UNIVERSIDAD TECNOLÓGICA DE LA MIXTECA

COMPILADOR Y MÁQUINA VIRTAL PARA UN LENGUAJE DE SOCIEDADES DE AGENTES REACTIVOS

TESIS
PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN

PRESENTA

SAYDE ALCÁNTARA SANTIAGO

DIRECTORES DE TESIS

M.C. RICARDO RUIZ RODRÍGUEZ
M.C. CARLOS ALBERTO FERNÁNDEZ Y FERNÁNDEZ

Huajuapan de León, Oaxaca Febrero de 2006.

Agradecimientos

A Dios, por darme la oportunidad de abrirme las puertas en muchas de las etapas de mi vida.

A mis padres, de quienes he recibido las enseñanzas y las herramientas más valiosas de la vida.

A mis hermanas, Ary, Ita, y Huichi, por su gran cariño y apoyo que me ha permitido llegar a donde estoy. Ary, con quien comparto grandes luchas y esfuerzos. Ita, tu apoyo incondicional que me impulsa a seguir adelante. Huichi, tu gran cariño y afecto que necesito en mis momentos de reflexión.

A mis maestros que me han motivado con su compromiso y dedicación en la enseñanza. Aquellos quienes me impartieron su conocimiento y experiencia y que permitieron cultivar la motivación para seguir aprendiendo.

A mis asesores Ricardo Ruiz Rodríguez y Carlos Alberto Fernández y Fernández quienes me apoyaron y exhortaron en todo momento porque este proyecto rindiera frutos. Sin su valioso apoyo no hubiera sido posible llegar a este momento.

A todos mis amigos, por su desinteresado apoyo. A quienes tomaron como suyo este proyecto, dándome palabras de aliento y motivación para continuar.

A todos aquellos quienes de una u otra forma han cultivado enseñanzas de la vida en mí, por todos los buenos momentos de aprendizaje que he vivido con cada uno de ustedes

Resumen

El estudio de nuestro entorno ha llevado a la creación de ciencias con las que ha permitido la comprensión de nuestro comportamiento, sentimientos, reacciones, fenómenos humanos y naturales, nuestro mundo, nuestras propias facultades, para así mejorar nuestras condiciones de vida.

Uno de los campos de investigación surgido a raíz de explicar e imitar la forma en que el hombre se desenvuelve como un ente inteligente ha sido la Inteligencia Artificial.

El nombre de Inteligencia Artificial fue adoptado en 1956, por McCarthy [Mishkoff 88]; esto tras una reunión de científicos en Darmouth College para discutir la posible construcción de máquinas que se caracterizaran por realizar tareas "inteligentes".

Al ser una ciencia nueva, no se ha logrado establecer una definición universal para ella, tampoco existe una vertiente única hacia el logro de sus objetivos, los cuales también son variados y polémicos.

En esta búsqueda, se han creado robots "inteligentes" para imitar el comportamiento humano. Algunos se han enfocado en desarrollar actividades que son mecanizadas o programables, y otras que han reflejado un nivel de complejidad importante en actividades simples y cotidianas [Brooks 91].

Una de las vertientes de Inteligencia Artificial ha sido la basada en el comportamiento, la cual se centra en mostrar comportamiento inteligente en la realización de tareas. Este enfoque fue adoptado a principios de los años 80, cuando investigadores y centros de investigación del mundo estudiaron la forma en cómo se organizaba la. Llegaron a la opinión en común de que los sistemas inteligentes deben reaccionar a los aspectos dinámicos del ambiente [Atkinson 98].

En la Universidad Tecnológica de la Mixteca, como parte de un proyecto de investigación de los profesores-investigadores Dr. Héctor Gabriel Acosta Mesa y M.C. Carlos Alberto Fernández y Fernández, plantearon la creación de un nuevo lenguaje para la declaración de una sociedad de agentes reactivos, bajo la Arquitectura Subsumption. Con este lenguaje se pretende realizar trabajos experimentales en el área de Inteligencia Artificial por parte de los profesores-investigadores y alumnos interesados en el área. Además, la herramienta puede ser utilizada en los cursos de Compiladores e Inteligencia Artificial en la universidad.

Esta arquitectura va dirigida a construir robots autónomos para navegación y manipulación en interiores. El lenguaje propone la construcción una sociedad formada por uno o más agentes, donde a cada agente se le especifica sus conductas; y a su vez estas conductas conllevan ciertas acciones.

El lenguaje, llamado Age2000, se encuentra compuesto de estructuras de control sencillas. Se encuentra organizada en tres secciones: definición de una sociedad de agentes, definición de las conductas, definición de las acciones.

Para poder desarrollar programas experimentales bajo este lenguaje, se hace necesaria una herramienta que proporcione un ambiente adecuado de edición. Además, como valor agregado e ilustrativo, es necesario contar con una máquina virtual que ilustre el comportamiento de los agentes que comprende la sociedad.

El presente documento se encuentra dividido en capítulos organizados de la siguiente manera: el capítulo 1 expone, de forma más precisa, el problema a resolver, con la solución y los objetivos que se pretenden alcanzar.

El capítulo 2 comprende las bases teóricas y técnicas para la construcción del compilador y la máquina virtual.

Los capítulos 3, 4 y 5 tratan sobre la construcción específica de una herramienta para la edición de programas en lenguaje Age2000, además de proporcionar la construcción de mundos virtuales en los que se desenvolverá la sociedad de agentes.

