0;
  primsimb = 1;
  while( ProGram[xxpr] != '\x0' && Err == 0 && Gram[regla][xx] != '\x0' && Gram[regla][xx] !=
'|' && vacia==0 )
  {

    if ( ProGram[xxpr] == '\r' || ProGram[xxpr] == '\n' )//|| ProGram[xxpr] == ' ' )
          xxpr++;
          else

    if( Gram[regla][xx] == ProGram[xxpr] )
    {
       xx++;
      xxpr++;
      primsimb = 0;
    }
    else

     if( Gram[regla][xx] == 'Å' && Gram[regla][xx+2] == 'Æ' )
     {
          PilaRec[ptrpila++]=regla;
          PilaRec[ptrpila++]=xx+3;
          primsimb = 0;
          LeeGram( Gram[regla][++xx] );
     }
     else

          if( Gram[regla][xx] == 'Ø' )
            vacia=1;
          else

             if( ProGram[xxpr] == ' ' )
               xxpr++;
            else

               Err = 1;
```

```
  if( Err == 1 && primsimb )
    if( BuscaOr(regla) )
    {
         xx++;
         Err = 0;
    }
}
if( ProGram[xxpr] == '\x0' && Gram[regla][xx] != '\x0' && Gram[regla][xx] != '|' )
{
  Form1->RichEdit2->Lines->Add("Programa Incompleto\n");
  Err = 1;
}
xx=PilaRec[—ptrpila];
PilaRec[ptrpila]='\x0';
regla=PilaRec[—ptrpila];
PilaRec[ptrpila]='\x0';
vacia=0;
return(Err);
```

ANEXO B
SINTAXIS DE LA GRAMÁTICA

La gramática de un lenguaje se describe por medio de símbolos terminales, símbolos no-terminales, símbolo de inicio y reglas de producción G={Vt, Vn, S, R}. El símbolo “S” se utiliza frecuentemente como el símbolo de inicio, por lo que en este sistema se será utilizado de esta manera. Esto implica que el símbolo “S” no puede ser utilizado como parte de ningún otro símbolo del lenguaje.

Las reglas de producción deben de iniciar con un y sólo un símbolo no-terminal, seguido del símbolo de producción (→), este símbolo se representa por medio de uno o más guiones y un símbolo “>”. Posteriormente, se describe el lado derecho de las reglas de producción. Por ejemplo:

Instrucciones → Decision | Ciclicas

Los símbolos no-terminales (Vn) se forman con cualquier cantidad de letras mayúsculas o minúsculas, números y guiones bajos. Aunque realmente se puede utilizar cualquier caracter como parte de los símbolos no-terminales, se han limitado, con el fin de no complicar más el sistema, a las letras, números y guiones bajos. Los siguientes son símbolos no-terminales válidos: Regla1, 2006Car, _Proy24, deCision.

Una vez que se ha etiquetado la regla de producción con un símbolo no-terminal y después de haberse escrito el símbolo de producción, se describe el lado derecho de la regla. El lado derecho de las reglas de producción está constituido por símbolos terminales y no terminales, los símbolos terminales (Vt) son cualquier cadena de letras, números o caracteres. No debe haber espacios en blanco entre los símbolos. La única diferencia que hay entre los símbolos terminales y los no-terminales, además de que los no-terminales no pueden mas que tener letras, números y guiones bajos, es que los símbolos no-terminales se encuentran del lado izquierdo de las reglas de producción. De esta manera, para identificar un símbolo no-terminal basta verificar si pertenece o no al lado izquierdo de alguna de las reglas de producción de la gramática.

S →e|aA

A →a|bA|cS|dAA

Una sola regla puede tener varias opciones de producción, éstas se separan por medio del símbolo "|" (or lógico). Cada una de las opciones de una regla de producción debe empezar con un símbolo diferente, esto quiere decir que debe estar factorizada.

