TRADUCTOR DE MODELOS DE SIMULACIÓN GALATEA A CÓDIGO JAVA

Autor: Yaritza Vargas

Tutores: Kay A. Tucci

Mayerlin Y. Uzcátegui

Trabajo presentado como requisito parcial para la obtención del grado de
INGENIERO DE SISTEMAS

UNIVERSIDAD DE LOS ANDES
MÉRIDA, VENEZUELA
Mérida, Noviembre 2003

Índice General

Índice de Figuras

Índice de Ejemplos

Resumen

Este documento presenta el prototipo del traductor, que permite establecer una conexión entre los modelos creados en el lenguaje de simulación GALATEA y el lenguaje de propóposito general Java, con el fin de establecer el puente de uni(ó)n de entre la Plataforma de simulaci(ó)n GALATEA y su interfaz gr(á)fica. El proceso de traducción, en las fases de análisis léxico y sintáctico, fue implementado por medio de las herramientas JFLex y CUP, respectivamente, las cuales producen códigos que se integran al traductor prototipo. Las etapas de análisis semántico, optimización de código y generación de código quedan para ser tratadas por el compilador de Java.

En este documento también se describe la sintaxis del lenguaje de simulación GALATEA explicando como se extiende la sintaxis de GLIDER. Para efectos instruccionales, se presentan versiones simples de las reglas gramaticales que definen la estructura de los programas GALATEA, en una variante de la notación BNF tal como es usada por la herramienta CUP. Versiones más complejas de esas reglas, pero con los mismos recursos de traducción se explican al final del documento mostrando como se usan para traducir un modelo GLIDER normalmente usado como referencia: El ejemplo de las taquillas del banco.

Palabras Clave:

* Simulación
* Computadoras Digitales

Agradecimiento

Este proyecto debe su culminación al apoyo y constancia de los profesores Kay Tucci, Mayerlin Uzcátegui y Jacinto Dávila.

Al ser m(á)s importante de mi vida, qui'en día a día ha demostrado que el gran amor de una madre supera barreras con grandes sacrificios pero gratas recompensas.

A mi esposo por su amor, entera disposición y ayuda incondicional.

Al mejor de los hermanos, quién por ser como es ha logrado inspirar en mi un profundo respeto y un inmenso cariño, eres como un padre para mi.

Al grupo de trabajo del Sistema Unificado de Microcomputación Aplicada (SUMA) por brindarme la mayor de las colaboraciones.

A todas aquellas personas y dependencias, que de una u otra forma intervinieron en alcanzar esta meta, en especial al CDCH, CESIMO y SUMA, a quienes se debe el financiamiento de este trabajo.

Yaritza Vargas

"No importa tanto el tiempo que logremos en alcanzar nuestros sueños, sino el disfrutarlos"

Introducción

Este documento describe el traductor para el lenguaje de simulación llamado GALATEA[1] [1]. La idea de traducción tiene sus orígenes en el año 1946, cuando surgió el primer computador digital, en el que se manejaban tan sólo códigos numéricos. Este proyecto pronto fue reemplazado por códigos más fáciles de manejar y hoy en día, la escritura de un buen compilador puede lograrse satisfactoriamente con el uso de herramientas de software y técnicas menos obscuras.

Comúnmente se designa a un traductor como un programa que convierte programas fuentes, escritos en un lenguaje, en programas objetos equivalentes, escritos en otro lenguaje. La traducción se particiona en dos análisis y síntesis.

La parte de análisis descompone al programa fuente en elementos básicos, creando con ellos una representación intermedia del programa fuente. El análisis comprende tres fases:

1. Análisis Léxico, lineal o de exploración, en la que la cadena de caracteres que constituye el programa fuente se leen de izquierda a derecha agrupando las secuencias de caracteres que tienen un significado colectivo. A estas secuencias se les conoce, como componentes léxicos ó lexemas.

[1]GALATEA: *GLIDER with Autonomous, Logic-based Agents, TEmporal reasoning and Abduction.*

GLIDER: plataforma de simulación que permite la especificación de modelos de simulación para sistemas continuos y de eventos discretos

2. Análisis Jerárquico o Sintáctico, implica agrupar los componentes léxicos del programa fuente en frases gramaticales, que el traductor utiliza para sintetizar la salida.

3. Análisis Semántico, en el que se realizan ciertas revisiones al programa fuente para tratar de encontrar errores semánticos. Generalmente en esta fase se utiliza la información de los tipos de datos para verificar la compatibilidad en las expresiones.

La síntesis, construye el programa objeto deseado, a partir de la representación intermedia, siendo ésta la que requiere de técnicas más especializadas. Es decir, la dificultad principal del traductor recae en esa compleja fase de análisis que produce la representación intermedia [**?**].

El traductor de modelos de simulación GALATEA a código Java, tiene el propósito de convertir modelos de simulación, escritos en el lenguaje de modelado GALATEA al lenguaje de propósito general Java. En este último lenguaje es en el que se ha desarrollado la plataforma de simulación GALATEA. Adicionalmente con la implementación del traductor se pretende formalizar la especificación sintáctica del lenguaje de simulación GALATEA que está directamente basada en la de su predecesor GLIDER[2], el cual no cuenta con una especificación sintáctica completa.

En la medida que la práctica del Modelado y Simulación de Sistemas continúa evolucionado de manera progresiva surgen nuevas exigencias para los lenguajes de simulación. En los últimos años, una de esas exigencias ha sido la de incorporar, el uso de entidades, llamadas Agentes, que perciben, razonan y actúan sobre su entorno; además de integrar los conceptos y herramientas para la simulación de los enfoque distribuido, interactivo, discreto y continuo. Ese es el desafío que se aborda con el proyecto de construcción de la plataforma de simulación GALATEA.

Para la implementación del prototipo, se seleccionó como lenguaje el Java [3], por ser el lenguaje matriz del primer prototipo de GALATEA[1], y por ofrecer características de sencillez, potencia, seguridad, eficacia, portabilidad y universalidad (independencia de sistemas operativos). Las herramientas empleadas en este trabajo

para el análisis léxico y sintáctico, que reconocen la sintaxis GALATEA y generan el código Java, son respectivamente, JFLex[2] y CUP[3]. Ambos pueden ser usados en las plataformas para las que exista la JVM[4] y su instalación no presenta mayores complicaciones.

Este documento se encuentra estructurado de la siguiente manera:

* El capítulo 1 contiene un marco teórico en el que se presenta el proyecto GALATEA y se enmarca el traductor desarrollado en esta tesis dentro del esquema general del proyecto, también se da a conocer una breve cronología de las teorías y conceptos de compilación.

* La gramática de GALATEA. se describe en el capítulo 2.

* En el capítulo 3 se describe el proceso de traducción explicando el traductor desarrollado a través de un ejemplo.

* En los apéndices, se muestra el manual del analista que indica el proceso de creación del traductor GALATEA a partir del código léxico y sintáctico y un segundo manual, el del usuario, que muestra como usar el código obtenido.

[2]JFLex: *Java Format Lex.*
[3]CUP: *Constructor of Useful Parsers.*
[4]JVM: *Java Virtual Machine*, máquina virtual de java

Capítulo 1

Marco Teórico

1.1 Proyecto GALATEA

La plataforma de simulación GALATEA pretende aprovechar el esfuerzo de las investigaciones realizadas en la Universidad de Los Andes (CESIMO — IEAC — SUMA) en las áreas de simulación, modelado y desarrollo de software.

GALATEA surge como un proyecto para incorporar o extender la plataforma GLIDER con los conceptos y herramientas que permitan simular sistemas bajo los enfoques distribuido, interactivo, continuo, discreto y combinado. Igualmente, esta nueva plataforma incorpora el soporte para modelado y simulación de sistemas multiagentes, así como también una serie de módulos que permiten la interacción entre los modelos desarrollados con GALATEA y un conjunto de sistemas externos como por ejemplo, Base de Datos, Sistemas de Información Geográficos, Herramientas de Visualización entre otras[1].

GLIDER, lenguaje de simulación, desarrollado en la Universidad de Los Andes, que se basa en el formalismo DEVS[1] que incluye herramientas para el modelado y simulación de sistemas continuos, además presenta facilidades, en su diseño, para la ejecución simultánea y cooperativa de ambos tipos de sistemas.

Este lenguaje representa el sistema modelado, en subsistemas o nodos, que intercambian información, a través de mensajes. Individualmente un subsistema puede almacenar, tranformar, transmitir, crear y eliminar información del estado del sistema, mediante un código enlazado, que precisa el cambio de los estados del sistema, cuando ocurre un evento en un subsistema.

1.1.1 Características de GALATEA

Un programa GALATEA, es una extensión sintáctica de un programa GLIDER. Además de las secciones: `TITLE`, `NETWORK`, `INIT`, `DECL`, que comúnmente se usan en GLIDER, GALATEA incorpora dentro de su estructura a las secciones `AGENTS` e `INTERFACE`(ver figura **??**). La plataforma de Simulación GALATEA, sigue el diseño propuesto en [4, 1] y está conformada por:

* Una familia de lenguajes que permiten simular sistemas multi-agentes. GALATEA no es un solo lenguaje para crear modelos monolíticos. Es una familia de lenguajes que incluye a una versión extendida de GLIDER, en el cual se concentra esta tesis, y a varios lenguajes de programación lógica para modelar los agentes [5, 6]

* Un compilador de ese GLIDER extendido, desarrollado en Java.

* Un ambiente para la construcción de modelos.

* Un motor de inferencia que funciona como interpretador independiente de los programas de cada agente.

[1] *DEVS: Discrete Event Systems Specifications*

* Un conjunto de módulos para la comunicación con sistemas externos.

	ENCABEZADO
TITLE	*TÍTULO DEL MODELO*
NETWORK	*DESCRIPCIÓN DE LA RED*
AGENTS	*DESCRIPCIÓN DE LOS AGENTES*
GOALS	*METAS*
BELIEFS	*CREENCIAS*
NETWORK	*PREFERENCIAS*
INTERFACE	*DESCRIPCIÓN DE LAS RELACIONES ENTRE LOS AGENTES Y EL AMBIENTE*
INIT	*VALORES INICIALES PARA VARIABLES Y ESTRUCTURAS*
DECL	*DECLARACIÓN DE VARIABLES*
END.	

Figura 1.1: Estructura de un programa GALATEA

1.1.2 Estructura prevista para los modelos GALATEA

Como muestra la figura **??** el programa comienza con un *ENCABEZADO* que permite describir al sistema y hacer algunos comentarios.

TITLE: Igual que en GLIDER, ésta sección permite incorporar un título al modelo.

NETWORK: Igual que en GLIDER, salvo porque se ha propuesto admitir instrucciones en Java adem(á)s del PASCAL original de GLIDER, es la sección donde se describen las relaciones entre los nodos y el código que simula su comportamiento. Los nodos poseen dos partes un encabezado donde se definen el nombre, el tipo, la

multiplicidad, los sucesores y las variables locales y un código escrito en GLIDER con instrucciones Java o PASCAL.

De los nodos podemos destacar las siguientes características:

* La constitución de cada red de nodos representa cada subsistema a ser estudiado en la simulación.
* Los hay de diferentes tipos:
  - **G** (Gate): Controla el flujo de Mensajes.
  - **L** (Line): Simula disciplinas de colas.
  - **I** (Input): Genera mensajes de entrada.
  - **D** (Desicion): Selecciona mensajes según reglas de selección indicadas.
  - **E** (Exit): Destruye los mensajes.
  - **R** (Resource): Simula recursos usados por los mensajes (entidades).
  - **C** (Continuous): Resuelve sistemas de ecuaciones diferenciales ordinarias de primer orden.
  - **A** (Autonomous): Permite programar eventos.
  - **O** (Other): Permite al usuario definir sus propios nodos.
* De acuerdo al código asociado a cada nodo, la información es descrita y luego ejecutada.
* El intercambio de información entre los nodos es realizado a través del código haciendo uso de pase de mensajes, variables y archivos compartidos.
* La información se almacena en variables, archivos o listas de mensajes.
* Cada nodo tiene asociado un conjunto de nodos predecesores de los que puede recibir mensajes y un conjunto de nodos sucesores hacia los cuales puede enviar mensajes.

La sección **AGENTS** es el primer anexo a la estructura de un programa GLIDER. En ella se detalla cada tipo de agente, como se define en [4], creando códigos particulares según sea el tipo. La especificación interna del agente, el estado mental (base de conocimiento, procedimientos, metas y preferencias) se incluye en esta sección. Como se explican en la misma referencia anterior, `AGENTS` por sí misma es una colección de varios dispositivos de representación de conocimiento. Para cada tipo de agente, hay una subdivisión (`GOALS`) para describir las metas del agente, una para describir las creencias (`BELIEFS`) y otra para describir las preferencias (`PREFERENCES`). Cada una de las subdivisiones utiliza su propio lenguaje lógico. Según explican en [4], la riqueza de los lenguajes y la conveniencia de la lógica para capturar conocimiento declarativo y procedural justifica su presencia en esta parte del modelo del sistema: el código de los agentes. Además, creemos que estos lenguajes (presentados en [5] son la mejor forma de incorporar piezas de conocimiento humano a una máquina).

INTERFACE: Es la segunda sección adicional del programa GALATEA con respecto a GLIDER. En esta sección se incluye la descripción de las relaciones entre los agentes y el resto del sistema. Esta sección contiene el código Java[2] que describe el comportamiento del simulador para efectos de permitir a los agentes percibir y actuar en el ambiente que comparten. Es decir, se relaciona las acciones de los agentes y las percepciones de los estados actuales del mundo. Aquí el modelista define las acciones que el agente ejecutará o cuando y porqué un agente percibirá ciertas propiedades de su ambiente.

En esta sección, con toda la expresividad del marco de trabajo orientado a los objetos que ofrece Java, los programadores pueden especificar los efectos de las acciones de los agentes, incluyendo esas acciones que involucran a muchos agentes y, por consiguiente, puede involucrar efectos sinergéticos por acciones en paralelo por parte de los agentes.

[2]Este código será invocado por el simulador cuando quiera que sea necesario para provocar los cambios asociados con los agentes. El hecho de que sea código Java simplifica esta invocación pues elimina la necesidad de traducir esta sección. La solución definitiva todavía se discute en el grupo de desarrollo de GALATEA.

También aquí, el programador declara todo aquello que perciben los agentes en cada circunstancia en la que podrían estar involucrados. Note, además que, por la expresividad Java, el programador tiene acceso, en esta y en cualquier sección donde use Java, al estado *interno* de las otras estructuras del programa, en particular de los nodos. Esto no es posible en el PASCAL original del GLIDER.

INIT: Al igual que en GLIDER, indica que la sección de Iniciación de variables ha comenzado. Acá se realizan asignación de valores que serán usados en tiempo de ejecución de la simulación, tales como:

* Valores iniciales a arreglos.
* Valores iniciales para la capacidades de los nodos tipo R.
* Valores de funciones multivaluadas.
* Valores de parámetros en tablas de frecuencia.
* Valores de arreglos a partir de tablas de base de datos.
* Activación inicial de nodos.

DECL: La sección de Declaraciones se inicia aquí y cada declaración se representa a través de una etiqueta, como en GLIDER, acá se reconocen, entre otros,a `VAR` y `STATISTICS`.

El código GALATEA, para mantener la compatibilidad con GLIDER debe terminar con la palabra reservada `END` seguida de un punto (`END.`).

El lector debe notar que en este proyecto de tesis **no** se implementan las secciones `AGENTS` e `INTERFACE` en las gramáticas del traductor a Java. Las razones para esto son particulares a cada sección. La sección AGENTES tiene una sintaxis de lenguaje de programación lógica [6] para el cual ya se está elaborando un interpretador, asociado al motor de inferencia de cada agente, que es independiente del simulador GALATEA [**?**]. La sección `INTERFACE` se define en Java directamente, así que la traducción es casi trivial. Los detalles pendientes son todavía objeto de discusión en el Grupo Galatea.