Contenido

Lista de figuras

Capítulo 1: Descripción del Problema

1.1 Introducción

El hombre, por el afán de comprender el mundo que lo rodea, se ha dado a la tarea de estudiar los diferentes fenómenos que se suscitan en él, y con esto se ha dado la formación de las diferentes ciencias.

El logro de estos conocimientos ha llevado al hombre a comprender su mundo, y como consecuencia, a mejorar sus condiciones de vida.

Así mismo, el estudio del hombre como un ente inteligente, lo ha llevado a grupos de investigación que actualmente tratan de imitar, en la construcción de entidades físicas, el comportamiento humano.

Para que una entidad sea reconocida como inteligente, ¿basta con que se comporte como inteligente? o, ¿debe de razonar de forma inteligente? [Brooks 91]. Es decir, que no basta entender las palabras y las estructuras sino que hay que entender el tema. Esto lo expresa la siguiente cita:

"Debe haber un nivel adicional de entendimiento en el cual el carácter de las tareas de procesamiento de información sea analizado y entendido en una forma que sea independiente de los mecanismos particulares y estructuras que los implementan en nuestras cabezas"
David Marr
MIT

Estas ideas se ven claras cuando hablamos de creatividad, sentido común, generalidad, etc., es decir, cuando tomamos conceptos en los que el hombre no tiene una descripción precisa.

Entre las capacidades de la inteligencia humana, se pueden citar las siguientes destacables: raciocinio, comportamiento, desarrollo de metáforas y analogías, creación y uso de conceptos.

En los intentos por imitar estos comportamientos, surge la llamada Inteligencia Artificial, y esto no es nuevo, a lo largo de la vida se ha visto los intentos del hombre de dicha imitación.

Existen desarrollos importantes donde se puede observar las diferentes vertientes que existen dentro de Inteligencia Artificial, y ninguna de ellas ha sido definitiva, de hecho se puede decir que se complementan para el logro de sus objetivos.

1.2 Planteamiento del problema

En la búsqueda por comprender el entendimiento humano, y aún más, lograr imitarlo, surge el campo de Inteligencia Artificial que, mediante el planteamiento de sus diversos enfoques, pretende la comprensión y construcción de máquinas inteligentes.

No existe una definición universal de Inteligencia Artificial que englobe los fines que pretende alcanzar, pero entre los objetivos prioritarios se encuentra la comprensión de la inteligencia natural humana, y el uso de máquinas inteligentes para adquirir conocimientos y resolver problemas que se consideren como intelectualmente difíciles.

Se puede observar que la palabra clave es "inteligencia". El entendimiento del comportamiento humano implica la comprensión del significado "inteligencia".

En esta búsqueda, se han creado robots "inteligentes" para asemejar el comportamiento humano. Algunos se han enfocado en desarrollar actividades que son mecánicas o programables, y otras que han reflejado un nivel de complejidad importante en actividades simples y cotidianas [Brooks 91].

Dentro de esta línea, Rodney Brooks, profesor del MIT y miembro del Laboratorio de Inteligencia Artificial, ha creado robots que no tienen que conocer el mundo en el que se desenvuelven, a este tipo de entidades se les denomina modelos reactivos.

El comportamiento que reflejan estos robots, dentro de modelos reactivos, se fundamenta de acción y reacción, y reflejan inteligencia. Estos robots recorren su mundo con ayuda de sensores para detectar los obstáculos que se presentan, y actuadores para superarlos [Brooks 91].

Estos modelos también pueden ser ilustrados a través de software. La creación de software que simule el comportamiento de estos robots, también es un medio para la concepción de modelos reactivos. Tal es el caso del lenguaje Age2000 [Acosta & Fernández 00], que define una sociedad de agentes[1] con conductas básicas que permite que los agentes participen para alcanzar un fin. El lenguaje está basado en la Arquitectura Subsumption para el desarrollo de robots reactivos propuesta por Rodney Brooks [Brooks 86].

El lenguaje Age2000 comprende la definición de una sociedad de agentes, donde cada agente tiene un comportamiento, y acciones. La actuación inteligente de los agentes emerge de la interacción entre ellos y con su entorno [Acosta & Fernández 00].

Con este lenguaje se pretende realizar desarrollos experimentales en el área de Inteligencia Artificial por parte de los profesores-investigadores y alumnos interesados, además de poder ser una muestra ilustrativa de esta arquitectura en el curso de Inteligencia Artificial impartido en la universidad.

Sin embargo, dicho lenguaje no cuenta con un ambiente de desarrollo para la creación de programas que prueben este comportamiento y que ayude en la comprensión y experimentación con

[1] Entiéndase por agente en este documento, a un objeto dotado de sensores y actuadores para percibir y actuar en un mundo, para poder lograr una tarea u objetivo específico.

tales teorías; por lo que el presente trabajo de tesis propone la creación de una herramienta que apoye en la edición de programas en el lenguaje Age2000, como también un ambiente que simule el comportamiento declarado para la sociedad de agentes.

La construcción del compilador, para el reconocimiento y validación del lenguaje Age2000, puede servir como una herramienta de apoyo en la enseñanza de la materia Compiladores de la universidad. Esto es, se muestra mecanismos específicos para la construcción del compilador, como la técnica de programación orientada a objetos para su implementación.