Regla	Incorrecta (sin factorizar)	Correcta (factorizada)
	S →eB|eS	S →eA
		A →B|S
		B →fin

Otro símbolo frecuentemente utilizado es el de la cadena vacía "Ø". Como su nombre lo indica este símbolo representa la cadena que se compone de cero símbolos y es una opción útil cuando es utilizada en las reglas recursivas, pues permite detener la recursividad. Por ejemplo, la siguiente gramática permite generar nombres que empiecen con letra minúscula o guion bajo, seguidos de cualquier cantidad de letras, guiones bajos o números

Nombre →LetraX3|_X3

Letra →a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

Num →0|1|2|3|4|5|6|7|8|9

X3 →LetraX3|_X3|NumX3|Ø

FUENTES DE INFORMACIÓN

[1] Formal Languages. Arto Salomaa. Academic Press, New York, 1973. Revised edition in the series “Computer Science Classics”, Academic Press, 1987.

[2] Parsing Techniques - A Practical Guide. Dick Grune and Ceriel J.H. Jacobs Ceriel Jacobs <ceriel@cs.vu.nl>. Vrije Universiteit, Amsterdam, The Netherlands.
http://www.cs.vu.nl/~dick/PTAPG.html.
Fecha de Consulta 5/V/2006.

[3] Un algoritmo de Inferencia Gramatical mediante Corrección de Errores. Reconocimiento de Formas. Inferencia Gramatical
http://www.uv.es/~hmr/Tesis/index.html
Fecha de Consulta 5/V/2006.

[4] On Patterns and Graphs. Brian Mayoh.
http://www.daimi.au.dk/PB/484/PB-484.pdf
Fecha de Consulta 5/V/2006.

[5] Grammar systems theory.
http://sziami.cs.bme.hu/~csima/phd1/node1.html
Fecha de Consulta 5/V/2006.

[6] Prhlt Thesis.
http://www.iti.upv.es/~prhlt/thesis/
Fecha de Consulta 5/V/2006.

[7] Compiladores.
http://www.monografias.com/trabajos11/compil/compil.shtml
Fecha de Consulta 5/V/2006.

[8] Word Grammar a brief introduction for graduate students. Richard Hudson.
http://www.phon.ucl.ac.uk/home/dick/WG/WG4PG/intro.htm
Fecha de Consulta 5/V/2006.

[9] Teoría de Lenguajes y Autómatas. Conceptos y teoremas fundamentales. Angeles Manjarrés Riesco
http://www.ia.uned.es/asignaturas/automatas-1/

[10] Gramática transformacional.
http://es.wikipedia.org/wiki/Gram%C3%A1tica_transformacional
Fecha de Consulta 5/V/2006.

[11] Gramaticas Formales
http://delta.cs.cinvestav.mx/~gmorales/ta/node1.html
Fecha de Consulta 5/V/2006.

[12] Chomsky y la Gramática Generativa. Miguel Angel Aguilar Alconchel
http://www.csi-csif.es/andalucia/ense/revista/Articulos/N_7_04_V3/Chomsky.PDF

[13] Un acercamiento a la realidad humana desde el lenguaje. Guillermo Brand Deisler.
http://www.redcientifica.com/doc/doc200106080001.html
Fecha de Consulta 5/V/2006.

[14] Inteligencia a partir de agentes tontos. Cristobal Baray y Kyle Wagner.
http://www.acm.org/crossroads/espanol/xrds5-4/dumbagents.html
Fecha de Consulta 5/V/2006.

[15] Computers and Linguistics Computers and Translation. Syntax Analisys and Theory of Grammar. Nico Weber.
http://www.spr.fh-koeln.de/Personen/Weber/weber_lectures/tomsk_lectures/Syntax.html

[16] Clasificación de Gramáticas.
http://espanol.geocities.com/lenguajesautomatasitq/gram.html

[17] Maestria en Inteligencia Artificial- Trabajos de Tesis -
Uso de gramáticas de rasgos para reconocimiento de oraciones en español. Arturo Nandayapa Parada.
http://www.lania.mx/biblioteca/newsletters/1997-otono-invierno/gramatic.html

[18] Fases de un Compilador.
http://www.icmc.usp.br/~gracan/download/sce126/SubItem23SCE126.html
Fecha de Consulta 5/V/2006.