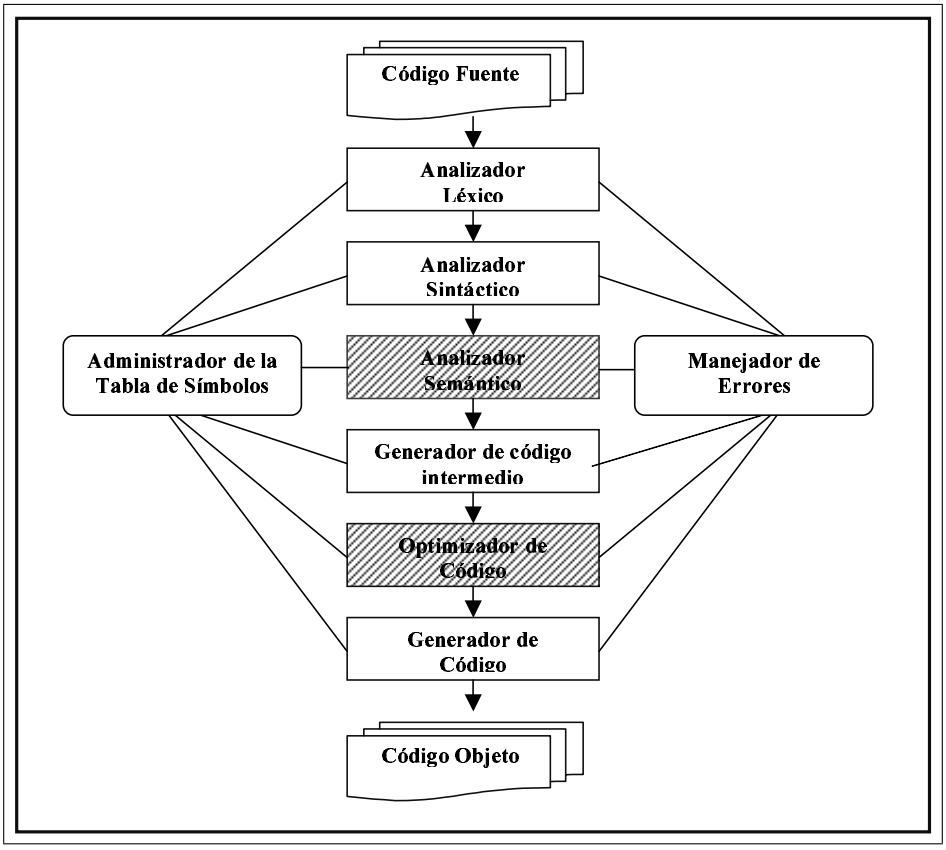

Figura 1.2: Fases del Traductor GALATEA

1.1.3 Objetivos

Los objetivos generales del proyecto GALATEA son:

* Implantar al Lenguaje GLIDER en una plataforma qiue favorezca su evolución.
* Proponer una teoría de simulación de sistemas multi-agentes.
* Definir una familia de lenguajes que permita simular sistemas multi-agentes.
* Diseñar la plataforma de simulación con la arquitectura adecuada para soportar distintos enfoques de simulación y que permita simular sistemas multi-agentes.
* Construir y mantener la plataforma de simulación.

1.1.4 Proyectos Desarrollados

La Plataforma de Simulación GALATEA, es la base de varios proyectos de grado, entre los que podemos mencionar:

1. Prototipo del Módulo GUI[3] para la Plataforma de Simulación GALATEA [7]
2. Diseño e Implementación de una estructura, para el soporte de simulación espacial en GLIDER[**?**]
3. Diseño de la plataforma de simulación de sistemas multi-agentes GALATEA[1]
4. Desarrollo de un prototipo del Módulo Agente para la Plataforma de Simulación GALATEA[**?**]
5. Integración de Sistemas de Información Geográficos a la Plataforma de Simulación GALATEA[8]

1.1.5 Ubicación del Traductor

La idea de crear el traductor, para convertir código GALATEA a Java, surge con la finalidad de conectar ésta plataforma de simulación de sistemas multi-agentes con su Interfaz Gráfica[7] que está desarrollada bajo el lenguaje de propósito general Java.

1.2 Teoría de Compilación

En 1946 se desarrolló el primer computador digital. En un principio, estas máquinas ejecutaban instrucciones consistentes en códigos numéricos, designado Lenguaje de Máquina, que señalan a los circuitos de la máquina los estados correspondientes a cada operación, pero resultaban muy engorrosos. Pronto aparecieron los llamados lenguajes ensambladores, que consistían en claves más fáciles de recordar, las cuales se generalizaron en cuanto se dio el paso decisivo de hacer que las propias máquinas

[3] *GUI: Graphis Usuaries Interfaz*

realizaran el proceso mecánico de la traducción. A este trabajo se le llama ensamblar el programa y tiene la tarea de programar cualquier función de una manera minuciosa e iterativa. Más tarde, las investigaciones se orientaron hacia la creación de un lenguaje que expresara las distintas acciones a realizar de una manera más sencilla para el hombre. En 1950, John Backus dirigió una investigación en IBM[4] acerca de un lenguaje algebraico que sirviera para describir la sintaxis de otros lenguajes en los términos más simples posibles, para que los humanos pudieran leerla, pero también rigurosos, para que pudieran ser procesados por el computador. En 1954 se empezó a desarrollar un lenguaje que permitía escribir fórmulas matemáticas de manera traducible por un computador, al que le llamaron FORTRAN[5]. Fue el primer lenguaje considerado de alto nivel. Permitía una programación más cómoda y breve que lo existente hasta ese momento, lo que suponía un considerable ahorro de trabajo. Surgió así por primera vez el concepto de un traductor, como un programa que traducía un lenguaje a otro lenguaje. En el caso particular de que el lenguaje a traducir es un lenguaje de alto nivel y el lenguaje traducido de bajo nivel, se emplea el término compilador. La tarea de realizar un compilador no fue fácil. El primer compilador de FORTRAN tardó muchos años en concretarse a pesar de ser muy sencillo. Como un ejemplo de ello se tiene el hecho de que los espacios en blanco fuesen ignorados, debido a que el periférico que se utilizaba como entrada de programas (una lectora de tarjetas perforadas) no contaba correctamente los espacios en blanco. Paralelamente al desarrollo de FORTRAN en América, en Europa surgió una corriente más universitaria, que pretendía que la definición de un lenguaje fuese independiente de la máquina y en donde los algoritmos se pudieran expresar de forma más simple. Con estas ideas surgió un grupo europeo encabezado por el profesor F. L. Bauer (de la Universidad de Munich) y en el que participó J. Backus como investigador, que definió un lenguaje de usos múltiples independiente de una realización concreta sobre una máquina, que en 1958, se llamó ALGOL[6] 58. En 1960 el lenguaje fue revisado

[4]IBM: *International Bussines Machine.*
[5]FORTRAN: *FORmulae TRANslator.*
[6]ALGOL: *ALGOritmic Language.*

y llevó a una nueva versión que se llamó ALGOL 60. La versión actual es ALGOL 68, un lenguaje modular estructurado en bloques y en donde figuran por primera vez muchos de los conceptos de los nuevos lenguajes algorítmicos:

* Definición de la sintaxis en notación BNF[7]
* Formato libre.
* Declaración explícita de tipo para todos los identificadores.
* Estructuras iterativas más generales.
* Recursividad.
* Paso de parámetros por valor y por nombre.
* Estructura de bloques, para determinar la visibilidad de los identificadores.

Junto a este desarrollo en los lenguajes, también se iba avanzando en la técnica de compilación. En 1958, Strong y otros proponían una solución al problema de que un compilador fuera utilizable por varias máquinas. Para ello, se dividía por primera vez el compilador en dos fases. La primera fase (*front end*) es la encargada de analizar el programa fuente y la segunda fase (*back end*) es la encargada de generar código para la máquina particular. El puente de unión entre las dos fases era un lenguaje intermedio que se designó con el nombre de UNCOL[8]. Para que un compilador fuera utilizable por varias máquinas bastaba únicamente modificar su *back end*. Aunque se hicieron varios intentos para definir el UNCOL, el proyecto se ha quedado simplemente en un ejercicio teórico. De todas formas, la división de un compilador en dos fases fue un adelanto importante. Para 1959 Rabin y Scott proponen el empleo de autómatas deterministas y no deterministas para el reconocimiento lexicográfico de los lenguajes. Rápidamente se aprecia que la construcción de analizadores léxicos a partir de expresiones regulares es muy útil en la implementación de los compiladores.

[7]BNF:*Backus-Naur Form.*
[8]UNCOL: *UNiversal Computer Oriented Language.*

En 1968, Johnson apunta diversas soluciones. En 1975, con la aparición de LEX[9] surge el concepto de un generador automático de analizadores léxicos a partir de expresiones regulares, adaptado para el sistema operativo UNIX. A partir de los trabajos de Chomsky (1954), se produce una sistematización de la sintaxis de los lenguajes de programación, y con ello un desarrollo de diversos métodos de análisis sintáctico. Con la aparición de la notación BNF – finalizada por Backus en 1960 cuando trabajaba en un borrador del ALGOL 60, modificada en 1963 por Naur y formalizada por Knuth en 1964 – se tiene una guía para el desarrollo del análisis sintáctico. Los diversos métodos de *parsers* ascendentes y descendentes se desarrollan durante la década de los 60. En 1959, Sheridan describe un método de *parsing* de FORTRAN que introducía paréntesis adicionales alrededor de los operandos para ser capaz de analizar las expresiones. Más adelante, Floyd introduce la técnica de la precedencia de operador y el uso de las funciones de precedencia. A mitad de la década de los 60, Knuth define las gramáticas LR[10] y describe la construcción de una tabla canónica de *parser* LR. Por otra parte, el uso por primera vez de un *parsing* descendente recursivo tuvo lugar en el año 1961. En el año 1968 se estudian y definen las gramáticas LL[11] así como los *parsers* predictivos. También se estudia la eliminación de la recursión a la izquierda de producciones que contienen acciones semánticas sin afectar a los valores de los atributos. En los primeros años de la década de los 70, se describen los métodos SLR[12] y LALR[13] de *parser* LR. Debido a su sencillez y a su capacidad de análisis para una gran variedad de lenguajes, la técnica de *parsing* o analizadores sintáctico LR va a ser la elegida para los generadores automáticos de *parsers*. A mediados de los 70, Johnson crea el generador de analizadores sintácticos YACC[14] para funcionar

[9]LEX: *LExical Analyzer Generator.*

[10]LR: Técnica ascendente de análisis sintáctico que consiste en examinar la entrada de izquierda a derecha (L) y construir una derivación por la derecha (R) del árbol asociado a la revisión de la gramática.

[11]LL: Técnica descendente de análisis sintáctico, que consiste en examinar la entrada de izquierda a derecha (L) y construir una derivación por la izquierda (L).

[12]SLR: .

[13]LALR:*Look Ahead LR*

[14]YACC: *Yet Another Compilere-Compiler.*

bajo un entorno UNIX. Junto al análisis sintáctico, también se fue desarrollando el análisis semántico. En los primeros lenguajes (FORTRAN y ALGOL 60) los tipos posibles de los datos eran muy simples, y la comprobación de tipos era muy sencilla. No se permitía la combinación de tipos, pues ésta era una cuestión difícil y era más fácil no permitirlo. Con la aparición del ALGOL 68 se permitía que las expresiones de tipo fueran construidas sistemáticamente. Más tarde, de ahí surgió la equivalencia de tipos por nombre y estructura. El manejo de la memoria como una implementación tipo pila se usó por primera vez en 1958 en el primer proyecto de LISP[15]. La inclusión en el ALGOL 60 de procedimientos recursivos potenció el uso de la pila como una forma cómoda de manejo de la memoria. Dijkstra introdujo posteriormente el uso de la instrucción *display* para acceso a variables no locales en un lenguaje de bloques. También se desarrollaron estrategias para mejorar las rutinas de entrada y de salida de un procedimiento. Así mismo, y ya desde los años 60, se estudió el pase de parámetros a un procedimiento por nombre, vlor y variable. Con la aparición de lenguajes que permiten la localización dinámica de datos, se desarrolla otra forma de manejo de la memoria, conocida por el nombre de *montículo*. Se han desarrollado varias técnicas para el manejo del *mont(í)culo* y los problemas que con él se presentan, como son las referencias perdidas y la recolección de basura. La técnica de la optimización apareció con el desarrollo del primer compilador de FORTRAN. Backus comenta cómo durante el desarrollo del FORTRAN se tenía el miedo de que el programa resultante de la compilación fuera más lento que si se hubiera escrito a mano. Para evitar esto, se introdujeron algunas optimizaciones en el cálculo de los índices dentro de un bucle. Pronto se sistematizan y se recoge la división de optimizaciones independientes de la máquina y dependientes de la máquina. Entre las primeras están la propagación de valores, el arreglo de expresiones, la eliminación de redundancias, etc. Entre las segundas se podría encontrar la localización de registros, el uso de instrucciones propias de la máquina y el reordenamiento de código. A partir de 1970 comienza el estudio sistemático de las técnicas del análisis de flujo

[15]LISP: *LISt Processing.*

de datos. Su repercusión ha sido enorme en las técnicas de optimización global de un programa. En la actualidad, el proceso de la compilación ya está muy asentado. Un compilador es una herramienta bien conocida, dividida en diversas fases. Algunas de estas fases se pueden generar automáticamente (analizador léxico y sintáctico) y otras requieren una mayor atención por parte del escritor de compiladores (las partes de traducción y generación de código). A pesar de lo que pueda pensarse, todavía se están llevando a cabo varias vías de investigación en este fascinante campo de la compilación. Por una parte, se están mejorando las diversas herramientas disponibles (por ejemplo, el generador de analizadores léxicos Aardvark para el lenguaje PASCAL[16] También la aparición de nuevas generaciones de lenguajes –ya se habla de la quinta generación, como de un lenguaje cercano al de los humanos – ha provocado la revisión y optimización de cada una de las fases del compilador. El último lenguaje de programación de amplia aceptación que se ha diseñado, el lenguaje Java, establece que el compilador no genera código para una máquina determinada sino para una virtual, que posteriormente será ejecutado por un intérprete, normalmente incluido en un navegador de Internet. El gran objetivo de esta exigencia es conseguir la máxima portabilidad de los programas escritos y compilados en Java, pues es únicamente la segunda fase del proceso la que depende de la máquina concreta en la que se ejecuta el intérprete[**?**, **?**].

1.2.1 Definiciones

Traductor

Podemos definir a un traductor como un algoritmo que convierte un programa fuente en un programa objeto equivalente. Hay distintos tipos de traductores, pero enfocaremos nuestra atención en los compiladores.

[16]PASCAL: Lenguaje de Programación que debe su nombre al matemático francés del siglo XVII, Blaise Pascal, quién construyó una de las primeras máquinas sumadoras mecánicas.

Compilador

Un compilador es un traductor, cuyo lenguaje objetivo está en un nivel "inferior" del lenguaje fuente. En el resto de este documento no se distingue entre compilador y traductor.

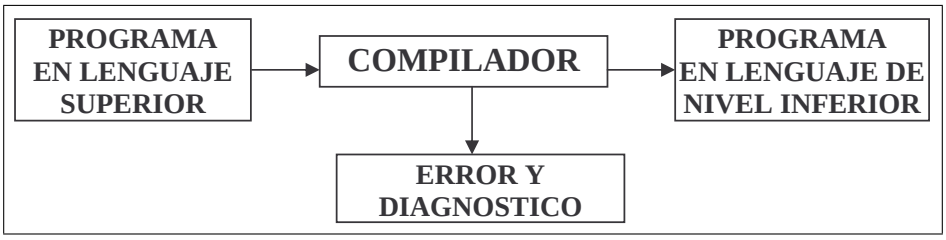

Figura 1.3: Proceso de Compilación

En los casi 60 años de historia de los lenguajes para computadores, han evolucionado numerosas herramientas, que haciendo uso de técnicas básicas (en las que se incluyen: los lenguajes de programación, la arquitectura de computadores, la teoría de lenguajes, los algoritmos, la ingeniería de software, así como también técnicas de programación lineal, estructurada, orientada a objetos, a eventos y/o paralelas), hacen menos difícil la tarea de crear un compilador.

1.2.2 Fases del proceso de traducción

El proceso de traducción se puede resumir en cualquier caso como sugiere la figura **??**. Un código de programa escrito en un lenguaje de programación de alto nivel (nivel superior) es convertido por el compilador en un código equivalente en un lenguaje de menor nivel, típicamente en código de máquina o lenguaje ensamblador que pueden ser usados directamente para controlar un computador. Note que el proceso de compilación produce también reporte sobre errores de sintaxis en el programa de entrada y, en muchos casos, hasta reportes de que tan óptimo es el programa traducido. Cada una de las fases en las que opera un compilador transforma al programa fuente de una representación a otra. La figura **??**, es una versión mas detallada del proceso. Como preparación para esas explicaciones, a continuación se define con más cuidado

el proceso de creación de un compilador y, especialmente, en que consisten el análisis léxico y el gramatical.