1.3 Solución Propuesta

El lenguaje Age2000 comprende tres secciones: definición de una sociedad de agentes, definición de conductas, definición de acciones. El lenguaje cuenta con estructuras de control y tipos de datos básicos, además de proporcionar primitivas de percepción las cuales se encuentran asociadas a dispositivos de sensado del robot.

De acuerdo al lenguaje Age2000, el planteamiento de una sociedad de agentes se dará mediante la interacción con el medio que les rodee. Así, los agentes se desenvolverán en el mundo mediante sensores. Y cuando estos sensores se activen, entonces toca el turno a los actuadores llevar a cabo su reacción. Todas estas conductas de los agentes en una sociedad se realizan de forma paralela, por lo que es necesario implementar un agente llamado "oráculo" para la sincronización de las peticiones, así como la resolución de los conflictos. Esta arquitectura se basa en la llamada Arquitectura Subsumption [Brooks 91].

En la Figura 1.1 se muestra la arquitectura del lenguaje Age2000 bajo la cual será declarada la sociedad de agentes. A cada agente se le asocia una serie de conductas, así como las acciones a tomar en caso de que una conducta solicite alguna acción.

Una sociedad se define como un conjunto de agentes cuya estructura está constituida por una o más conductas de propósito específico que reaccionan a estímulos detectados por sensores (percepción), mediante la activación de un actuador (acción).

Para el uso de este lenguaje, y para poder llevar a cabo trabajos experimentales, es preciso contar con una herramienta de desarrollo que facilite la construcción de sociedades de agentes; además de que es importante contar con una interfaz en la que se pueda observar el comportamiento de la sociedad declarada.

Con esto se busca que el desarrollo de programas en el lenguaje Age2000 se facilite contando con un software que permita la edición de dichos programas en un ambiente amigable (IDE - *Integrated Development Environment*), y además se realice el análisis sobre la correcta declaración de las estructuras en el lenguaje Age2000, sintetizándolo a un lenguaje equivalente.

Para esto último, es necesario la construcción de un compilador que analice y sintetice el programa fuente, y traduzca ese lenguaje en uno equivalente (código objeto). En el caso que existan errores en la declaración del lenguaje Age2000 en el programa fuente, el compilador tiene como tarea el dar aviso de la existencia de éstos.

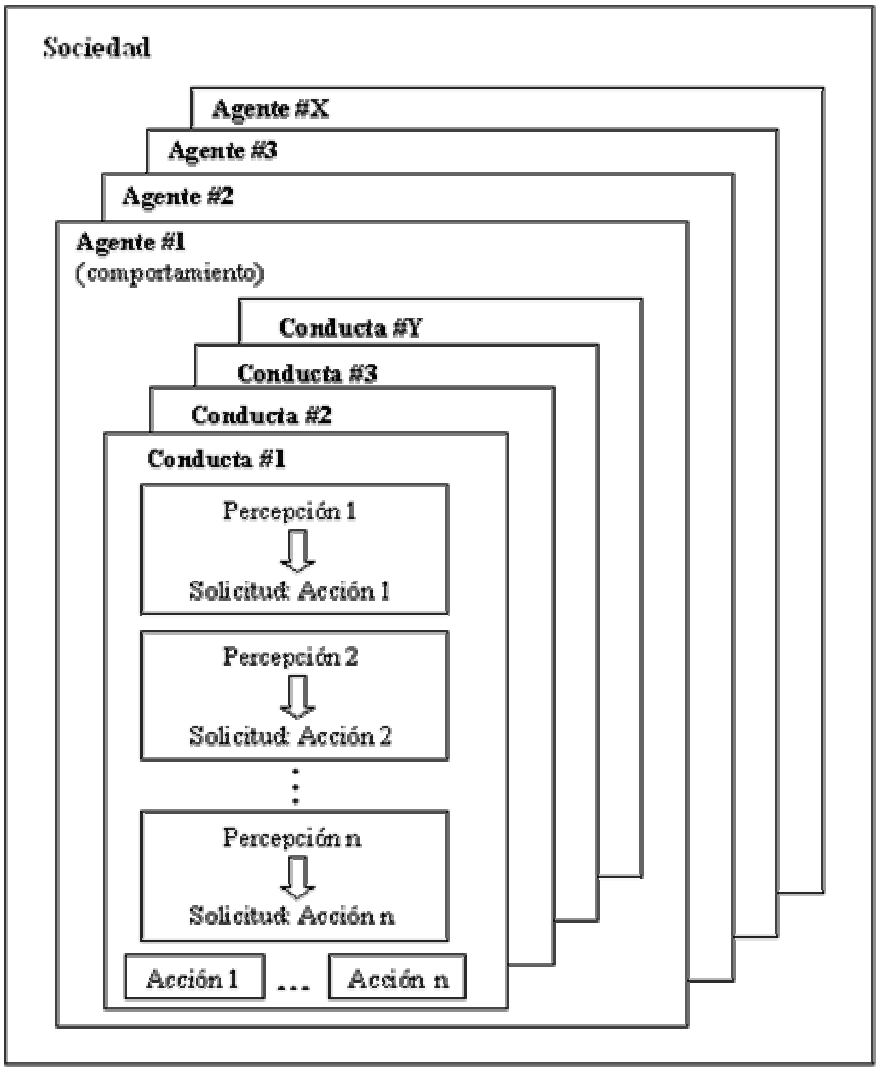


Fig. 1.1 *Arquitectura del lenguaje Age2000 [Acosta & Fernández 00].*

Adicionalmente, para contar con un medio ilustrativo de las conductas de los agentes en la sociedad, es necesario elaborar un ambiente de simulación.