[19] Random Walks, Markov Chains and the Monte Carlo Method.
http://www.taygeta.com/rwalks/rwalks.html
Fecha de Consulta 5/V/2006.

[20] Wikipedia, la enciclopedia libre
http://es.wikipedia.org/wiki/Compilador
Fecha de Consulta 5/V/2006.

[21] 402 Procesadores de Lenguajes I 96/97.
http://www.lcc.uma.es/docencia/ETSIInf/pl/pl1.html

[22] First Principles of Physio Informatic Systems
http://www.quasar.org/memes/phd-meme-mill/2llu4phd/phd-please-1-31-00.htm

[23] Chomsky and the Universal Grammar.
http://www.southerncrossreview.org/9/chomsky.htm
Fecha de Consulta 5/V/2006.

[24] Cómo crear un lenguaje.
http://www.geocities.com/finis_stellae/sp/lng/como/index.html

[25] Yacc Yet Another Compiler Compiler.
http://www.lcc.uma.es/docencia/ETSIInf/pl/apuntes/pl1/lecc48.html

[26] Compiladores y lenguajes de programación gratuitos
http://www.geocities.com/pretabbed/compiladores.htm
Fecha de Consulta 5/V/2006.

[27] Papers acerca de lenguajes.
http://citeseer.ist.psu.edu/context/220350/0
Fecha de Consulta 5/V/2006.

[28] Polynomial-Complexity Deadlock Avoidance Policies for
Sequential Resource Allocation Systems. Spiridon A. Reveliotis
http://gilbreth.ecn.purdue.edu/~malawley/journalp5_1.pdf
Fecha de Consulta 5/V/2006.

[29] Lingüística Cartesiana: Un Capítulo Polémico de la Historia de la Lingüística. Xavier Laborda Gil.
http://www.sant-cugat.net/laborda/pdf%20h11%20Ling%C3%BC%C3%ADstica%20cartesiana.pdf
Fecha de Consulta 5/V/2006.

[30] Agentes Autónomos Inteligentes.
http://www.redcientifica.com/doc/doc199903310001.html
Fecha de Consulta 5/V/2006.

[31] Red Científica, Ciencia, Tecnología y Pensamiento.
http://www.redcientifica.com/
Fecha de Consulta 5/V/2006.

[32] Compiladores e Interpretes.
http://www.ucse.edu.ar/fma/compiladores/
Fecha de Consulta 5/V/2006.

[33] Generalised Recursive Descent Parsing and Follow Determinism.
http://www.dcs.rhbnc.ac.uk/research/languages/projects/rdp.shtml

[34] Patterns and Pattern Languages. Douglas C. Schmidt, Ralph E. Johnson, Mohamed Fayad.
http://www.cs.wustl.edu/~schmidt/CACM-editorial.html
Fecha de Consulta 5/V/2006.

[35] La humanización de la Lingüística estructural: Los problemas de Lingüística general de Émile Benveniste.
José María Jiménez Cano (Universidad de Murcia).
http://www.um.es/tonosdigital/znum7/peri/peri.htm
Fecha de Consulta 5/V/2006.

[36] L-Systems (Lindenmayer).
http://matap.dmae.upm.es/cursofractales/capitulo2/1.html
Fecha de Consulta 5/V/2006.

[37] Teorías sobre la adquisición del lenguaje.
http://www.apsique.virtuabyte.cl/tiki-index.php?page=ApreLenguaje
Fecha de Consulta 5/V/2006.

[38] Precc - A Prettier Compiler-Compiler.
http://vl.fmnet.info/precc/
Fecha de Consulta 5/V/2006.

[39] Accent Compiler-Compiler.
http://accent.compilertools.net
Fecha de Consulta 5/V/2006.

[40] Computacional Linguistics and Intelligent Text Processing
6th International Conference, CicLing 2005 Mexico City.
Alexander Gelbukh

[41] La Cátedra de Diseño de Compiladores
http://www.frsf.utn.edu.ar/universidad_virtual/catedras/sistemas/electivas/compiladores/archive/ponencia.doc

[42] Principia El Lenguaje de Programación STOP
http://homepage.mac.com/eravila/Stop/stop01.html
Fecha de Consulta 5/V/2006.

[43] Estructuras Sintácticas. Noam Chomsky. Siglo Veintiuno Editores 13ª edición.

[44] Programming Languages Principles and Practice. Kenneth C. Louden. Ed. Thompson Book/ Cole. Second Edition.