1.2.3 Componentes de un traductor

La figura **??** muestra los componentes de un traductor.

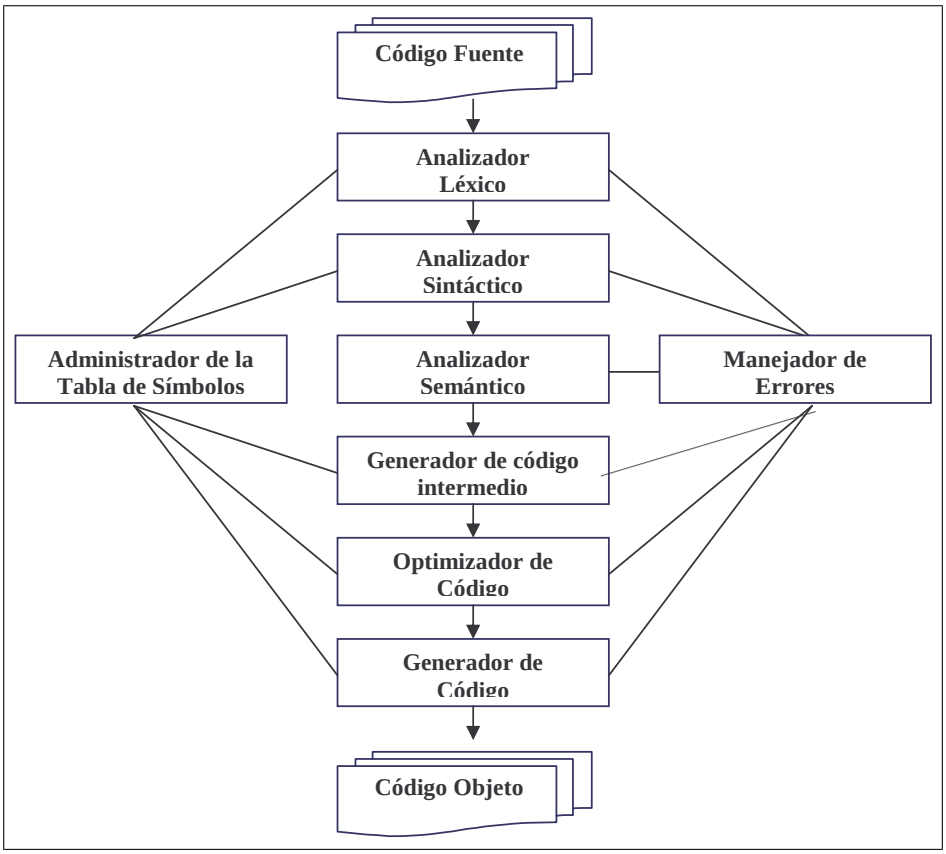

Figura 1.4: Fases generales de compilación

El analizador lexicográfico

El análisis léxico, es la fase de reconocimiento de los elementos del lenguaje, convierte el flujo de caracteres del código fuente, que contiene instrucciones escritas en un lenguaje de programación, en una secuencia de símbolos, útiles para el análisis sintáctico. Las funciones más relevantes de un analizador Léxico, son:

* Leer los caracteres de entrada, de uno en uno y arrojar como salida una secuencia de componentes léxicos (*tokens*).
* Filtrar espacios en blanco, tabuladores, comentarios y caracteres de nueva línea.
* Reconocer identificadores y palabras claves.
* Identificar constantes y numerales.
* Determinar el número de líneas analizadas.
* Generar copias del archivo fuente que agregue notas con mensajes de error.

El problema se basa en la especificación y diseño de programas que ejecuten acciones activadas por patrones dentro de las cadenas (programación dirigida por patrones). Un lenguaje patrón-acción para la especificación de nuestro análisis lineal es el JFLex.

Los patrones se especifican con expresiones regulares y un compilador de JFLex genera un reconocedor de la expresión regular, mediante un autómata finito eficiente.

Expresiones Regulares

Permiten definir un lenguaje para describir el patrón asociado a una marca (*token*). Una regla de representación general de una expresión regular se muestra a continuación:

```
identificador = letra ( letra | digito ) * letra | digito
```

Algunos ejemplos de expresiones regulares, que se muestran a continuaci(ó)n, han sido extraidos de los archivos empleados en la producción de `galateac` del traductor desarrollado.

Palabras claves: Las palabras claves o reservadas van escritas con letra mayúscula e incluidas dentro de dobles comillas `" "`.

```
"DECL"  , "INIT",    ")"
```

Identificadores: Un identificador es una secuencia de uno o m(á)s caracteres cuyo primer caracter es una letra mayúscula o minúscula o un guión de subrayado, seguido por ning(ú)n, uno o varias secuencia de letras, guión de subrayado (ó) d(í)gitos. Dentro de las reglas léxicas, un identificador va incluido dentro de llaves, `{ID}`. Esto es lo que establece la siguiente expresión regular.

```
ID = ([A-Za-z_][A-Za-z_0-9]*)
```

Numerales: Un número cualesquiera se rige por la regla l(é)xica de incluir entre llaves el nombre del identificador definido:

```
Number =
({DecIntegerLiteral}|{DecLongLiteral}|{FloatLiteral}|{DoubleLiteral})
```

Según su tipo, cada una de éstas secciones se desglosan en:

* Para números de tipo entero

  ```
  DecIntegerLiteral = 0 | [1-9][0-9]*
  ```

* Para números de tipo entero largo

  ```
  DecLongLiteral = {DecIntegerLiteral} [lL]
  ```

* Para números de tipo flotante

  ```
  FloatLiteral  = ({FLit1}|{FLit2}|{FLit3}) {Exponent}? [fF]
  ```

* Para números de tipo dobles

  ```
  DoubleLiteral = ({FLit1}|{FLit2}|{FLit3}) {Exponent}?
  ```

Generación de un analizador lexicográfico

Para producir el analizador léxico del traductor se empleó el JFLex, herramienta que admite la descripción léxica del lenguaje como expresiones regulares y produce el código Java del analizador léxico correspondiente. Éste generador, trabaja acoplado

con CUP, es decir, la tabla de símbolos que produce es la misma que el programa correspondiente, generado por CUP y, se emplea para verificar la sintaxis de un lenguaje. El reconocimiento del análisis léxico, puede verse como una transformación en un flujo de caracteres en un conjunto de símbolos bien definido (entrada filtrada). También debe relacionar los mensajes de error del compilador con el programa fuente (Ej.: asociar un número de línea a un mensaje de error). Es decir, se puede identificar la posición de cada símbolo en ese flujo de caracteres y así referir cualquier error con los símbolos o con su ordenamiento, a las posiciones exactas en el flujo de caracteres. En algunos casos se hace una copia del fuente con los errores marcados. Para asociar caracteres tomados de un archivo fuente cualquiera, con símbolos de un lenguaje, se puede escribir tablas en la que una expresión regular queda conectada con una acción que involucra ciertos símbolos del lenguaje. En la entrada a JFLex, esa asociación se puede hacer escribiendo registros con el formato:

```
patron  { acciones }
```

donde los patrones son expresiones regulares y las acciones se describen en código Java que involucra métodos y atributos de clase especiales. Este es un ejemplo concreto:

```
"TITLE"   { yybegin(TITLE);
            return symbol(sym.TITLE); }
```

Cuando se aparea una cadena de caracteres del archivo fuente de entrada con el patrón, las acciones son ejecutadas.

Este registro de asociaciones entre patrones y acciones, para JFLex, debe atender las siguientes indicaciones:

1. Estar estructurado de la siguiente manera:

```
/* -- CODIGO DE PATRON ACCCIONES PARA GALATEA --*/
   import java_cup.runtime.*;
%%
/* -----------DECLARACIONES Y OPCIONES-------- */
%class Lexer
%line
%column
%cup
```

```
    private Symbol symbol(int type) {
        return new Symbol(type, yyline, yycolumn);
    . . .
%state TITLE
    . . .
%%
/* -----------REGLAS LEXICAS------------------- */
<YYINITIAL> {
    . . .
            }
```

2. Devolver cada marca identificada, utilizando la instrucci(ó)n `return symbol`. En nuestro ejemplo para la marca `TITLE`:

```
return symbol(sym.TITLE);
```

3. Cuando se reconoce una marca de tipo identificador que no es una palabra reservada del lenguaje, hay que indicar que es un identificador del programa, usando `yytext()`. Cuando se reconoce la expresión regular correspondiente a un identificador `ID`, se tiene que ejecutar la acción siguiente:

```
return symbol(sym.ID,yytext());
```

   Esto ocurre porque las palabras no reservadas deben ser agregadas a la tabla de símbolos del programa particular que se esté analizando.

4. Al reconocer una marca de tipo numérico, se indica al analizador sintáctico, el valor de dicho número utilizando `yytext()`. Además, hay que indicar el tipo de número. Por ejemplo para el caso de números enteros tenemos:

```
{Number}   {
   return symbol(sym.INTEGER_LITERAL,yytext()); }
```

5. Normalizar los comentarios.

Proceso lexicográfico

Es importante notar que el proceso que realiza el analizador lexicográfico puede verse como el recorrido sobre una serie de estados. Los estados son identificados con etiquetas en el código de entrada a JFLex, siendo el estado inicial marcado con `YYINITIAL`. Ese es el estado en el que se encuentra el traductor al momento de comenzar a revisar el archivo que contiene un programa fuente. A medida que va identificando símbolos del lenguaje, el programa cambia de estado. Las etiquetas de estado sirven para marcar el contexto de trabajo en el que se encuentra el sistema en un momento dado. El objeto es poder actuar de acuerdo a ese contexto. Por ejemplo, suponga que el traductor, que comienza en el estado `YYINITIAL`, está usando la asociación patrón-acción que se mostró en la sección anterior.

```
 "TITLE"    { yybegin(TITLE);
              return symbol(sym.TITLE); }
```

Si el archivo fuente que está revisando contiene la secuencia de caracteres `TITLE` (sin las comillas), el sistema cambiará al estado de "TITLE". Esto se indica con la instrucción `yybegin(TITLE)`. Para definir la conducta del analizador lexicográfico mientras está en el estado "TITLE", el código de entrada a JFLex incluye una unidad como esta:

```
<TITLE> {
 "NETWORK" {System.out.println("\t\n\n TITLE -> NETWORK");
            yybegin(NETWORK);
            return symbol(sym.NETWORK); }
 {EOL}     { /* ignore */ }
 [^]       {  return symbol(sym.ANY,yytext()); }
}
```

la cu(á)l básicamente dice que el analizador permanecerá en el estado `TITLE` ignorando cualquier aparición de los caracteres especiales que definen fines de línea (`EOL`) y mientras lea cualquier secuencia de caracteres diferente a `NETWORK`. Si llega a leer la secuencia `NETWORK` (sin las comillas) abandonará el estado TITLE y siguiendo las instrucciones indicadas pasa al estado NETWORK indicado por la etiqueta `NETWORK` en la instrucci(ó)n `yybegin(NETWORK)`.

En el manual del analista (ver ap(é)ndice A se explica como usar un registro completo, como el que se ha creado en este trabajo para GALATEA, de manera de producir el analizador lexicográfico.

El analizador sintáctico

El análisis sintáctico es el reconocimiento de la estructura del lenguaje. Consiste en agrupar los componentes léxicos del programa fuente en frases gramaticales que el traductor emplea para sintetizar la salida. Para construir el analizador sintáctico de un lenguaje debemos tener una descripción de la gramática del lenguaje. Hay muchas formas de describir reglas gramaticales. En este trabajo se ha usado el formato BNF que puede ser leído por el programa CUP.

Gramáticas

Una gramática, describe de forma natural, la estructura jerárquica de las construcciones de los lenguajes de programación[**?**]. Una gramática normalmente tiene 4 tipos de componentes (se har(á) uso de la notaci(ó)n CUP, para mostrar los ejemplos):

1. Componentes léxicos denominados símbolos terminales.

```
terminal                POINT;
terminal                TITLE;
terminal String         ANY;
```

2. Componentes léxicos no terminales:

```
non terminal    String UnsigFloat;
non terminal    Network;
```

3. Un conjunto de reglas de producción o producciones. Cada regla de producción consiste de un símbolo no terminal llamado lado izquierdo de la producción, separada del lado derecho por ::= y una secuencia de componentes léxicos y no terminales, o ambos, llamado lado derecho de la producción. Por ejemplo:

```
Sign ::= PLUS {:RESULT = "+";:} | MINUS {:RESULT = "-";:};
```

Esta regla de producción dice que el símbolo `Sign` puede ser reducido al símbolo `PLUS` ó al símbolo `MINUS`. En cada caso, esta regla, que es reconocida por CUP, transforma la correspondiente identificación ("+" o "-") en un resultado (`RESULT`) que está disponible en otras reglas y para otros efectos de traducción como se muestra en la secci(ó)n 2 En el capitulo 2 se explican también las reglas de producción CUP que han sido empleadas para crear el compilador GALATEA.

4. La denominación de uno de los símbolos no terminales, como símbolo inicial.

Una gramática de este tipo se puede representar como [G = (T, N, S, P)] Donde:

* T: Conjunto de componente l(é)xico terminal.
* N: Conjunto de componente l(é)xico no terminal.
* P: Conjunto de reglas de producci(ó)n.
* S: Simbolo Inicial, no terminal.

El uso de gramáticas trae consigo numerosas ventajas, algunas citadas a continuación:

* Son especificaciones sintácticas y precisas de lenguajes de programación.
* Un analizador sintáctico, puede ser generado considerando una gramática.
* El proceso de construcción puede llevar a descubrir ambigüedades.
* Una gramática proporciona una estructura a un lenguaje de programación, siendo más fácil generar código y detectar errores.
* Es más fácil ampliar / modificar el lenguaje si está descrito con una gramática.

Existen tres tipos generales de analizadores sintácticos para gramáticas:

Métodos Universales: aquellos que analizan cualquier tipo de gramática (como el algoritmo de Cocke, Younger-Kasami y el de Earley). Son excesivamente ineficientes, por lo que no se usan en la producción de compiladores.

Análisis Descendente: Construyen árboles sintácticos desde la raíz hasta las hojas. Examinan la entrada de izquierda a derecha. Se basan en una subclase de gramáticas: las LL, que permiten analizar una frase de entrada sin que existan bloqueos mutuos. En este tipo de análisis se consideran los analizadores por descenso recursivo, por predicción recursiva y no recursiva.

Análisis Ascendente: Construyen árboles sintácticos de abajo hacia arriba. Examinan la entrada de izquierda a derecha. Se basan en una subclase de gramáticas: LR.

El traductor de código GALATEA a código Java, generado por CUP, basa su recorrido en las gramáticas LALR (del inglés, lookahead-LR, análisis sintáctico LR con símbolo de anticipación), que es una subclase de análisis ascendente. Las principales funciones que realiza un analizador sintáctico, son las siguientes:

* Tomar las marcas del Analizador Léxico y producir como salida una representación del árbol sintáctico que reconoce la entrada de acuerdo a la gramática especificada.

* Verificar que la asignación de tipos sea la correcta y evitar con ello pérdida de información o errores semánticos.

* Generar un código intermedio, ya sea para una máquina virtual o real, que permita la ejecución o interpretación de la entrada.