Para esto, a partir del código objeto generado por el compilador, es preciso contar con una máquina virtual que simule la interacción entre los agentes y el mundo. Es decir, para lograr esta simulación de los agentes en un mundo, es necesario contar con un ambiente que nos ayude a definir estructuras básicas para la construcción de este mundo, en donde se pueda observar el comportamiento de los agentes.

Por la necesidad de experimentar en el campo de agentes reactivos, y dado que no se cuenta con una herramienta que contenga estas características, es que se propone la construcción de un sistema que permita el desarrollo de programas experimentales.

1.4 Objetivos

El objetivo del presente proyecto de tesis es:

Proporcionar una herramienta que permita al desarrollador definir sociedades de agentes fácilmente, reflejando su comportamiento en un simulador básico, resultando transparente la forma en que se sincronizan, y permitiendo rápidos desarrollos experimentales.

Se destacan cuatro objetivos específicos:

- Construir un compilador que permita definir, mediante el lenguaje Age2000, una sociedad de agentes reactivos cada uno con conductas básicas para alcanzar un fin común.
- Construir un IDE[2] para el desarrollo de programas basados en lenguaje Age2000.
- Construir un IDE para la elaboración del mundo en el que actuarán los agentes de la sociedad.
- Construir una máquina virtual, como un valor agregado, que permita simular la interacción de los agentes como resultado del comportamiento que emerge de la interacción de las conductas que los forman, aún sin conocer la existencia de otros agentes o tener un modelo del mundo en que se desenvuelve.

Para alcanzar estos objetivos, se debe de destacar la importancia de elaborar un buen diseño del sistema en general. Es por esto que se hace uso de herramientas que apoyen a un buen desarrollo de software, tal como el empleo de UML.

[2] IDE - *Integrated Development Environment.*

Capítulo 2: Marco Teórico

2.1 Introducción

El proyecto de tesis parte del lenguaje de programación Age2000, el cual está basado en la arquitectura Subsumption. Esta arquitectura está dirigida al desarrollo de robots reactivos en el que el comportamiento de un agente está definido por una serie de conductas básicas e independientes que se ejecutan en forma paralela.

Con miras a lograr los objetivos en la construcción de la herramienta que permita definir fácilmente una sociedad de agentes, es necesario implementar un compilador para validar la correcta declaración en el lenguaje Age2000. Como aportación adicional, se provee de una máquina virtual para observar el comportamiento que emerge de la convivencia de los agentes.

A continuación se da inicio al estudio de compiladores, exponiendo las partes que conforman el compilador, y esencialmente las técnicas que fueron utilizadas para el desarrollo del presente proyecto.

2.2 Compiladores

Para que exista una comunicación entre el humano y la computadora se necesita de un lenguaje, el cual permita la declaración de expresiones que ayuden a resolver un problema.

A lo largo del tiempo, ha habido una evolución en estos lenguajes que ha brindado la declaración de expresiones más complejas [Teufel et.al. 95].

Dada las exigencias por parte de estos nuevos lenguajes de programación se fueron descubriendo técnicas sistemáticas para manejar muchas de las importantes tareas que surgen para la validación de un programa de acuerdo a un lenguaje, es decir, el proceso de compilación.

El término “compilador” fue introducido por primera vez en 1950 [Aho et.al. 98], apareciendo los primeros trabajos que mostraban a un compilador como un programa difícil de implantar. Por ejemplo, el primer compilador de FORTRAN necesitó para implantarse 18 años de trabajo en grupo.

Con un compilador se logra una mejor comunicación con la computadora, a partir de lenguajes que permiten la eficiencia en la resolución de problemas, optimización del tiempo, localización de errores con rapidez, claridad en la programación, más complejidad en el planteamiento del problema, etc. Además, se han desarrollado buenos entornos de programación y herramientas de software.

Por fortuna, con algunas técnicas básicas de escritura de compiladores se puede construir traductores para una gran variedad de lenguajes y máquinas.

Para iniciar con este estudio, se presenta la definición de lo que es un compilador, y las fases que lo componen desde una perspectiva conceptual; y se exponen las técnicas adoptadas para la construcción del compilador del lenguaje Age2000.

Entiéndase como compilador a un programa que lee un programa escrito en un lenguaje determinado, programa fuente, y lo traduce a un programa equivalente en otro lenguaje, programa objeto. Como parte importante de este proceso de traducción, el compilador informa al usuario de la presencia de errores en el programa fuente [Aho et.al. 98].

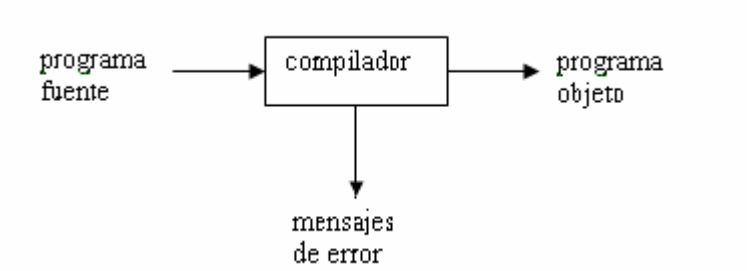


Fig. 2.1 Esquema de *un compilador [Aho et.al. 98]*.