[45] Programming Languages Design and Implementation. Terrence W. Pratt, Marvin V. Zelkowitz. Ed. Prentice Hall. Third Edition.

[46] Tesis. Aplicación de los Sistemas Evolutivos a la Composición Musical. Horacio Alberto García Salas. UPIICSA-IPN.

[47] Konrad Zuse
http://ei.cs.vt.edu/~history/Zuse.html
Fecha de Consulta 5/V/2006.

[48] El Problema del Año 2000 Un Recuento Final. M. en C. Eduardo René Rodríguez Ávila
http://homepage.mac.com/eravila/y2k.pdf
Fecha de Consulta 5/V/2006.
[49] El problema del año 2000 Y2K
http://www.geocities.com/CollegePark/Dorm/7635/economic.html
Fecha de Consulta 5/V/2006.

[50] Introducción al Diseño de Compiladores y Lenguajes de Cómputo. M. en C. Eduardo René Rodríguez Ávila
http://www.chemedia.com/cgi-bin/smartframe/v2/smartframe.cgi?http://homepage.mac.com/eravila/indexold.html
Fecha de Consulta 5/V/2006.

[51] Semantics with Applications a Formal Introduction. Hanne Riis Nielson, Flemming Nielson.
http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.pdf
Fecha de Consulta 5/V/2006.

[52] Partial Evaluation and Automatic Program Generation. Neil D. Jones DIKU, Department of Computer Science, University of Copenhagen Universitetsparken 1, DK-2100 Copenhagen, Denmark. Carsten K. Gomard DIKU and Computer Resources International A/S Bregner dvej 144, DK-3460 Birker, Denmark. Peter Sestoft Department of Computer Science, Technical University of Denmark Building 344, DK-2800 Lyngby, Denmark
http://www.dina.kvl.dk/~sestoft/pebook/jonesgomardsestoft-letter.pdf
Fecha de Consulta 5/V/2006.

[53] Writing Compilers and Interpreters. Ronald Mak. Wiley Computer Publishing 1996.

[54] Breve Historia de los lenguajes de Programación
http://www.cibercalli.com/index.pl/erick/news/breve-historia-de-los-lenguajes-de-programacin
Fecha de Consulta 5/V/2006.

[55] A Slightly Skeptical View on Scripting Languages
http://www.softpanorama.org/Scripting/index.shtml
Fecha de Consulta 5/V/2006.

[56] Manual de Programación de Sistemas I. Lic. Martha Martínez Moreno
http://www.itver.edu.mx/comunidad/material/Progra_Sistemas_I/Manual_Completo_de_Programacion_de_Sistemas_I.doc
Fecha de Consulta 5/V/2006.

[57] Compiladores. Traductores y Compiladores con lex/yacc, jflex/cup y javacc. Sergio Gálvez Rojas, Miguel Ángel Mora Mata
http://www.lcc.uma.es/~galvez/ftp/libros/Compiladores.pdf
Fecha de Consulta 5/V/2006.

[58] Compilers.net. Gramáticas, Compiladores, Metacompiladores, Papers
http://www.compilers.net/
Fecha de Consulta 5/V/2006.

[59] Compiler Construction. Ryan Stansifer Department of Computer Sciences Florida Institute of Technology Melbourne, Florida USA
http://www.cs.fit.edu/~ryan/cse4251/chapter3.pdf
Fecha de Consulta 5/V/2006.

[60] The Lex & Yacc Page
http://dinosaur.compilertools.net/
Fecha de Consulta 5/V/2006.

[61] Fundamentos de Compiladores. Karen A. Lemone. Ed. CECSA 1996.

[62] Historia de la Informática
http://elvex.ugr.es/decsai/java/pdf/1B-Historia.pdf
Fecha de Consulta 5/V/2006.

[63] Introduction to Automata Theory, Languages and Computation. John E. Hopcroft, Jeffrey D. Ullman. Addison-Wesley Publishing Company. 1979.

[64] Theory of Computation Formal Languages, Automata, and Complexity. J. Glenn Brookshear. The Benjamin/Cummings Publishing Company, Inc. 1989.

[65] Teoría de Autómatas y Lenguajes Formales. Pedro García, Tomás Pérez, José Ruiz, Escarna Segarra, José M. Sempere, M. Vázquez de Parga. Alfaomega Grupo Editor. 2001.