Como se ha dicho, CUP es una herramienta para generar programas de análisis de LALR de especificaciones simples. Cumple el mismo papel que el analizador sintáctico YACC. EL CUP ha sido trabajado para generar código Java. A continuación se listan

las primeras consideraciones prácticas para el uso de CUP para producir el generador sintáctico de un lenguaje como GALATEA. La estructura de un código que describe la reglas de producción se puede describir así[**?**]:

a) **Definición de paquete y sentencias import:** En esta sección se incluyen las construcciones para indicar que las clases Java generadas a partir de este archivo pertenecen a un determinado paquete o para importar las clases Java necesarias. Para ambas cosas se utiliza exactamente la misma sintaxis que en Java. Esta parte contendrá como mínimo la siguiente línea:

```
import java_cup.runtime.*;
```

b) **Sección de código de usuario:** En esta sección se puede incluir código Java que el usuario desee incorporar en el analizador sintáctico que se va a generar con CUP. También se permite introducir código Java en la clase `parser`, generada por CUP. Y se redefinen los métodos que se invocan como consecuencia de errores de sintaxis. El código siguiente es original de CUP:

```
parser code {:
  public void report_error(String message, Object info) {
   StringBuffer m = new StringBuffer("Error");
   if (info instanceof java_cup.runtime.Symbol) {
    java_cup.runtime.Symbol s = ((java_cup.runtime.Symbol) info);
    if (s.left >= 0) {m.append(" in line "+(s.left+1));
    if (s.right >= 0)m.append(", column "+(s.right+1)); }
    }
   m.append(" : "+message);System.err.println(m);    }
  public void report_fatal_error(String message, Object info) {
   report_error(message, info); System.exit(1); }
    :};
```

c) **Declaración de símbolos terminales y no terminales:** En esta sección se declaran los símbolos *terminales* y *no terminales* de la gramática que define el analizador sintáctico que deseamos producir. Tanto los símbolos *no terminales* como los símbolos *terminales* pueden, opcionalmente, tener asociado un objeto

Java de una cierta clase. Por ejemplo, en el caso de un símbolo *no terminal*, esta clase Java puede representar los subárboles de sintaxis abstracta asociados a ese símbolo no terminal. En el caso de los símbolos *terminales* (*tokens*), el objeto Java representa el dato asociado al *token* (por ejemplo un objeto de la clase `Integer` que represente el valor de una constante, o un `String` que represente el nombre de un identificador). Para declarar símbolos *terminales* y *no terminales* se utiliza la siguiente sintaxis:

```
terminal [<nombre_clase>] nombre1, nombre2, ... ;
non terminal [<nombre_clase>] nombreA, nombreB, ... ;
```

Cada uno de los símbolos gramaticales del estado del analizador sintáctico se representa en CUP por medio de un objeto de la clase Symbol:

```
public class Symbol {
  public int sym;
  public int left, right;
  public Object value;
  public Symbol(int id, int l, int r, Object o){
        ...  }...}
```

El significado de los atributos de un objeto de la clase `Symbol` es el siguiente:

- `sym`: identifica cuál es el símbolo terminal o no terminal representado por el objeto. A cada símbolo terminal y no terminal definidos se les asocia un número diferente que le identifica unívocamente.
- `value`: si el objeto de la clase `Symbol` representa un símbolo terminal o no terminal que tenga asociado un objeto Java, dicho objeto se guarda en el atributo `value`.
- `left` y `right`: aunque su uso no es imprescindible, habitualmente se utilizan para identificar la línea de texto en la que comienza (`left`) y termina (`right`) la parte del programa que se corresponde con ese símbolo terminal o no terminal. Se usan cuando se produce un error de sintaxis para indicar el lugar donde se encuentra el error.

d) **Declaraciones de precedencia:** En CUP, es posible definir niveles de precedencia y la asociatividad de símbolos terminales.

```
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
precedence left IF;
precedence left ELSE;
```

Como se puede ver, las declaraciones de precedencia de un fichero CUP consisten en una secuencia de construcciones que comienzan por la palabra clave `precedence`. A continuación, viene la declaración de asociatividad, que puede tomar los valores `left` y `right`. Finalmente, la construcción termina con una lista de símbolos terminales separados por comas, seguido del símbolo punto y coma. La precedencia de los símbolos terminales viene definida por el orden en que aparecen las construcciones `precedence`. Los terminales que aparecen en la primera construcción `precedence` son los de menor precedencia, a continuación vienen los de la segunda construcción `precedence` y así sucesivamente, hasta llegar a la última, que define a los terminales con mayor precedencia. La asociatividad se aplica cuando no es posible resolver una ambigüedad utilizando reglas de precedencia. La asociatividad de un símbolo terminal puede ser:

(a) `left`: el terminal asocia por la izquierda.

(b) `right`: el terminal asocia por la derecha

e) **Definición del símbolo inicial de la gramática y las reglas de producción:** Para definir el símbolo inicial para el análisis se utiliza la construcción `start with`. Por ejemplo, en este proyecto es así:

```
start with Program;
```

f) **El encabezado** de un archivo de especificación de reglas de producción luce así:

```
/* ------------DECLARACIONES PRELIMINARES-------------*/
```

```
import java_cup.runtime.*;  /*Importacion de clases*/
init with {: scanner.init(); :}; /*Inicializacion*/
scan with {: return scanner.next.token(); :}; /*Invoca*/
parser code {:
    public void report_error(String message, Object info) {
    . . .
:};
/* DECLARACION DE TERMINALES Y NO TERMINALES------ */
terminal        PLUS, MINUS, TIMES, DIVIDE, LPAREN; . . .
non terminal    Id_list,
                Stat_list;
          . . .
/* - PRECEDENCIA Y ASOCIATIVIDAD DE LOS TERMINALES-  */
/* Aqui se especifica que token debe tomarse en cuenta
 primero */
  precedence left PLUS, MINUS;
  precedence left TIMES, DIVIDE;
```

g) En CUP, las acciones están contenidas en la secuencia de código rodeada por los delimitadores de la forma `{: <accion a ejecutar> :}` En general, el sistema registra todos los caracteres dentro de los delimitadores, pero no intenta comprobar que contengan código válido de Java. Por ejemplo:

```
{: System.out.println("\tStatement:\t\t"+stat_str); :}
```

h) La instrucción `{RESULT:}` almacena valores de variables captadas con anterioridad. Ej. para variable capturada `expr_str`

```
IT ASSIGNMENT Expr: expr_str
       {: RESULT = "it("+expr_str+")"; :}
```

i) Para definir todas las reglas de producción que tengan a un mismo símbolo no terminal como antecedente, se escribe el símbolo no terminal en cuestión, seguido de `::=` y a continuación las reglas de producción que le tengan como antecedente, separadas por el símbolo |. Después de la última regla de producción se termina con punto y coma. Si nos fijamos, por ejemplo, en el consecuente de la primera regla de producción, vemos que está formado por una secuencia de símbolos terminales y no terminales

```
Program ::= TITLE Title:t NETWORK Network:red
            DECL Decl:vdecl INIT Init:vinit END
```

algunos de los cuales llevan adyacente un símbolo de dos puntos seguido de un identificador. Recordemos que cuando se presentó la declaración de símbolos terminales y no terminales, se dijo que éstos podían tener asociado un objeto Java. Los identificadores que vienen después de un símbolo terminal o no terminal representan variables Java en las que se guarda el objeto asociado a ese símbolo terminal o no terminal. Estas variables Java pueden ser utilizadas en la parte que viene a continuación. Entre `{: :}` se incluye el código Java que se ejecutará cuando se reduzca la regla de producción en la que se encuentre dicha cláusula. Si el no terminal antecedente tiene asociada una cierta clase Java, obligatoriamente dentro de la cláusula habrá una sentencia `RESULT`. El objeto Java guardado en la variable `RESULT` será el objeto Java que se asocie al no terminal antecedente cuando se reduzca la regla de producción en la que se encuentre esa cláusula.

j) Los elementos de la semántica del lenguaje son incorporados en el analizador sintáctico al asociar instrucciones para producir el código Java equivalente a cada estructura del programa original. El código que se muestra a continuación ilustra esto: Un código Java se usa para identificar la forma apropiada de traducir el cuerpo de un nodo en la red.

El analizador semántico

El analizador semántico revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos para la fase posterior de generación de código. Para esta fase la estructura jerárquica determinada por la fase de análisis sintáctico, es usada para identificar los operadores y operandos de expresiones y proposiciones. Entre las funciones más importantes que realiza un analizador semántico, podemos mencionar:

* Chequear que los tipos de datos están asignados correctamente, para evitar pérdida de información o los errores semánticos.
* Generar un código intermedio, ya sea para una máquina virtual o real, que permita la ejecución o interpretación de la entrada.
* Informar sobre los errores encontrados en la entrada.

El análisis semántico debe garantizar que se consideren todas las reglas dependientes del contexto del lenguaje de programación. Por ejemplo, en Pascal los identificadores deben declararse antes de usarse.

Generación de código intermedio

Una de las ventajas de generar código intermedio es crear independencia y portabilidad entre las fases de producción de software. Una vez concluidas las fases de análisis sintáctico y semántico, algunos compiladores, generan una representación intermedia explícita del programa fuente, cuyas propiedades deben destacarse en la facilidad para producir y traducir al programa fuente. Los árboles sintácticos, la notación postfija y el código de tres direcciones son algunas clases de representaciones intermedias.

Optimizador del código

Con esta fase se pretende lograr que el código objeto se ejecute rápidamente o que a su vez, sea poco pesado, partiendo de modificar al código intermedio, realizando las simplificaciones pertinentes para minimizar la cantidad de instrucciones a ejecutar.

Generación de código ejecutable

La generación del código objeto, es la fase final en la construcción de un compilador, generalmente consiste en generar código de máquina relocalizable o código ensamblador. Las posiciones de memoria se seleccionan para cada una de las variables usadas en el programa y cada una de las instrucciones intermedias se traduce a una secuencia de instrucciones de máquina que ejecuta la misma tarea.

Administrador de la tabla de símbolos

Toda la información del programa fuente, generada a raíz de la construcción de un compilador, como variables, declaraciones de tipos, etiquetas, palabras reservadas entre otras, deben almacenarse en una estructura de datos interna conocida como Tabla de Símbolos. El compilador debe desarrollar una serie de funciones relativas a la manipulación de esta tabla como insertar un nuevo elemento en ella, consultar la información relacionada con un símbolo, borrar un elemento, etc. Como se tiene que acceder mucho a la tabla de símbolos los accesos deben ser lo más rápidos posible para que la compilación sea eficiente. Una tabla de símbolos es una estructura de datos que contiene una entrada por cada identificador, cada una con los atributos: Nombre, Tipo, Ámbito. La mayoría de los atributos son establecidos durante el análisis semántico. Las tablas de símbolos y su uso forman parte del proceso de análisis semántico, para el caso de estudio. En general, la revisión semántica produce una **representación interna** o **código intermedio**, que será enviado al generador de código.

Manejador de errores

En cada fase de compilación completada, se logra determinar una cantidad considerable de errores, permitiendo detectarlos a tiempo y corregirlos para depurar el programa fuente. Las fases lexicográfica y sintáctica, son las que detectan errores donde los caracteres restantes de la entrada no forman ningún componente léxico del lenguaje y aquellos que no cumplen con las reglas estructurales, respectivamente. En el análisis de la semántica, el compilador intenta detectar construcciones que tengan la estructura sintáctica correcta, pero que no tengan significado para la operación implicada, por ejemplo, si se intenta sumar dos identificadores, uno de los cuales es el nombre de una matriz, y el otro, el nombre de un procedimiento.

Cada fase puede encontrar errores. Cuando se encuentra un error, la fase que lo ha detectado tiene de alguna forma que reportarlo, ignorarlo y recuperarlo para poder seguir con el análisis. De esta forma se quiere conseguir que el compilador reporte la

máxima cantidad de errores.

Cada fase detecta un tipo de error:

* Léxico: la cadena de entrada no conforma ningún componente léxico,

* Sintáctico: los componentes léxicos incumplen alguna regla sintáctica,

* Semántico: una construcción no tiene el significado o tipo apropiado.

Para evitar inconvenientes, luego de la detección de errores, se sugiere seguir alguno de los criterios para el manejo de errores:

* Pararse al detectar el primer error.

* Detectar todos los errores de una pasada.

En este capítulo sólo se ha ofrecido una visión general de los conceptos y procesos asociados a la creación de traductores de lenguajes de programación. Se ha hecho un esfuerzo para mostrar como es posible aprovechar herramientas modernas, como el JFLex y el CUP, para la producción de compiladores. En el capítulo siguiente se presentan las especificaciones que se han creado con el fin de producir el primer compilador del lenguaje de simulación GALATEA.

Capítulo 2

Gramáticas del Lenguaje GALATEA

2.1 Especificación de las reglas gramaticales de GALATEA

Gramaticalmente cada una de las partes del lenguaje GALATEA, se definen como sigue.

2.1.1 Program

Un programa en GALATEA tiene la estructura que indica esta regla:

```
Program ::= TITLE Title:t NETWORK Network:red AGENTS Agents:ag
INTERFACE Interface:i DECL Decl:vdecl INIT Init:vinit END.
```

Esta es la regla de más alto nivel que se incluye en el código CUP para producir el traductor con el fin de identificar que un programa GALATEA esta estructurado de la forma que se ha descrito en la secci(ó)n 1.1.1

Las palabras escritas sólo en mayúsculas representan, por convención en esta tesis, a los símbolos *terminales*. Es decir, son palabras *reservadas* del lenguaje de simulación y su especificación final se incluyen en el código que se suministra al analizador lexicográfico JFLex para producir la tabla de símbolos de GALATEA.

La regla establece que en un programa o modelo GALATEA (`Program`), esos símbolos terminales deben aparecer en ciertas posiciones relativas unos a otros. Es decir, por ejemplo, `TITLE` define el primer símbolo del programa y `END.` el último.

Pero la regla también establece la posición relativa de los símbolos *no terminales*, es decir, las adiciones y combinaciones de símbolos terminales con los que se construye un programa particular. Otras reglas definen, a continuación, la estructura interna asociada a esos símbolos no terminales, como los agregados que son.

Se explica aquí, sin embargo, como funciona la referencia a esos símbolos no terminales. Por ejemplo, la secuencia `Title:t` indica que en esa posición, entre `TITLE` y `NETWORK`, debe aparecer un texto correspondiente al título o nombre del modelo. Para saber si un texto es un título se usa la regla que se muestra más adelante. Note, antes de revisar esa regla, como la plataforma CUP nos permite **capturar** ese texto, en la variable `t` que acompaña a `Title`. Con este recurso podemos, entonces, generar la salida apropiada en el formato Java. En particular, `Title:t` conduce a la inclusión de la instrucción:

```
System.out.println("Glider.setTitle(" + t + ");");
```

justo en el punto donde se produce el método *main()* del programa Java equivalente al modelo que se está traduciendo (Ver en la secci(ó)n **??** la descripción del ejemplo y las reglas CUP con esa sección de generación de código que aquí se omite por simplicidad).

2.1.2 Title

Continuando con las reglas de producción de otros símbolos no terminales, un titulo se define así:

```
Title ::=   Title:t ANY:a   {: RESULT = t+" "+a; :}
        | ANY:a {: RESULT = a; :}          ;
```

Es decir, un titulo es cualquier cadena de caracteres (`ANY`) o un titulo seguido de cualquier cadena de caracteres (`Title:t ANY:a`).

Aprovechando la simplicidad de este regla, podemos explicar otro elemento muy importante de las herramientas de traducción que se han usado en este proyecto. CUP permite asociar a cada regla de producción, código de procedimientos, posiblemente incluyendo instrucciones del que hacer con cada agregado de símbolos que se construye.

Ese código de procedimientos se incluye en la regla delimitado por los símbolos : y : como se muestra en esa última regla sobre `Title`. Pero eso no es todo, hay otro servicio muy especial que provee CUP en analizado sintáctico. La variable especial RESULT puede ser usada para "devolverün valor resultado a cualquier otra regla que haya invocado el símbolo no terminal que se define.

Por ejemplo, en la regla que estamos discutiendo sobre `Title`, `RESULT` "recoge"el símbolo tal como fue identificado por la regla, ahora dispuesto como una cadena de caracteres en Java.

2.1.3 Network, Gnode, GnodeHead, GnodeBody

La sección `NETWORK`, está especificada de la siguiente manera:

```
Network ::= Gnode Network | Gnode;

Gnode ::=   GnodeHead GnodeBody;

GnodeHead ::=   GnodeId  IdList | GnodeId ;

GnodeId ::= ID:name LPAREN ID:t RPAREN LBRACK
    INTEGER_LITERAL:i1 INTEGER_LITERAL:i2 RBRACK
        | ID:name LPAREN ID:t RPAREN
        | SimpStat:statement SEMICOLON;

GnodeBody ::= StructStat;
```

Esta sección es más elaborada, porque en ella se incluyen las reglas gramaticales de los distintos tipos de nodos, sus funciones y procedimientos. Al igual que para

todas las demás secciones, los *no terminales* tienen código Java asociado cuya sintaxis va siendo producida por las instrucciones incluidas entre los símbolos `{: :}`.

2.1.4 Decl

La sección `DECL` viene dada por la siguiente especificación:

```
Decl ::=  Decl_List_Item | Decl_List_Item Decl;

Decl_List_Item ::= VAR IdList COLON GType
    | STATISTICS IdList SEMICOLON ;

GType ::= INT | FLOAT ;
```

Como su nombre lo indica, (é)sta secci(ó)n permite estructurar las declaraciones de variables, adem(á)s de permitir determinar si se desea encontrar las estadisticas a todos los nodos o a uno en especifico.