2.2.1 Fases de un compilador

El proceso de compilación se divide en dos etapas [Aho et.al. 98]:

1) **Análisis.-** Divide al programa fuente en sus elementos componentes y crea una representación intermedia del programa fuente.

2) **Síntesis.-** Construye el programa objeto deseado a partir de la representación intermedia.

Desde un punto de vista conceptual, un compilador opera en fases (en la realidad, esta separación es menos estricta). Estas fases se esquematizan en la siguiente figura.

A continuación, se describen brevemente estas fases [Aho et.al. 98]:

Análisis Léxico. También llamado análisis lineal o scanner. Es la primera fase de un compilador. Su principal función es leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer el análisis.

Análisis Sintáctico. También llamado análisis jerárquico o parser. Tiene como principal función el comprobar si la cadena pueda ser generada por la gramática del lenguaje fuente, es decir, revisa si los símbolos aparecen en el orden correcto y forman unidades gramaticales.

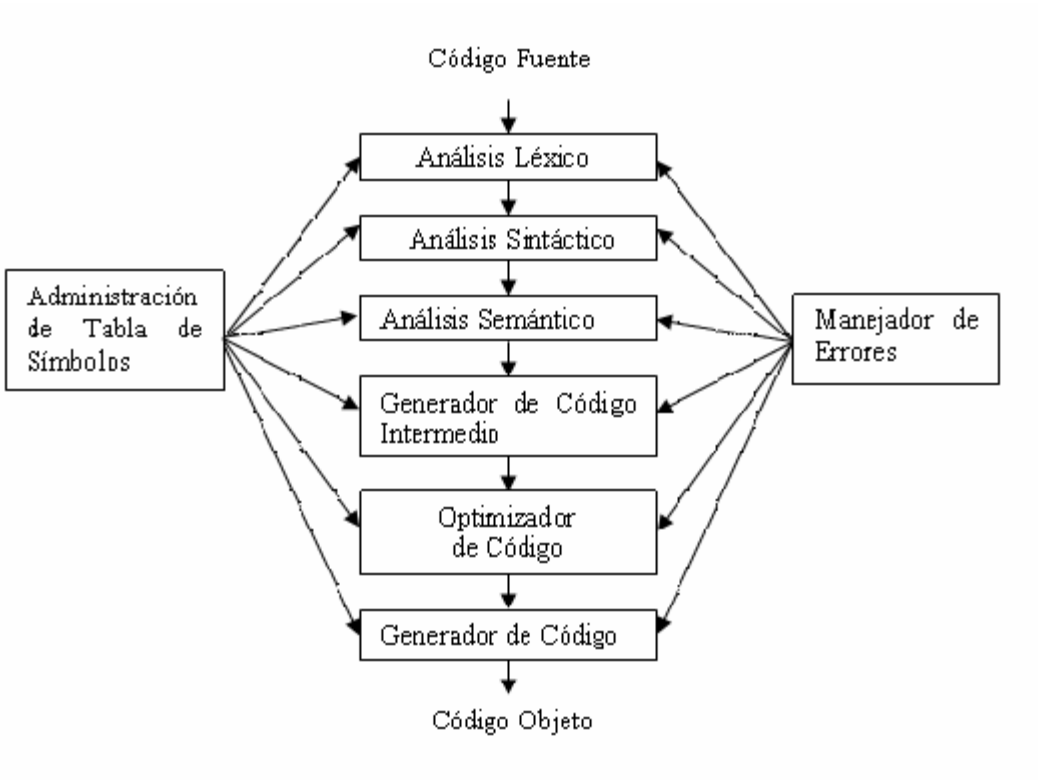


Fig. 2.2 *Fases de un compilador [Aho et.al. 98].*

Análisis Semántico. Su función se centra en comprobar que la cadena de componentes léxicos tenga significado. Un componente importante del análisis semántico es la verificación de tipos.

Generación de Código Intermedio. Traduce el programa fuente a una representación intermedia a partir de la cual la etapa final genera el código objeto.

Optimización de Código. Su función es mejorar el código objeto con la finalidad de que el código producido se ejecute más rápidamente y/o ocupe menos espacio.

Generación de Código. Como el nombre de la fase lo dice, su función es generar el código objeto. Cada instrucción intermedia se traduce a una secuencia de instrucciones de máquina que ejecutan la misma tarea.

Además, se cuenta con un administrador de la tabla de símbolos, que es una estructura de datos donde se almacenan los identificadores utilizados en el programa fuente así como sus distintos atributos, es decir, se almacena cada identificador, tipo de dato al que pertenece; si es un procedimiento, almacena sus argumentos y el tipo de argumento que regresa (si es que lo hace).

Los compiladores deben ayudar al programador a identificar y localizar errores. En cualquiera de las fases del compilador se pueden encontrar errores, así que es necesario que cada fase maneje los errores adecuadamente, de tal manera que permita continuar la compilación, y detectar más errores en el código fuente.

A menudo, la detección y recuperación de errores se centra en la fase de análisis sintáctico, esto es debido a que muchos de los errores son de naturaleza sintáctica o se manifiestan cuando la cadena de componentes léxicos (provenientes del analizador léxico) no concuerda con las reglas gramaticales que definen al lenguaje de programación.

En las siguientes secciones se tratan con mayor detalle los procesos que siguen las fases de análisis.

2.2.2 Análisis Léxico

El analizador léxico es la primera fase de un compilador. Su función principal es leer los caracteres de entrada de un programa fuente y elaborar como salida una secuencia de componentes léxicos (también llamado tokens) [Salas 92] que utiliza el analizador sintáctico para hacer el análisis [Aho et.al. 98].

Además, el analizador léxico tiene las tareas de eliminar información innecesaria (como comentarios, espacios en blanco), introducir información preliminar a la tabla de símbolos, y relacionar los errores encontrados con el programa fuente [Fischer & LeBlanc 88].

Durante el análisis léxico, se pueden encontrar errores que necesariamente debe tratar, recuperarse de los errores y reportarlos para poder proseguir con el análisis.

Las estrategias de recuperación que se pueden adoptar en el analizador léxico pueden ser: recuperación en "modo de pánico", es una de las estrategias más sencillas en las que se borran caracteres sucesivos de la entrada restante hasta que el analizador pueda encontrar un componente léxico bien formado; borrar un carácter extraño; insertar un carácter que falta; reemplazar un carácter incorrecto por otro correcto; intercambiar dos caracteres adyacentes [Aho et.al. 98].

Para la implantación de un analizador léxico existen tres métodos generales [Aho et.al. 98]:

- Utilizar un generador de analizadores léxicos. Producir el analizador léxico a partir de una especificación basada en expresiones regulares (como LEX).
- Desarrollar el analizador léxico en un lenguaje convencional de programación de sistemas.
- Desarrollar el analizador léxico en lenguaje ensamblador.

Para el caso del compilador para el lenguaje Age2000, el método adoptado fue el desarrollar el analizador léxico mediante un lenguaje convencional de programación, el cual es java, mediante la tecnología de programación orientada a objetos. La implementación del analizador léxico se ve en el Capítulo 3 en la Sección 3.2.

En cualquier caso, para iniciar la construcción del analizador léxico, es importante definir las expresiones regulares correspondientes al lenguaje en cuestión. Estas expresiones regulares permiten especificar la estructura de los componentes léxicos (tokens) usados en un lenguaje de programación. El conjunto de cadenas definidas por expresiones regulares son llamadas *conjunto regular* [Fischer & LeBlanc 88].

La importancia del uso de expresiones regulares para definir las reglas lexicográficas de un lenguaje, radica principalmente en las siguientes razones [Aho et.al. 98]:

- Las reglas lexicográficas de un lenguaje frecuentemente son sencillas, así que definirlas no se necesita una notación poderosa como las gramáticas, éstas últimas se explican en la siguiente sección.
- Las expresiones regulares proporcionan una notación más concisa y fácil de entender.
- Se pueden construir analizadores léxicos eficientes a partir de expresiones regulares.

Los lenguajes aceptados por autómatas finitos (AF) se describen con expresiones regulares. (Uno de los principales usos de los autómatas es la construcción de analizadores léxicos [Hopcroft & Ullman 79]).

Se entiende por autómata a un dispositivo que recibe una entrada y, pasando por varios estados, produce la salida apropiada.

Los autómatas pueden representarse gráficamente usando diagramas de transición (grafo de transiciones), los cuales usan los elementos que se muestran en la Fig. 2.3 [Fischer & LeBlanc 88].

Los autómatas finitos se pueden dividir en: no determinísticos y determinísticos [García et.al. 01].

Un autómata finito no determinístico (AFN) es un dispositivo de reconocimiento en el que para cada estado y cada carácter de entrada, puede existir más de un estado de transición, y además permite transiciones para la entrada vacía.

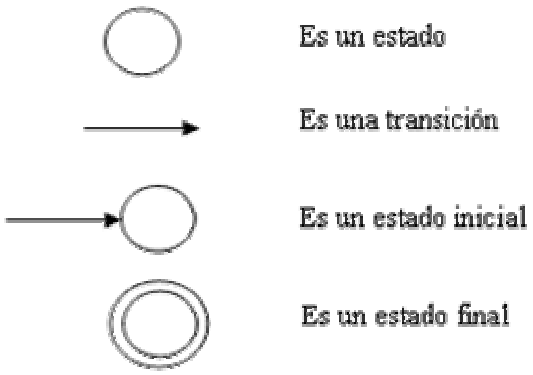


Fig. 2.3 *Elementos de diagramas de transición [Fischer & LeBlanc 88]*.

Un autómata finito determinístico (AFD) se emplea como dispositivo de reconocimiento que discrimina entre palabras de entrada en un alfabeto finito. Los estados tienen un próximo estado único (para un caracter y estado dado), es decir, que sólo se cuenta con una transición desde cada estado con cualquier entrada. Además, no permite transiciones para la entrada vacía.

Formalmente, un AFD consiste de [Hopcroft & Ullman 79]:

- Un conjunto de estados, algunas veces denotado por Q.
- Un conjunto de símbolos de entrada, algunas veces denotado por $\sum$.
- Una función de transición que toma como argumentos un estado y un símbolo de entrada y regresa un estado. La función de transición comúnmente será denotado por δ. La función de transición, δ, ha sido representada mediante arcos entre estados y la etiqueta sobre los arcos. Si q es un estado, y a es un símbolo de entrada, entonces $\delta(q,a)$ es el estado p tal que hay un arco etiquetado con el símbolo a del estado q a p.
- Un estado de inicio, estado incluído en Q.
- Un conjunto de estados de aceptación o finales denotado por F. El conjunto F es un subconjunto de Q.