2.1.5 Init

La especificación de INIT es :

```
Init ::= SimpStat:statement SEMICOLON
         {: System.out.println("/** Init */\n\t "+statement+";"); :}
         | SimpStat:statement1 SEMICOLON
         {:System.out.println("\n\t"+statement1+";"); :} Init ;
```

Aprovechamos con esta regla para ilustrar la técnica de traducción. Por cada instrucción identificada en la sección (capturada por `statement`), CUP produce un par de líneas de código Java. La primera con un comentario y, con indentación apropiada, la segunda con la instrucción Java correspondiente a la que fue identificada en el programa original (luego de ser traducida por la regla que define a `SimpStat`, por ejemplo, que no se muestra en este capítulo, pero si en el código final).

2.1.6 Agents

Aunque no se ha implementado en este trabajo, se plantean a continuación las reglas principales que define la sección de agentes de un programa GALATEA.

```
Agents ::= AGENTS AgentList ;

AgentList ::= AgentList GAgent | GAgent ;

GAgent ::=   AGENT GAgentType:t GAgentInternal:b
      {: // Abrir nuevo archivo con nombre t y
            como contenido b :};

GAgentType ::=
    ID:typename COLON COLON {: RESULT = typename; :} ;

GAgentInternal
    ::= GOALS Goals:g BELIEFS Beliefs:kb PREFERENCES
        Preferences:p
    {: RESULT = g + "\n " + kb + "\n " + p; :}
```

La sintaxis detallada de las metas (`Goals`) ha sido definida en [9]. La de la sección de las creencias (`Beliefs`) en [6]. La sección de preferencias del agente tiene todavía pendiente el trabajo de especificación del lenguaje. Pero versiones preliminares de todas las reglas gramaticales, en formato BNF, se pueden encontrar en [5]. Note que esas sintaxis son lenguajes de programación muy diferentes e independientes del GLIDER y que no son procesados por el simulador GALATEA, sino por los motores de inferencia de cada agente.

2.1.7 Interface

La sección de interfaz es todavía más simple que la anterior. Las reglas se muestran a continuación aunque, repetimos, no han sido implementadas en este proyecto de tesis. La razón de esta exclusión es la estrategia de desarrollo que definió el grupo Galatea y que consiste de enfocar todo el esfuerzo en tratar de recuperar los modelos de simulación que han sido codificados en GLIDER, tanto como sea posible.

```
Interface ::= INTERFACE MethodList ;

MethodList ::= MethodList JavaMethod | JavaMethod ;
```

Aquí `JavaMethod` se refiere a la sintaxis de un método en Java directamente. Ahora se puede ver la ventaja de dejar el código de la interfaz en ese lenguaje pues se minimiza el trabajo de traducción. El sistema debe limitarse a colocar esos métodos en un archivo en bytecode *.class* (es decir, después de que el compilador Java los procese) desde donde el simulador pueda incorporarlos en su ejecución.

En este capítulo se ha discutido la sintaxis del lenguaje de simulación GALATEA. En las primeras secciones, se mostró como extiende la sintaxis de GLIDER. La última sección se dedica a presentar versiones simples de las reglas gramaticales que definen la estructura de los programas GALATEA, en una variante de la notación BNF tal como es usada por la herramienta CUP.

Versiones más complejas de esas reglas, pero con los mismos recursos de traducción que se han mostrado en este capítulo, se usan en el capítulo siguiente para traducir un código GALATEA que se usa como ejemplo de referencia: El ejemplo de las taquillas del banco.

Capítulo 3

El traductor y su funcionamiento

En este proyecto (observe con detenimiento a la figura 3.1) la atención se ha concentrado en las fases: análisis léxico y análisis sintáctico. Se ha construido un programa, que llamamos `galateac`, que toma un código en el lenguaje de alto nivel de GALATEA y, luego de realizar los análisis léxico y sintáctico, produce un código Java (con referencias a Clases y objetos predefinidos para el simulador GALATEA).

En otras palabras, `galateac` es un compilador de GALATEA a Java. Para producir este compilador, se ha hecho uso de dos herramientas de software auxiliares. La primera es el JFLex, un programa que, dada la descripción de los símbolos y palabras reservadas de un lenguaje, produce un analizador lexicográfico para ese lenguaje. La segunda herramienta es el CUP, un programa que dada la descripción de las reglas gramaticales de un lenguaje, produce un código que puede identificar e "interpretar" si un programa tiene la sintaxis establecida para ese lenguaje. La explicación del cómo usar JFLex y CUP se presenta en el ap(é)ndice el A anexo a esta tesis. En este capítulo se explican las descripciones lexicográficas del lenguaje GALATEA que se han suministrado como entrada a JFLex para producir el compilador.

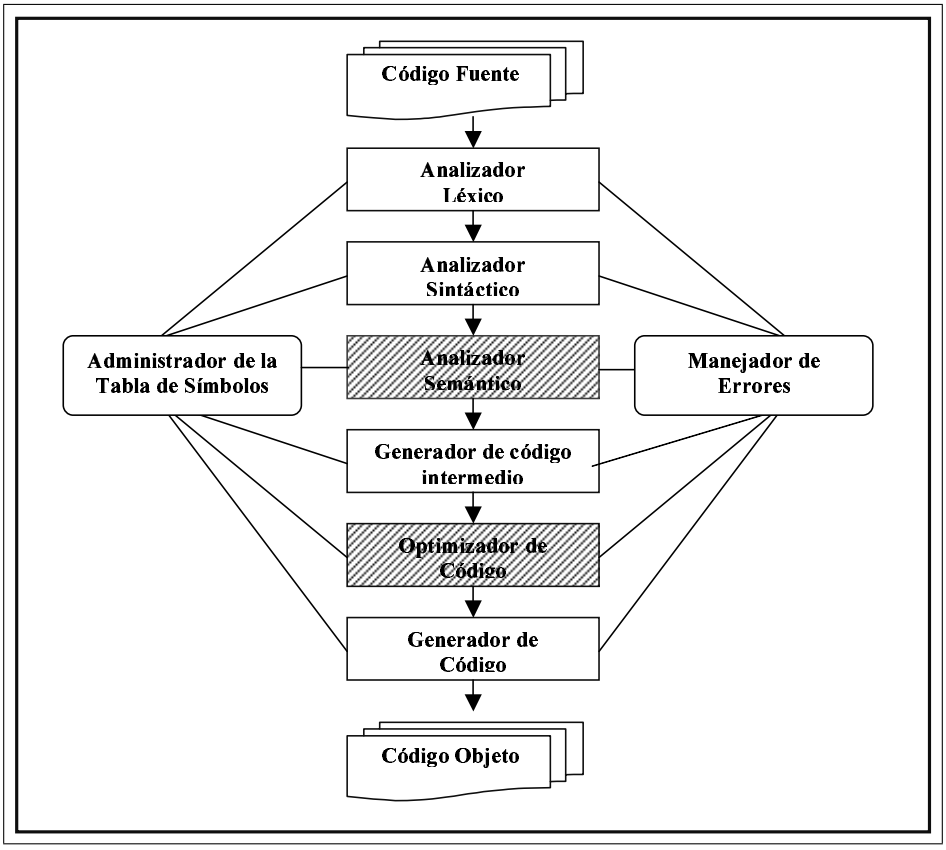

Figura 3.1: Fases del Traductor GALATEA

Para mostrar la funcionalidad actual del compilador GALATEA, en este capítulo se explica como traducir un código de un modelo de simulación en GALATEA: el modelo del sistema de taquillas.

3.1 El sistema simple de las taquillas

En el manual original del GLIDER[10], se define un modelo como ejemplo para el cual se describe el sistema a modelar así:

> Los clientes llegan a una taquilla con tiempos entre llegadas que tienen distribución exponencial con media *TELL* (*Nodo EN*). Son atendidos en

una taquilla (*Nodo TAQ*) con tiempos de atención que tienen una distribución Gaussiana de Media *MAT* y Desviación *DAT*. Luego salen del sistema (*Nodo EX*). Hallar cuantas llegadas encuentran cola ¿ 20. Puede variarse el tiempo medio entre llegadas y el tiempo medio de atenci(ó)n y su desviaci(ó)n.

El código del modelo correspondiente es:

Código 1 Taquilla.gld (Este es el modelo GLIDER, adaptado)

```
/**En este espacio debe aparecer, en comentarios, el codigo glider
*/
TITLE
      Sistema simple de una taquilla
NETWORK
      EN (I) TAQ,EX
       {
        IT:=EXPO(TELL);
        IF LL(EL_TAQ)>20 THEN
          SENDTO(EX)
        ELSE
          SENDTO(TAQ);
       }
      TAQ (R) EX
       {
        STAY:=GAUSS(MAT,DAT);
       }
      EX (E){}
DECL
    VAR TELL, MAT, DAT:FLOAT;
    VAR A:INT ;
    STATISTICS EN,TAQ,EX;
INIT
    ACT(EN,0);
    MAT:=3.80; DAT:=0.8; TELL:=4;
    (*PARAMETROS CUYA MODIFICACION SE CONSULTA EN EL INIT*)
    INTI TELL:8:2:TIEMPO MEDIO ENTRE LLEGADAS;
    INTI MAT :8:2:TIEMPO MEDIO DE ATENCION;
    INTI DAT :8:2:DESVIACION TIEMPO DE ATENCION;
END.
```

3.2 Cómo se produce la traducción de GALATEA a Java

Para explicar la traducción es conveniente recordar como se constituye el compilador `galateac`. Este programa es el producto de integrar el analizador lexicográfico, generado por JFLex e implementado en la clase `Lexer`, con las clases que implementan el analizador sintáctico y que fueron generadas con el CUP, `parser`, el analizador propiamente y `sym`, la tabla de símbolos de GALATEA (observe con detenimiento a la figura **??**). Recuerde que la información acerca de GALATEA necesaria por cada uno de esos programas, fue plasmada en el contenido de los archivos `galatea.flex` y `galatea.cup` respectivamente.

3.2.1 Análisis Léxico del Traductor

Al ejecutar el programa `Lexer`, que produjo el JFLex, sobre un programa particular, se crean los componentes léxicos o *tokens* que servirán de entrada al analizador sintáctico.

1) La actuación del analizador lexicográfico, comienza con dividir el programa fuente escrito en GALATEA en secciones, sin tomar en cuenta los espacios en blanco ni los comentarios. Éstas secciones son identificadas por el analizador con los siguientes nombres:

   (a) `TITLE`

   (b) `NETWORK`

   (c) `DECL`

   (d) `INIT`

   Los comentarios para GALATEA han sido normados de la siguiente manera, para efectos de JFLEX:

```
Comment = {TraditionalComment} | {EndOfLineComment} |
{DocumentationComment} | {PascalComment}

TraditionalComment = "/*" [^*] ~"*/"

EndOfLineComment = "//" ANY?

DocumentationComment = "/**" ~"*/"

PascalComment = "(*" ~"*)"
```

Esta es la forma como se le asigna contexto a las reglas de análisis lexicográfico. La lista completa de las descripciones y etiquetas de estados usadas en este proyecto para GALATEA puede verse en el archivo anexo `galatea.flex`. Las secuencias de cambios de estado en ese caso es así:

```
// desde
<YYINITIAL> {
// pasa a TITLE
"TITLE"   { yybegin(TITLE);
            return symbol(sym.TITLE); }
// o pasa a NETWORK
"NETWORK" { yybegin(NETWORK);}}
// desde
<TITLE> {
// pasa a NETWORK
"NETWORK" { yybegin(NETWORK);
            return symbol(sym.NETWORK); }
        }
// desde
<NETWORK> {
// pasa a DECL
"DECL"   { yybegin(DECL);
           return symbol(sym.DECL); }
// o pasa a INIT
"INIT"   { yybegin(INIT);
           return symbol(sym.INIT); }
// o pasa a END
"END."   { yybegin(END);
           return symbol(sym.END); }
         }
// desde
<DECL>   {
// pasa a VAR
"VAR"    { yybegin(VAR);
           return symbol(sym.VAR); }
// o pasa a STATISTICS
```

```
"STATISTICS" { yybegin(STATISTICS);
               return symbol(sym.STATISTICS); }
// o pasa a INIT
"INIT"   { yybegin(INIT);
           return symbol(sym.INIT); }
         }
// desde <VAR> {
// pasa a INIT
"INIT"  { yybegin(INIT);
          return symbol(sym.INIT); }
// o vuelve a DECL
";"     { yybegin(DECL);
          return symbol(sym.SEMICOLON); }
// los errores terminan el proceso
[^]     { return symbol(sym.ANY,yytext()); }
              }
// desde
<STATISTICS> {
// pasa a INIT
"INIT"  {  yybegin(INIT);
           return symbol(sym.INIT); }
        }
// desde
<INIT> {
// pasa a END
"END."  { yybegin(END); }
// o pasa a INIT
"INTI"  { yybegin(INTI);
          return symbol(sym.INTI); }
        }
// desde
<INTI> {
// pasa a VARDEF
{ID}    { yybegin(VARDEF);
          return symbol(sym.ID,yytext()); }
// o pasa a END
"END."  { yybegin(END);
          return symbol(sym.END); }
// o vuelve a INIT
";"     { yybegin(INIT);
          return symbol(sym.SEMICOLON); }
// o pasa a VARDEF por otra razon
":"     { yybegin(VARDEF);
          return symbol(sym.COLON); }
       }
// desde
<VARDEF> {
// vuelve a INTI
":"     { yybegin(INTI);
          return symbol(sym.COLON); }
{Number}{ yybegin(INTI);
```

```
          return symbol(sym.INTEGER_LITERAL,yytext()); }
// o pasa a INIT
";"     {   yybegin(INIT);
            return symbol(sym.SEMICOLON); }
                }
// desde
<END>{// permanece alli hasta que consigue el fin de archivo
<<EOF>> {   return symbol(sym.EOF);}
     }
```

haciendo uso del código recien mostrado, se generan cuatro variables, las cuales contendrán cada una de las partes del programa.

2) Una vez que el programa se ha dividido, cada una de estas secciones pasa a ser procesada de forma independiente a la otra, según el código asociado. En el caso de `TITLE` se comienza por leer todos los caracteres que se encuentran después de su palabra reservada y los captura para poder mostrar el título del modelo.

   En este caso, cuando al recorrer el archivo fuente se encuentra la marca `TITLE`, se ejecutan las instrucciones Java mostradas entre llaves, que en nuestro ejemplo particular,indican que debe asignar un nuevo simbolo correspondiente al lexema `TITLE`. En la secci(ó)n siguiente se muestran con mayor detenimiento los detalles del analizador sintáctico del traductor.

3.2.2 Análisis Sintáctico del Traductor

Cuando éste último comienza a ejecutarse evalúa todo el código fuente GALATEA, que se le presenta y en la primera pasada divide al programa secciones. Cada sección comienza, para el analizador sint(á)ctico, después del nombre de cierta marca predefinida y termina antes de la marca siguiente. Es decir, en este ejemplo las secciones que se identifican son cinco:

(a) El encabezado

(b) `TITLE`

(c) `NETWORK`

(d) `DECL`

(e) `INIT`

como producto de aplicar la regla:

```
Program ::= TITLE Title:t NETWORK Network:red
            DECL Decl:vdecl INIT Init:vinit END
          {: ... :}
```

tal como fue descrita en el capítulo 1.

Así, en la medida que el analizador sint(á)ctico revisa el programa identificando cada sección de acuerdo esa regla, va ejecutando el código asociado a cada una. De esta forma, el código asociado a la `NETWORK` se ejecuta antes que el código asociado a `Program`.

Esto es muy conveniente para la traducción a Java, pues como se verá a continuación en la medida en que la sección `NETWORK` es revisada es el momento oportuno para generar el código Java de cada uno de los tipos de Nodos GALATEA que conforman en sistema modelado.

Note que, en el ejemplo, la sección `NETWORK`, se divide en tres subsecciones cada una, en la especificación de un nodo (`EN`, `TAQ`, `EX`). Para el caso del nodo `EN`, su identificación comienza con el nombre, luego su tipo seguido por la multiplicidad y sucesores son variables y su respectivo código.

Habiendo hecho eso, el sistema puede invocar esos tipos de nodos para crear instancias u objetos de esos tipos que representen las estructuras particulares del sistema a simular. Esto último es hecho por un programa principal en Java que bien puede generarse en el código asociado a la regla de `Program`.