Por lo tanto, un AFD A está representado por la quíntupla,

$A = (Q, \sum, \delta, q_0, F)$,

donde Q es el conjunto de estados, $\sum$ es el conjunto de símbolos, δ es la función de transición, q_0 es el estado inicial, y F es el conjunto de estados de aceptación.

La forma de implementación de un AF es codificarlo en una tabla de transiciones, cuya adaptación toma menos espacio. La tabla de transiciones es representada por un arreglo de dos dimensiones [Hopcroft & Ullman 79].

Las filas de la tabla de transiciones, hacen referencia a cada estado del autómata; y las columnas, hacen referencia a cada símbolo de entrada [Aho et.al. 98].

Por medio de la tabla de transiciones se simula el autómata, y permite localizar las cadenas de caracteres que corresponden a identificadores, constantes numéricas, palabras reservadas etc. En este proceso, el analizador léxico sólo necesita recordar una cantidad finita de información [García et.al. 01].

Entonces, para lograr la construcción de un Analizador Léxico es necesario definir las expresiones regulares, y entonces construir su representación en un AFD. Una vez hecho, se debe de vaciar la información del AFD a una tabla de transiciones. Esto es, para que con cada carácter de entrada se logre la identificación de los componentes léxicos, o en otro caso, la identificación de una cadena no permitida en el lenguaje.

2.2.3 Análisis Sintáctico

Como ya se mencionó, el analizador sintáctico tiene como principal función el validar una cadena de componentes léxicos, provenientes del analizador léxico, dentro de la gramática del lenguaje fuente. En el caso de que hubiera errores, el analizador sintáctico tiene que informar de esos errores, así como también se debe de recuperar para poder continuar procesando el resto de su entrada [Aho et.al. 98].

Para poder compilar un programa escrito en un lenguaje de programación específico es necesario conocer la definición de dicho lenguaje. Definir un lenguaje de programación significa describir la sintaxis y la semántica del mismo.

Para ello, se puede especificar la sintaxis de un lenguaje de programación en un lenguaje formal usando la forma de Backus-Naur (BNF – *Backus Normal Form*) o también llamada gramática independiente del contexto (a la que en adelante se mencionará como gramática) [Hopcroft & Ullman 79].

BNF es una notación la cual describe de forma natural, la estructura jerárquica de muchas construcciones de los lenguajes de programación [Teufel et.al. 95] [Aho et.al. 98].

La gramática independiente del contexto tiene cuatro componentes [Teufel et.al. 95] [Aho et.al. 98]:

- Símbolos terminales: Conjunto de componentes léxicos, es decir, son los símbolos válidos del lenguaje.
- Símbolos no terminales: Son variables sintácticas.
- Producciones: Las reglas para la sustitución de cadenas. Se compone de un no terminal, llamado lado izquierdo, una flecha, y una secuencia de componentes léxicos y no terminales, o ambos, llamado lado derecho.
- Símbolo Inicial: La denominación de una símbolo no terminal como símbolo inicial.

Para especificar la gramática se realiza una lista de sus producciones, donde las producciones del símbolo inicial se listan primero. Los símbolos no terminales se mostrarán entre "<" y ">", los símbolos terminales en negritas, siguiendo la notación BNF de la Tabla 2.1 [Teufel et.al. 95].

Tabla 2.1 *Notación BNF [Teufel et.al. 95]*

Símbolo	Significado
→	"se define como"
|	"or", alternativa
[x]	una o ninguna ocurrencia de x
{x}	número arbitrario de ocurrencias de x(0, 1, 2, ...)
(x | y)	selección (x o y)

De la construcción de la gramática se puede destacar que su uso, para el diseño de lenguajes y compiladores, proporciona las siguientes ventajas [Aho et.al. 98]:

- Una gramática da una especificación sintáctica precisa y fácil de entender de un lenguaje de programación.
- A partir de algunas clases de gramáticas se puede construir automáticamente un analizador sintáctico eficiente que determine si un programa fuente es sintácticamente correcto.
- Una gramática diseñada adecuadamente imparte una estructura a un lenguaje de programación útil para la traducción de programas fuente a código objeto apropiado y para la detección de errores.

- Los lenguajes evolucionan con el tiempo, adquiriendo nuevas construcciones y realizando tareas adicionales. Estas nuevas construcciones se pueden añadir con más facilidad a un lenguaje cuando existe una aplicación basada en una descripción gramatical del lenguaje.

Una vez especificado el lenguaje de acuerdo a la notación BNF, se debe especificar el sentido de las derivaciones, la aplicación de una producción para sustituir un símbolo no terminal ("derivación de"). Cuando se deriva una secuencia de palabras, si más de un símbolo no terminal está presente, se tiene que elegir cuál es el siguiente símbolo no terminal a derivar. Para caracterizar una secuencia de derivaciones, se necesita especificar en cada paso, qué símbolo no terminal se expande, y qué producción debe aplicarse.