Cada uno de los nodos componentes son identificados de acuerdo a las reglas siguientes:

```
GnodeId ::= ID:name LPAREN ID:t RPAREN LBRACK
    INTEGER_LITERAL:i1 INTEGER_LITERAL:i2 RBRACK
{: System.out.println("\n /**** Clase"+name+" */ \n");
   System.out.println("\n final class "+name+" extends Node{");
   System.out.println(" private static int mult = 1;");
   System.out.println(" "+name+"(Node s){");
   System.out.println(" super(\""+name+"\",mult,'"+t+"',s);");
   System.out.println(" GBase.nodesl.add(this)};");
   System.out.println(" mult++;\n");
   System.out.println("\n");// } /** fin del nodo "+name+" */ \n");
   RESULT = t+""+name;
:}

| ID:name LPAREN ID:t RPAREN {: System.out.println("\n /**** Clase
"+name+" */ \n\n");
   System.out.println("\n final class "+name+" extends Node{");
   System.out.println("\t "+name+"(Node s){");
   System.out.println("\t\t super(\""+name+"\",'"+t+"',s);");
   System.out.println("\t\t GBase.nodesl.add(this); }\n");
   System.out.println("\n"); //} /** fin del nodo "+name+" */ \n");
   RESULT = t+""+name;
:}

|  SimpStat:statement SEMICOLON
       ;

GnodeBody ::= StructStat:s   {: RESULT = s; :} ;
```

Note que es, en este punto, donde el sistema produce las clases Java que implementan cada uno de los tipos de nodos en el código modelo original. Ese código, sin embargo, no esta completo. La siguiente regla recibe de las anteriores, via la variable especial `RESULT`, la informacion necesaria para, sobre criterios semánticos, decidir como completar el código de cada tipo de nodo.

```
Gnode ::=   GnodeHead:tname
            GnodeBody:b
{:
// extraer primer caracter de tname en t
   String t;
   int i=0, f=1;
   t= tname.subString(i,f);

   RESULT = " public static name tname = New name()";
   if (t.equals("G")||t.equals("L")||t.equals("D")
        ||t.equals("E")||t.equals("R")){
      System.out.println("\n    public boolean fscan() {");
      System.out.println("\n    public void sendto(Message m){ ");
      System.out.println(b+"\n");
      System.out.println("\n    return true; } \n");
```

```
      System.out.println("\n    public boolean fact() {");
      System.out.println("\n    return fscan(); }; \n");

 } else {
      System.out.println("\n    public boolean fact() {");
      System.out.println("\n    public void sendto(Message m){");
      System.out.println(b+"\n");
      System.out.println("\n    return true; } \n");
        };
 :};
```

El valor de `tname`, que fue obtenido de la sección donde se identifica al nodo, es usado para decidir el tipo del nodo y para colocarle un nombre al nodo en cuestión. Todo gracias a un uso apropiado de las String en Java.

Para efecto de comprensión, mostramos como especificamos los terminales y no terminales en el programa `galatea.cup`:

```
% .
terminal    POINT;
% , ; :
terminal    COMMA, SEMICOLON, COLON;
% ( ) { } [ ]
terminal LPAREN, RPAREN, LLLAVE, RLLAVE, LBRACK, RBRACK;
% + - * /
terminal PLUS, MINUS, TIMES, DIVIDE;
% =
terminal EQUAL;
% >
terminal GREAT;
% <

terminal SUB;

terminal String INT, FLOAT;

terminal ASSIGNMENT, ASSIGTYPE;

terminal String ID;

terminal EL, IL;

/* Pascal reserved words */

terminal IF, THEN, ELSE;

terminal VAR;

/* Definiciones Basicas */

terminal String DOTS;

terminal String INTEGER_LITERAL;
```

```
terminal String FLOATING_POINT_LITERAL;

terminal String UnsigInt;

non terminal String UnsigFloat;

non terminal String UnsigNumb;

non terminal String UnsigConst;

/* Galatea Section terminals and non terminals ****/

non terminal Program;

non terminal Network;

/* Title Expression terminal and non terminals ****/

terminal TITLE;

terminal String ANY;

non terminal String Title;

/* Decl Expression terminal and non terminals ****/

terminal DECL;

terminal STATISTICS;

non terminal Decl;

non terminal Decl_List;

non terminal Decl_List_Item;

/* Init Expression terminal and non terminals ****/

terminal INIT, INTI;

non terminal Init;

non terminal GType;

/* End Expression terminal and non terminals ****/

terminal END;

terminal End;

/* Galatea Expression terminals and non terminals ****/ non
terminal String Variable;

non terminal String Entire_var;

non terminal String Var_id;

non terminal String Func_desig;
```

```
non terminal String Func_id;
non terminal String Act_par_list;
non terminal String Expr; non terminal String Simp_expr;
non terminal String Factor;
non terminal String Term;
non terminal String Relat_op;
non terminal String Add_op;
non terminal String Mult_op;
non terminal String Sign;
/* Galatea node terminals and non terminals ****/
terminal NETWORK;
non terminal Gnode, GnodeHead, GnodeId, GnodeBody;
/* Galatea Statament terminals and non terminals ****/
terminal ACT;
terminal ASSEMBLE;
terminal EXPO;
terminal GAUSS;
terminal IT;
terminal SENDTO;
terminal STAY;
terminal TSIM;
terminal LL;
terminal ORDER;
non terminal String Stat;
non terminal String SimpStat;
non terminal String StructStat;
non terminal String CompStat;
non terminal String CondStat;
non terminal String IfStat;
non terminal IncStat;
/* Galatea lists terminals and non terminals ****/
```

```
non terminal IdList, StatPart, List;

non terminal String Gfunction;




/* --------- GRAMATICA ------- */

Program ::= TITLE Title:t NETWORK Network:red DECL Decl:vdecl INIT
Init:vinit END

{:  System.out.print("\n/**** Class GalateaTaq */\n

public class GalateaTaq {");

System.out.println("\n/**** Construccion de la Red: */\n");
         System.out.println(red);
         t.println("\n/**** Declaraciones globales */\n");
         System.out.println(vdecl);
         System.out.println("\n/**** Programa principal */\n");
         System.out.println("\n public static void  main(String args[]){\n");
         System.out.println("\n/**** Inicializacion */\n");
         System.out.println(vinit);
         System.out.println(" Glider.setTitle(\""+t+"\");");
         System.out.println("\n/**** Simulacion */\n");
         System.out.println("\n   GRnd.inisem();");
         System.out.println("\n   Glider.setTsim(...);");
         System.out.println("**** Traza de la simulacion en archivo: **\n");
         System.out.println("\t\t Glider.trace(...);");
         System.out.println("\n/**** Programacion del primer evento: */");
         System.out.println("\n   Glider.act(..,0);");
         System.out.println("\n/**** Procesamiento de la red: */");
         System.out.println("\n   Glider.process();");
         System.out.println("\n/**** Estadisticas de los nodos **/");
         System.out.println("\t\t Glider.stat(...);");
         System.out.println("\n/**** Fin de simulacion */\n");
         System.out.println("\n      }\n");
         System.out.println("\n  } /*** fin de la clase **/");
:} ;



Network ::= Gnode Network | Gnode ; Gnode ::=   GnodeHead:tname
GnodeBody:b

{:  String t;
    int i=0, f=1;
    t= tname.subString(i,f);
    RESULT = " public static name tname = New name()";
    if (t.equals("G")||t.equals("L")||t.equals("D")||t.equals("E")||t.equals("R")){
       System.out.println("\n/***** fscan */\n");
       System.out.println("\n   public boolean fscan() {");
       System.out.println("\n   Message m; /");
       System.out.println(b+"\n");
       System.out.println("\n   return true; } \n");
```

```
        System.out.println("\n/***** fact */\n");
        System.out.println("\n   public boolean fact() {");
        System.out.println("\n   return fscan(); }; \n");
        } else {
        System.out.println("\n/***** fact */\n");
        System.out.println("\n   public boolean fact() {");
        System.out.println("\n   Message m; ");
        System.out.println(b+"\n");
        System.out.println("\n   return true; } \n");
                };
:};

GnodeHead ::= GnodeId:n {: RESULT = n; :} IdList | GnodeId:n
              {:RESULT = n; :};

GnodeId ::= ID:name LPAREN ID:t RPAREN LBRACK INTEGER_LITERAL:i1
INTEGER_LITERAL:i2 RBRACK

{:    System.out.println("\n /****  Clase "+name+" */ \n");
      System.out.println("\n final class "+name+" extends Node{");
      System.out.println(" private static int mult = 1;");
      System.out.println(" "+name+"(Node s){");
      System.out.println(" super(\""+name+"\",mult,'"+t+"',s);");
      System.out.println(" GBase.nodesl.add(this)};");
      System.out.println(" mult++;\n");
      System.out.println("\n");// } /** fin del nodo "+name+" */ \n");
      RESULT = t+""+name;
     :}
             | ID:name LPAREN ID:t RPAREN
{:    System.out.println("\n /**** Clase "+name+" */ \n\n");
      System.out.println("\n final class "+name+" extends Node{");
      System.out.println("\t "+name+"(Node s){");
      System.out.println("\t\t super(\""+name+"\",'"+t+"',s);");
      System.out.println("\t\t GBase.nodesl.add(this); }\n");
      System.out.println("\n"); //} /** fin del nodo "+name+" */ \n");
      RESULT = t+""+name;
:}
              |  SimpStat:statement SEMICOLON;

GnodeBody ::= StructStat:s   {: RESULT = s; :} ;
```

En el manual del analista, anexo a esta tesis, se explica como usar un código con esa estructura para producir el analizador sintáctico. El código empleado para producir el compilador en este trabajo se incluye en archivo `galatea.cup` y se muestra también en el anexo.

La mayoría de las secciones, exceptuando a el encabezado y a `TITLE`, poseen su propio código asociado, por lo que se procura revisar las veces que sea necesario, hasta encontrar y evaluar la mínima expresión.

Así es como usando el compilador GALATEA, `galateac`, se genera los siguientes códigos 2 y 3 en Java:

Código 2 Código generado por el traductor de modelos de simulación GALATEA a código JAVA

```
import galatea.glider.*;

/**
En este espacio debe aparecer,en comentarios, el codigo glider
*/

 /******************** Clase EN */
 class EN extends Node{
     EN(Node s){
         super("EN",'I',s);
         GBase.nodesl.add(this); }
     public void sendto(Message m){
        if (GRnd.expo(EL_TAQ)) {
             sendto(m,Modelo.taq);}
          if (EL_TAQ>20) {
             sendto(m,Modelo.taq);}
          else {
             sendto(m,Modelo.taq);}
             }
         public boolean fact(){
           it(GRnd.expo(TELL));
         return true; }
 }
 /******************** Clase TAQ */
 class TAQ extends Node{
     TAQ(Node s){
         super("TAQ",'R',s);
         GBase.nodesl.add(this); }

     public void sendto(Message m){
     public boolean fact() {
         stay(GRnd.gauss(MAT,DAT));
     return true; }
```

Código 3 continuación código generado por el traductor de modelos de simulación GALATEA a código JAVA

```
 /******************* Clase EX */
 class EX extends Node{
     EX(Node s){
         super("EX",'E',s);
         GBase.nodesl.add(this); }
     public boolean fact() {
       public void sendto(Message m){}
     return true; }

/************************************* Class GalateaTaq */
    public class GalateaTaq {
/************************************* Construccion de la Red: */

/************************************* Declaraciones globales */

        public static float DAT,MAT,TELL;
        public static int A;
        /* ESTADISTICA DE EX,TAQ,EN*/
/************************************* Programa principal */
      public static void  main(String args[]){

          /************************************* Inicializacion */
            GBase.setTitle("Sistema simple de una taquilla");
          /************************************* Simulacion */

    GRnd.inisem();
    GBase.setTsim(300);
    GBase.act(EN,0);
    GBase.process();
    GBase.stat();
   }
} /*** fin de la clase **/
```

3.2.3 Análisis Semántico del Traductor

En este proyecto, las funciones del analizador semántico se han repartido entre varios componentes, combinando algunas con el mismo analizador sintáctico (usando el código Java que se puede asociar a las reglas de producción. Pero las funciones principales el análisis semántico, como la verificación de tipos en expresiones, son delegadas al compilador Java. Note que el compilador GALATEA produce código Java con referencias a Clases predefinidas para GALATEA[1]. De esta forma, la coherencia semántica puede ser verificada por el compilador Java. El traductor GALATEA solo genera código intermedio.

3.2.4 Código Intermedio del Traductor

Éste traductor, como código intermedio, en lenguaje Java, permitirá enlazar y simular los modelos de simulación en GALATEA, usando al Compilador Java.

La fase de optimización de código, no está en consideración en este proyecto.

El programa generado por el compilador GALATEA se ejecutará en Java para obtener y capturar los errores correspondientes. El análisis semántico y el sintáctico, pueden realizarse al mismo tiempo.

En el compilador GALATEA, producido por JFLex y CUP, cada error es reportado, mostrando en pantalla la línea y columna donde se encuentra, y hasta tanto no se corrija, la fase correspondiente no puede ser ejecutada.

En el anexo de este documento, en el A, se explica el proceso de creación del compilador GALATEA a partir del código del léxico para JFLex y de las reglas de producción para CUP.

En este capítulo se ha mostrado un ejercicio puntual de traducción de un modelo particular escrito de acuerdo a la sintaxis GALATEA. El objetivo de la discusión ha sido ilustrar, con referencias a los componentes del traductor que fueron generados a partir de las herramientas de apoyo JFLex y CUP, como se realiza el proceso de traducción.

Se ha mostrado, entonces, que se dispone de un prototipo operativo de un compilador de códigos de modelos de simulación GALATEA a Java para la plataforma de simulación GALATEA, tal como se planteó en los objetivos de esta tesis.

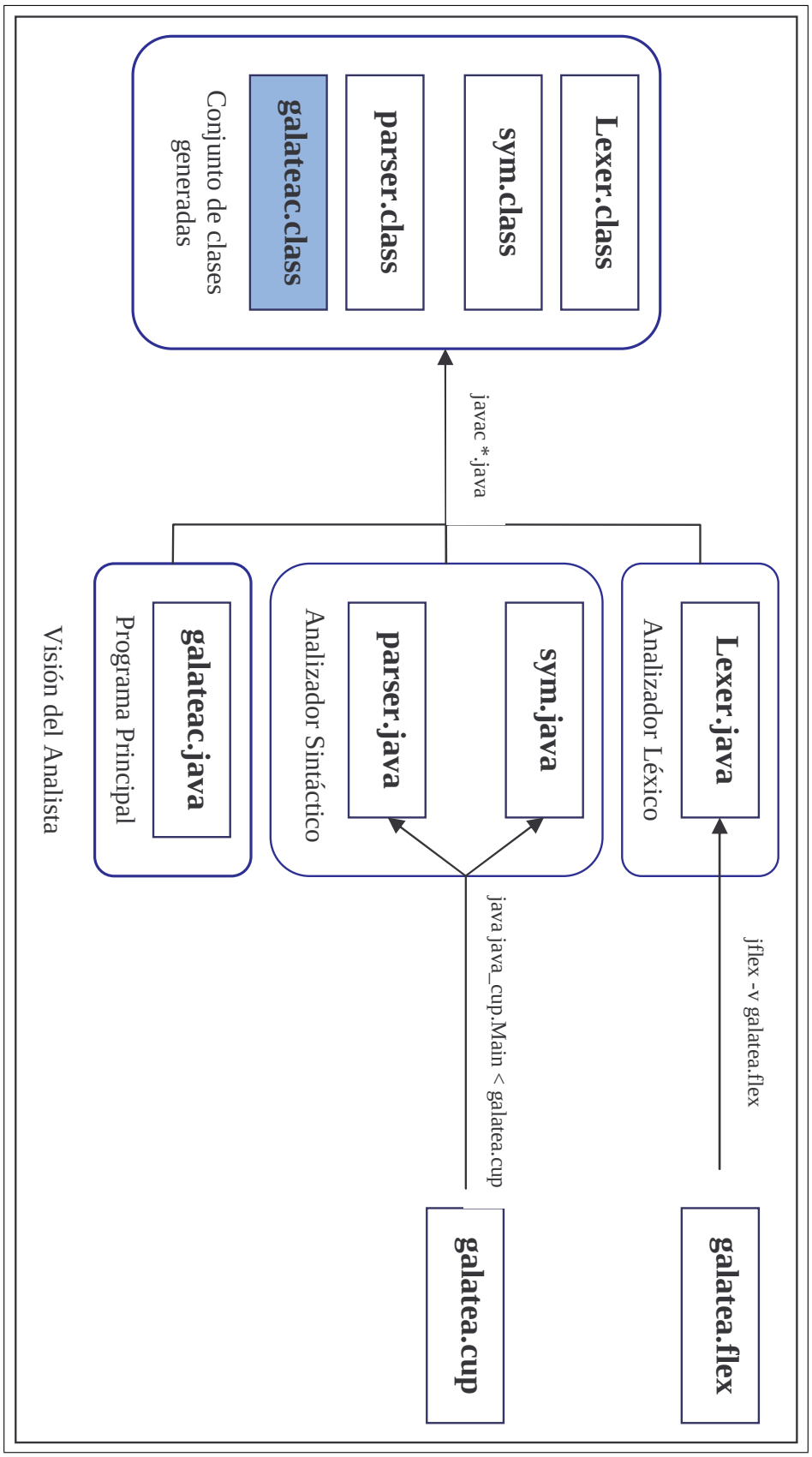

Figura 3.2: Fases del Traductor GALATEA

Conclusiones

El traductor que se describe en este documento y el prototipo desarrollado durante la tesis, son fruto esperado de este proyecto, ya que se ha logrado convertir modelos de simulación codificados en GALATEA a código JAVA.

Hemos formalizado la sintaxis y estructura lexicográfica que comparten GLIDER y GALATEA y hemos usado esa formalización para construir los archivos base del traductor (`galatea.cup` y `galatea.flex`)

Éste traductor es un componente esencial de la plataforma planeada para GALATEA, con la cual se espera poder atender las necesidades de los modelistas, con un lenguaje de alto nivel y al mismo tiempo, ofrecer las ventajas de una plataforma profesional para desarrollo de software JAVA.

El prototipo actual se encarga de las secciones `TITLE`, `NETWORK`, `DECL` e `INIT`, por lo que se propone, para trabajos futuros, involucrar las secciones `AGENTS` e `INTERFACE` y así completar la estructura del programa GALATEA. Otro punto que puede considerarse, como extensión a este trabajo, es la incorporación de los llamados embellecedores de código, para obtener una salida de fácil lectura.

Con la creación de este prototipo se ha conectado a la Plataforma de Simulación GALATEA con el Módulo GUI[1]. Una de las limitaciones en este tipo de trabajo, son la falta de documentación bibliográfica para el uso de herramientas nuevas y la poca información existente se encuentra en un lenguaje distinto al español.

[1] GUI:Interfaz Gráfica de Usuario

Referencias

[1] M. Y. Uzcátegui, "Diseño de la plataforma de simulación de sistemas multi-agentes galatea," Master's thesis, Maestría en Computación, Universidad de Los Andes. Mérida. Venezuela., 2002.

[2] C. Domingo, G. Tonella, and M. Sananes, *GLIDER Reference Manual*. CESIMO–IEAC.Universidad de Los Andes, Mérida, Venezuela, 1 ed., August 1996. CESIMO IT-9608.

[3] Sun Microsystems Inc, "The source for the java technology," 1994.

[4] J. A. Dávila and M. Uzcátegui, "Galatea: A multi-agent simulation platform," in *International Conference on Modelling, Simulation and Neural Networks [MSNN-2000]*, (Mérida, Venezuela), pp. 217–233, AMSE & ULA, October, 22-24 2000.

[5] J. A. Dávila, *Agents in Logic Programming*. PhD thesis, Imperial College of Science, Technology and Medicine., London, UK, June 1997.

[6] J. A. Dávila, "Openlog: A logic programming language based on abduction," in *PPDP'99*, Lecture Notes in Computer Science. 1702, (Paris, France), Springer, 1999.

[7] M. Cabral, "Prototipo del módulo GUI para la plataforma de simulación galatea," 2001.

[8] L. Díaz, "Integración de sistemas de información geográfica a la plataforma de simulación galatea," September 2002. Tutor: Tucci, Kay.

[9] J. A. Dávila, "Actilog: An agent activation language," *Proceedings of PADL2003. New Orleans, USA*, vol. LNCS, 2003.

[10] G. D. Group, "GLIDER simulation language," tech. rep., Interdisciplinary Research Center. Department of Mathematical Sciences. College of Sciences, San Diego State University. California. USA, August 1996.

[11] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modelling and Simulation*. Academic Press, second ed., 2000.

[12] K. A. Tucci, "Prototipo del compilador GLIDER en C++," 1993.

[13] G. Tonella, C. Domingo, M. Sananes, and K. Tucci, "El lenguaje GLIDER y la computación orientada hacia objeto," in *XLIII Convención Anual ASOVAC*, (Mérida, Venezuela), Acta Científica Venezolana, November, 14-19 1993.

[14] O. Terán, "Simulación de cambios estructurales y análisis de escenarios," Master's thesis, Maestría en Estadística Aplicada, Universidad de Los Andes. Mérida. Venezuela., 1994.

[15] F. J. Palm, "Simulación combinada discreta/continua orientada a objetos: Diseño para un lenguaje GLIDER orientado a objetos," Master's thesis, Maestría en Matemática Aplicada a la Ingeniería, Universidad de Los Andes. Mérida. Venezuela., 1999.

[16] C. Domingo, "GLIDER, a network oriented simulation language for continuous and discrete event simulation," in *International Conference on Mathematical Models*, (Madras, India), August, 11-14 1988.

[17] C. Domingo, "Proyecto GLIDER. e-122-92. informe 1992-1995," tech. rep., CDCHT, Universidad de Los Andes. Mérida. Venezuela, December 1995.

[18] C. Domingo and M. Hernández, "Ideas básicas del lenguaje GLIDER," tech. rep., Instituto de Estadística Aplicada en Computación, Universidad de Los Andes. Mérida. Venezuela, October 1985.

[19] C. Domingo and G. Tonella, "Towards a theory of structural modeling," vol. 10(3), pp. 1–18, Elsevier, 2000.

[20] G. Booch and J. Rumbaugh, *Unified Method for Object Oriented Development.* Rational Software Corp, 1996.

[21] G. Booch, *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, Inc, second edition ed., 1996.

Apéndice A

Manual del Analista

A.1 Para reconstruir el compilador GALATEA

Este manual contiene las instrucciones para reconstruir el compilador GALATEA, `galateac`, usando las herramientas de producción del analizador lexicográfico, JFLex, y la herramienta de producción del analizador sintáctico, CUP. En las últimas secciones, incluimos también las instrucciones mínimas para instalar todos los programas y configurar los archivos requeridos en ese proceso.

Los pasos que debe seguir el traductor de modelos de simulación GALATEA a código Java, antes de convertirse en producto final, tienen continuidad y dependen entre sí. Estos pasos se explican a continuación:

1. Una vez que se ha introducido el código que manejará este analizador, en el archivo `galatea.flex`, éste es pasado por la herramienta JFLex, con el uso de la instrucción

```
jflex -v galatea.flex
```

para que éste produzca el archivo fuente del analizador léxico en java, llamado `Lexer.java` en este proyecto (el nombre es configurable). Observe la Figura A.1.

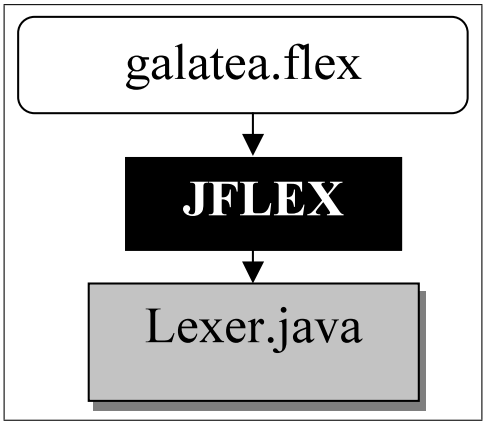

Figura A.1: Proceso para obtener el fuente del Analizador Léxico

`Lexer.java` contiene las declaraciones de cada *estado léxico* o *sección* con su respectiva asignación numérica, en valores constantes, según sea su ubicación dentro del código `galatea.flex`.

2. Seguidamente se construye al analizador sintáctico. En el archivo `galatea.cup`, se especifica la gramática que reconocerá el traductor y luego se ejecuta CUP, usando la instrucción java

```
java_cup.Main  < galatea.cup
```

generando así los archivos fuentes del analizador sintáctico en lenguaje java, designados `sym` y `parser` (Ver Figura A.2).

3. Al completarse las fases de análisis léxico y sintácticos se procede a ejecutar la instrucción:

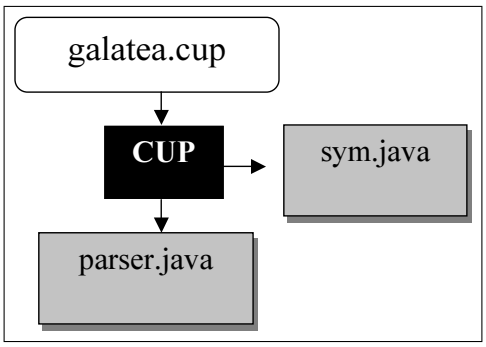

Figura A.2: Proceso para obtener los fuentes del Analizador Sintáctico

```
javac *.java
```

la cuál procesa todos los archivos java generados con JFLEX, CUP y también al archivo principal `galateac.java`. Este archivo contiene un pequeño programa Java que hemos preparado para que sirva como integrador de los analizadores lexicográfico y sintácticos, definidos en las clases `Lexer`, `sym` y `parser`. En este punto se debe usar el compilador Java para producir todos los archivos bytecode .class que definen al traductor. La figura A.3 resume todo el proceso y las relaciones entre los archivos.

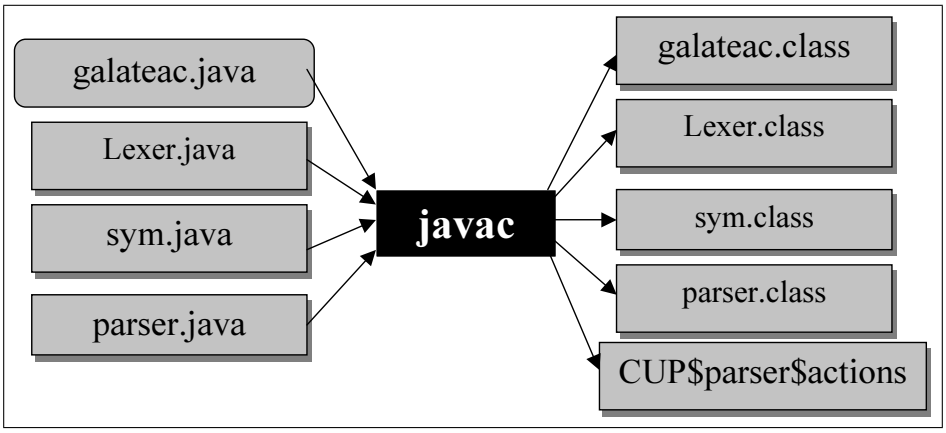

Figura A.3: Fase de generación de los archivos .class

4. En este punto, el analista dispondrá del nuevo programa `galateac` con el cual podrá traducir los archivos .gld . En el manual del usuario se describe el proceso

tal como ha sido establecido en este proyecto.

Es importante notar que debido a que algunas decisiones de diseño del compilador aún se discuten en el grupo de desarrollo de GALATEA (como, por ejemplo, la estrategia para la representación de la listas de nodos sucesores) el programa `galateac` aún no está completamente parametrízado para traducir cualquier modelo GALATEA completamente. El compilador `galateac` actual producirá una salida que **debe ser revisada** por un programador Java y adecuada manualmente para sus necesidades particulares. En el capitulo 3 de la tesis se muestran algunas de esas adecuaciones en relación con el modelo que fue usado como referencia en el desarrollo de este prototipo.

En este apartado se muestra como utilizar cada una de las herramientas, desde su instalación, pasando por la estructura de cada programa, hasta las instrucciones para hacer modificaciones y lograr con éxito la traducción.

Hoy día, existen dos herramientas que están a la disposición para la construcción de compiladores y que son útiles tanto bajo ambiente Unix como en Windows, tal es el caso de JFLex y CUP. Estas herramientas ejecutan actividades diferentes, la primera se concentra en la creación del analizador léxico y la segunda se ocupa de producir al analizador sintáctico. Ambas han sido creadas para generar sus productos en código Java.

A.1.1 JFLex

JFLex, es descendiente de JLex, quién fue desarrollado por Elliot Berk, pero a pesar de ello no tienen código en común. Gerwin Klein, ha sido quién ha documentado el desarrollo de JFLex.

JFLex, actualmente cuenta con dos versiones, la 1.3.5 lanzada en el 2001 y la 1.4(2003). Por ser la 1.3.5 más estable y por estar bien documentada, se recomienda para la generación del analizador léxico.

La entrada a JFLex, debe especificar a un sistema de expresiones regulares y sus respectivas acciones a seguir. La salida es un archivo fuente Java, que dentro de nuestra especificación se denominará `Lexer`, el cual incluye: un constructor que se

llama con un argumento y un método que retorna el símbolo siguiente de la entrada. JFLex debe funcionar en cualquier plataforma que se apoye en Java JRE/JDK 1,1 o superior.

InstalaciÓn de JFLex

Visitar el web site http://www.jflex.de. Se recomienda una vez all(í) analizar las versiones existentes, seleccionar la versión que desea instalar y verificar la compatibilidad con el sistema operativo de su computado.

Ambiente Windows

La instalación de JFLex en Ambiente Windows 95/98/NT, conlleva los siguientes pasos:

1. Descomprimir el archivo descargado de la página principal de JFLex y crearle un directorio, si por ejemplo, desea que sea en el directorio `C:\`, al descomprimir el archivo se crea el directorio JFlex que est(á) estructurado de la siguiente manera:

```
C:\JFlex \
      + -- bin\       (start scriptts)
      + -- doc\
      + -- examples
                +-- binary\
                +-- byaccj\
                +-- cup\
                +-- interpreter\
                +-- java\
                +-- simple\
                    +-- standalone
        + -- lib\
        + -- srcs\
                +-- Jflex\
                +-- Jflex\gui
                +-- java_cup\rutine
```

2. Modificar el archivo `bin\jflex.bat` para que la variable `JAVA_HOME` contenga el directorio donde reside el JDK de Java (por ejemplo `C:\java`) y la variable `JFLEX_HOME` debe contener el directorio que contiene JFlex (en el ejemplo:

`C:\JFlex`)

3. Incluir el directorio de JFlex en su trayectoria (la que contiene la escritura del comienzo, modificando el path)

Ambiente Unix

Para instalar JFlex en un sistema de Unix, es necesario obtener el archivo tar (`set PATH = %PATH%; C:\JFlex`) y seguir los pasos que se muestran a continuación:

1. Descargar el archivo dentro de un directorio que tenga permiso de escritura como por ejemplo a `/usr/share`: tar - C/usr/share - xvzf jflex-1.3.5.tar.gz.

2. Especificar el enlace dentro de su trayectoria binaria a `bin/jflex`, `ln - s/usr/share/JFlex/`

3. Si el intérprete de Java no está en su trayectoria, usted necesita proveer su localización.

A.1.2 Especificación de JFLex

Para crear el programa en JFLex, comience por conocer su estructura. JFLex, se divide en tres secciones, separadas debidamente con el uso de dos signos de porcentaje (%%):

1. **User code:** o código de usuario, en esta sección se especifican los paquetes de librerías (`import java_cup.runtime.*;`), que serán utilizadas con sus respectivos comentarios.

2. **JFlex directives:** Directiva de JFLex, esta secci(ó)n contiene las definiciones macro, aquellas que cambian el comportamiento del Programa. Ejemplo:

`%class Lexer`	indica la clase creada al ejecutar JFlex
`%line`	el número de la línea actual `(yyline)`
`%column`	señala que para obtener usa la variable `yycolumn`
`%cup`	crea compatibilidad con el programa CUP

3. **Regular expression rules:** ésta sección permite declarar las reglas por las cuales se regirá el analizador. Cada regla, incluye una lista de estados, una expresión regular y una acción asociada.

A.1.3 Como compilar código en jflex y generar código en Java o cualquier otro lenguaje

Cuando se escribe un programa en JFLex, lo que hay que hacer es compilarlo, utilizando para ello el comando:

```
jflex <nombre_archivo.flex>
```

que da como resultado a una clase java.

Para mayor información de la herramienta JFLex, consulte su página principal `http://wwww.jflex.de`.

JFLex es diseñado para trabajar junto con el generador del análisis con estrategia LALR, CUP.

A.1.4 CUP

El analizador sintáctico CUP, fue desarrollado por S. Hudson, F. Flannery y C. Scott Ananian. Desde mediados de 1999, permanece vigente la versión 0.10. INSTALACIÓN DEL CUP El software de CUP puede ser descargado, por ejemplo, desde:

`http://www.cs.princeton.edu/#appel/modern/java/`

en un archivo `.tar` o `.zip`. Al estar implementado en Java completamente, los archivos puede ser instalados en cualquier sistema operativo que implemente la maquina virtual de Java.

La instalación de CUP se reduce a colocar los archivos contenidos en ese archivo comprimido en el directorio `bin` del árbol que se describió en la sección anterior para el JFLex.

A.1.5 Forma de la especificación CUP

Un archivo de entrada para CUP consta de cinco partes, que se explican a continuación en forma resumida. Para mayores detalles y ejemplos, se recomienda consultar a 1 de este documento de tesis.

1. **Definición de paquete y sentencias import:** En esta sección se incluyen las construcciones para indicar que las clases Java generadas a partir de este archivo pertenecen a un determinado paquete o importar las clases Java necesarias. Para ambas cosas se utiliza exactamente la misma sintaxis que en Java. Esta parte contendrá como mínimo la siguiente línea:

   ```
   import java_cup.runtime.*;
   ```

2. **Sección de código de usuario:** En esta sección se puede incluir código Java que el usuario desee incorporar en el analizador sintáctico que se va a obtener con CUP. También se permite introducir código Java en la clase `parser`, generada por CUP. Y se redefinen los métodos que se invocan como consecuencia de errores de sintaxis.

3. **Declaración de símbolos terminales y no terminales:** En esta sección se declaran los símbolos terminales y no terminales de la gramática que define el analizador sintáctico que deseamos producir. Tanto los símbolos no terminales como los símbolos terminales pueden, opcionalmente, tener asociado un objeto Java de una cierta clase. Por ejemplo, en el caso de un símbolo no terminal, esta clase Java puede representar los subárboles de sintaxis abstracta asociados a ese símbolo no terminal. En el caso de los símbolos terminales (o *tokens*), el objeto Java representa el dato asociado al *token* (por ejemplo un objeto de la clase `Integer` que represente el valor de una constante, o un `String` que represente el nombre de un identificador).

   Cada uno de los símbolos gramaticales del estado del analizador sintáctico se representa en CUP por medio de un objeto de la clase Symbol.

4. **Declaraciones de precedencia:** En CUP, es posible definir niveles de precedencia y la asociatividad de símbolos terminales. Las declaraciones de precedencia de un fichero CUP consisten en una secuencia de construcciones que comienzan por la palabra clave precedence. A continuación, viene la declaración de asociatividad, que puede tomar los valores `left` y `right`. Finalmente, la construcción termina con una lista de símbolos terminales separados por comas, seguido del símbolo punto y coma. La precedencia de los símbolos terminales viene definida por el orden en que aparecen las construcciones precedence. Los terminales que aparecen en la primera construcción precedence son los de menor precedencia, a continuación vienen los de la segunda construcción precedence y así sucesivamente, hasta llegar a la última, que define a los terminales con mayor precedencia. La asociatividad se aplica cuando no es posible resolver una ambigüedad utilizando reglas de precedencia.

5. **Definición del símbolo inicial de la gramática y las reglas de producción:** Para definir el símbolo inicial de la gramática se utiliza la construcción:

   ```
   start with Program;
   ```

   Para definir todas las reglas de producción que tengan a un mismo símbolo no terminal como antecedente, se escribe el símbolo no terminal en cuestión (en el ejemplo, `Program`), seguido de ::= y a continuación las reglas de producción que le tengan como antecedente, separadas por el símbolo `$\mid$`. Después de la última regla de producción se termina con punto y coma.

A.1.6 Reglas Generales para la generación de código CUP

1. Todos los miembros pueden tener el modificador de acceso por omisi(ó)n, por lo que puede evitarse la especificación de modificadores de acceso *public, protected, private.*

2. Las clases en Java que se deseen interpretar tienen que tener un método main, cuyo prototipo es *public static void main(String args[])*.

3. Una buena estrategia a seguir en la generación de código es generar las variables del programa como variables miembro, y los procedimientos y funciones como métodos.

A.1.7 Como compilar código en CUP y generar código en Java o cualquier otro lenguaje

Para producir un programa de análisis (*parser*) de esta especificación, se usa el generador CUP. Si las definiciones y especificaciones han sido almacenadas en *galatea.cup*, entonces se invoca a CUP (en el sistema Unix o Windown) usando el siguiente comando:

```
c:\> java java_cup.Main  < galatea.cup
```

Esta acción produce dos archivos de fuente java, que contienen partes del programa de análisis generado:

* `sym.java`: el cuál contiene las declaraciones de cada símbolo terminal, referencia a símbolos.

* `parser.java`: a partir del programa dado construye un programa de análisis completo.

Visite el web site http://www.cs.princeton.edu/ appel/modern/java/CUP/

para obtener más detalles acerca de instalación, versiones y manual de usuario entre otros.

A.1.8 Para reportar errores al compilar

Cuando se produce un error en el análisis sintáctico, CUP invoca a los siguientes métodos:

```
public void syntax_error(Symbol s);
public void unrecovered_syntax_error(Symbol s) throws java.lang.Exception;
```

El código java generado por el traductor sigue a otro nivel donde se ejecuta con JAVA para generar su respectiva clase y esta a su vez es procesada con el conjunto de clases de la Plataforma GALATEA para obtener el archivo *ModeloGlider.java* con las clases GALATEA que implementa el modelo correspondiente.

Apéndice B

Manual del Usuario

Este manual es una descripción muy sencilla del como usar el compilador `galateac` para procesar un archivo .gld y producir el esqueleto básico del programa Java equivalente sobre la plataforma GALATEA.

Algunas explicaciones y aclaratorias son necesarias en este punto.

1. El programa `galateac` es un programa Java (es decir, esta contenido en uno o mas archivos bytecode .class) y por tanto requiere la máquina virtual de Java (el programa java) para poder ejecutarse.

2. El programa `galateac` traduce archivos GALATEA (es decir con la sintaxis de GLIDER basada en PASCAL y algunas modificaciones provistas por la plataforma GALATEA) en un esqueleto Java que invoca los paquetes que implementan la semántica GALATEA que puede ser procesada por el simulador GALATEA.

3. Ese esqueleto NO es un programa Java completo, pues algunas decisiones de diseño del compilador aún son discutidas en el grupo de desarrollo de GALATEA.

Por lo tanto, el archivo Java generado por `galateac` debe ser revisado por un programador Java, que entienda además el API de Galatea. Una vez hecha esa revision y los ajustes apropiados, ese esqueleto Java se convertirá en un programa completo que podrá ser compilado con el compilador `javac` y ejecutado con las clases del simulador GALATEA.

B.1 Para traducir un modelo GALATEA a Java

Con las consideraciones anteriores presentes, podemos describir que la traducción de GALATEA a Java, usando `galateac`, es tan simple como invocar la maquina virtual Java con esta instrucción al sistema operativo:

```
java galateac  Modelo.gld  >  Modelo.java .
```

donde Modelo.gld es el programa GALATEA que contiene el modelo escrito en GALATEA y Modelo.java el c(ó)digo java gennerado por el traductor. La figura B.1 ilustra el proceso general de compilación:

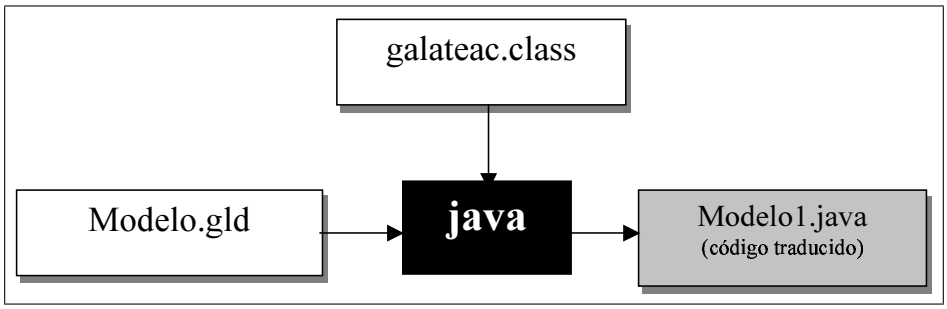

Figura B.1: Fase Final del Traductor

B.2 Para generar el ejecutable del modelo

1. Instale los archivos de la plataforma GALATEA. Para ellos coloque los archivos del paquete `galatea` (al menos el `galatea.glider`), en un directorio cuya raíz

aparezca registrada en las variables de ambiente `PATH` y `CLASSPATH`, de su entorno operativo.

2. En ese entorno operativo, compile con el `javac`, el archivo Java del modelo (javac Modelo.java), que fue producido por `galateac`, obteni(é)ndose el c(ó)digo `Modelo.class`.

B.3 Para simular con ese modelo

Simplemente ejecute el programa creado en el paso anterior, `Modelo.class`, aprovechando el mismo entorno operativo (es decir, la misma configuración del `CLASSPATH` y `PATH`). Por ejemplo, la instrucción en el caso que hemos discutido seria:

```
java Modelo
```

si se ejecuta en el mismo directorio donde se almacena el archivo `Modelo.class`.

La (ú)ltima figura ilustra los últimos dos procesos de generar el ejecutable con el `javac` y realizar simulaciones con `java`.

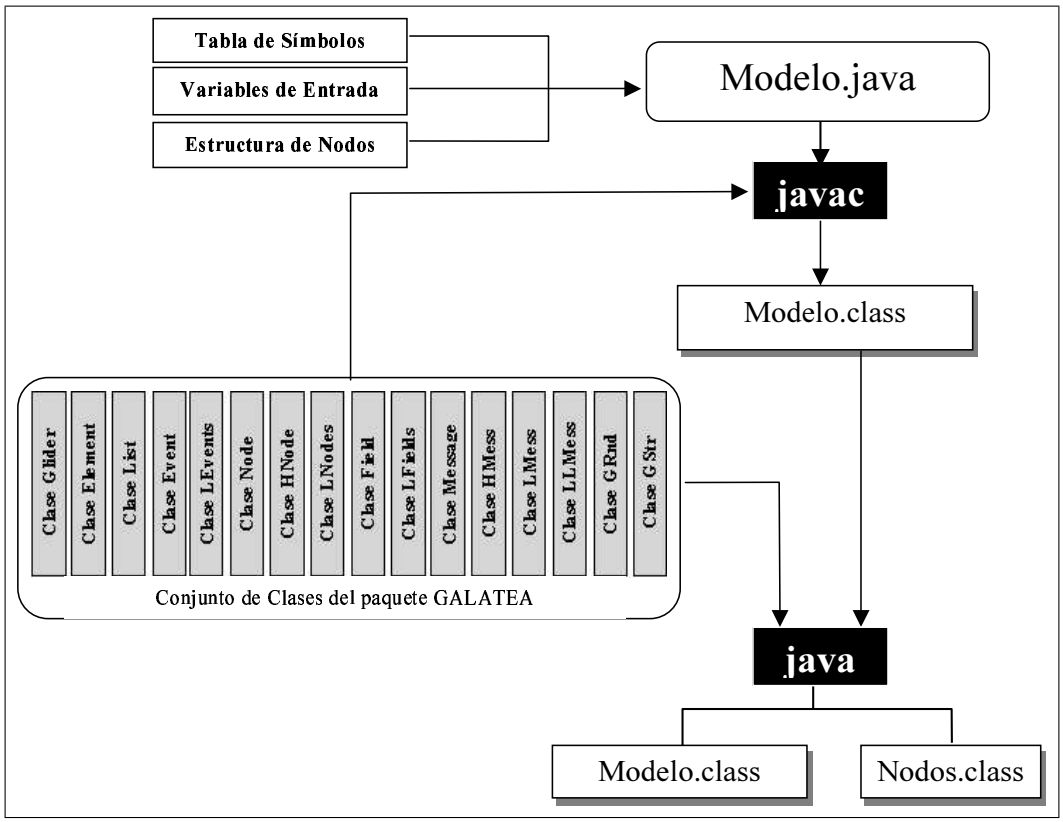

Figura B.2: Fase para generar los ejecutables

Glosario

ALGOL: ALGOritmic Language.

Árbol Sintáctico: representación gráfica de la derivación de las reglas gramaticales que genera una entrada para un analizador sintáctico.

Archivos JAR: archivo de Java. Formato de archivo utilizado para agregar muchos archivos a uno.

Argumento: elemento de datos especificado en una llamada de método. Un argumento puede ser un valor literal, una variable o una expresión.

Autómata: diagrama generado al compilar una Expresión Regular, usando un reconocedor de lenguaje. También se usa para designar a la máquina abstracta.

CLASSPATH: es una variable de entorno que indica a la máquina virtual Java y a otras aplicaciones Java (por ejemplo, las herramientas Java ubicadas en el directorio JDK1.1.X) dónde encontrar las bibliotecas de clase, incluidas las bibliotecas de clase definidas por el usuario.

Código de Tres Direcciones: secuencias de proposiciones de la forma

`x:=` y `operador z`

Código Intermedio: es una representación cercana al lenguaje de bajo nivel, que representa la entrada en función de un conjunto reducido de instrucciones, que

pueden ser ejecutados por una máquina virtual. Si se genera el código de una máquina real, entonces el conjunto de instrucciones corresponde con el código de esa máquina real.

Compilador: Programa que lee las l(í)neas escritas en un Lenguaje de Programaci(ó)n de alto nivel y las convierte a otro de nivel inferior.

Componente Léxico: *token* o símbolo, es cualquier cadena de longitud finita, que tiene un significado especial dentro de un lenguaje.

CUP: Constructor of Useful Parsers.

Display: Matriz d de apuntadores a registros de actuación.

FORTRAN: FORmulae TRANslator.

Gramática: Conjunto de reglas de producción que permiten definir todas las cadenas válidas que son aceptadas por un analizador sintáctico.

Heap: Montículo

JDK: Kit de desarrollo Java(TM). Entorno de desarrollo de software para escribir applets y aplicaciones en Java.

JFLex: Java Format Lex.

JVM: Máquina virtual Java. Parte del entorno runtime de Java que es responsable de interpretar los bytecodes de Java.

LALR: Analizador sintáctico que significa LA: anticipar la acción, LR

LEX: Lenguaje de patrón - acción, para especificar los analizadores léxicos.

Lexema: Secuencia de caracteres que forman un componente léxico. Cualquier cadena que concuerde con un patrón.

Léxico: Relacionado con cómo se traducen los caracteres del código de origen a símbolos que puede comprender el compilador.

LISP: es un lenguaje de programación funcional desarrollado en los EEUU y está orientado al procesamiento de listas y términos funcionales. Fue el primer representante exitoso de la familia de lenguajes de programación declarativos que luego incluyó a los lenguajes de programación lógica como PROLOG. LISP y PROLOG siguen siendo muy utilizados en las aplicaciones de la Inteligencia Artificial.

LL: Técnica descendente de análisis sintáctico, que consiste en examinar la entrada de izquierda a derecha (L) y construir una derivación por la izquierda (L).

LR: Son analizadores sintácticos ascendentes, que analizan gramáticas independientes del contexto. L: exámen de la entrada de izquierda a dererecha, R: construcción una derivación por la dererecha en orden inverso.

Parser: análisis sintáctico.

Parsers: analizadores sintácticos.

Parsing:

PASCAL: (é)ste lenguaje es uno de los mejores exponentes de la idea de programación estructurada o basada en estructuras disciplinadas para instrucciones secuenciales, de decisión y de repetición condicional e incondicional, creado en la d(é)cada de los 70, por el profesor suizo Niklaus Wirth, con el objetivo de disponer de un lenguaje de progrmaci(ó)n de alto nivel y prop(ó)sito general.

Patrón (pattern): Describe formalmente el conjunto de strings que se asocian a un *token*, es decir, son la especificación formal de un componente léxico.

Representación Postfija: Lista de nodos en la que figura el nodo en primer plano y sus hijos están por debajo. Es una representación lineal del árbol sintáctico.

UNIX:

YACC: El acrónimo se desglosa como “Yet Another Compilere-Compiler” y se traduce como ”otro compilador de compiladores” expresando, con un toque de modestia, que este era sólo un esfuerzo más por proveer herramientas que apoyaran la creación de compiladores, automatizando parte del trabajo. Pero YACC se convirtió en el ejemplo a seguir en automatización de la traducción, en la misma tradición técnica, que condujo a FLEX y luego a JFLEX y CUP, que se usan en este proyecto.