Existen dos sentidos de derivación: derivación por la izquierda, y derivación por la derecha. Para el caso de la derivación por la izquierda, el símbolo no terminal a expandir es el que se encuentre más a la izquierda de esa producción. A su vez, la derivación por la derecha consiste en sustituir el símbolo no terminal que se encuentre más a la derecha de tal producción. De este modo, se puede realizar la derivación de una entrada y así observar los caminos que va tomando para generarla mediante la gramática [Ullman 76].

Hay una manera gráfica para observar estas derivaciones, que es mediante la construcción de árboles de análisis sintácticos. Estos árboles sintácticos muestran cómo el símbolo inicial de una gramática deriva una cadena del lenguaje, aunque no muestra la elección relativa al orden de sustitución.

Adicionalmente, por medio de los árboles sintácticos podemos reconocer una gramática ambigua, ya que generaría más de un árbol sintáctico diferente para una misma entrada, y por lo tanto, una estructura no única [Ullman 76].

La derivación adoptada para la gramática del lenguaje Age2000 es derivación por la izquierda. Ésta se explica en el Capítulo 3.

Hay que destacar la importancia que tiene la construcción de la gramática, ya que de ella se debe generar el lenguaje de interés. Al construirla se tiene que tener cuidado en no caer en ambigüedades, en producciones inalcanzables, o símbolos inútiles; y de todos, el más grave, sería el que la gramática generara un lenguaje incorrecto.

Una vez creada la gramática, se debe elegir el método sobre el que se construirá el analizador sintáctico. Básicamente, existen dos métodos generales para la construcción de un analizador sintáctico: método descendente, y ascendente [Aho et.al. 98]. Los términos descendente y ascendente hacen referencia al orden en que se construyen los nodos del árbol de análisis sintáctico. En el caso de métodos descendentes la construcción parte del símbolo inicial (raíz), llegando a las hojas (símbolos terminales), y para el caso de los métodos ascendentes es viceversa.

La popularidad que tienen los métodos descendentes radica en que facilitan la construcción manual de analizadores sintácticos eficientes. Bajo esta clase destacan los siguientes métodos: Análisis Sintáctico Descendente Recursivo, Análisis Sintáctico Predictivo No Recursivo [Aho et.al. 98].

Para el caso de los métodos ascendentes, su uso radica en que puede manejar una clase mayor de gramáticas y esquemas de traducción. Por esta razón, las herramientas de software para

generar analizadores sintácticos basan su construcción en métodos ascendentes. Entre los métodos ascendentes se encuentran: Análisis Sintáctico LR, Análisis Sintáctico por Desplazamiento y Reducción, Análisis Sintáctico SLR, etc. [Aho et.al. 98].

Para la realización de este proyecto, se selecciona el método de Análisis Sintáctico Descendente Recursivo, específicamente el Análisis Sintáctico Predictivo, como el método bajo el cual se desarrolla el analizador sintáctico para el lenguaje Age2000, ya que sus características permiten el poder construirlo bajo un paradigma de programación orientada a objetos, y es bajo este paradigma con el cual se diseña e implementa el sistema completo.

El Análisis Sintáctico Descendente Recursivo consiste en la ejecución de un conjunto de procedimientos recursivos para procesar la entrada. A cada símbolo no terminal de una gramática se asocia un procedimiento. Así que, cuando se analiza una entrada, se inicia el análisis con el símbolo inicial que es un símbolo no terminal el cual tiene asociado un procedimiento, y continúa ingresando a las reglas de producción que generen la entrada, ya sea en comparación directa del componente léxico esperado con el leído, o entrando a los procedimientos que hacen referencia a los símbolos no terminales.

Por lo anterior, para la implantación del paradigma de programación orientada a objetos para el analizador sintáctico del lenguaje Age2000, se asocia cada símbolo no terminal de la gramática con una clase. La elección de programación orientada a objetos ha sido utilizada para poder proponer y experimentar con este modelado en la construcción de un compilador.

Como ya se ha mencionado, es muy importante tener cuidado en la construcción de la gramática a analizar. Para este caso, la gramática adopta las siguientes características [Aho et.al. 98]:

La gramática no es recursiva por la izquierda. Esto es, en caso de que existan producciones de la gramática que tengan como su primer símbolo de derivación al mismo símbolo no terminal que lo deriva (es decir, $A \rightarrow A\ \alpha$, donde α es una cadena de símbolos terminales y/o no terminales) se deben de replantear, ya que dicha forma implica recursividad por la izquierda.

Hace uso de factorización por la izquierda. En el caso que en la gramática existan producciones que al inicio de su alternativa presenten los mismos símbolos (símbolos terminales o símbolos no terminales) y que el resto de esa derivación sea la diferente, entonces se factoriza. La factorización consiste en que la parte que se tiene en común colocarla en una producción y el resto de ambas producciones colocarlas como alternativas de otra producción.

Con estas características se puede hablar de una gramática analizable con un Analizador Sintáctico Descendente Recursivo que no necesite retroceso, es decir, un Análisis Sintáctico Predictivo.

Por consiguiente, el Análisis Sintáctico Predictivo está basado en la gramática LL[3], la primera "L" significa que será leída de izquierda a derecha, y la segunda "L" indica derivaciones por la izquierda. Una gramática LL(1) significa que su lectura y derivación serán por la izquierda, y el "1" significa que se lee por anticipado un símbolo en cualquier paso del proceso de análisis.

[3] "L" viene de la palabra "left" en inglés que significa "izquierda".

Para continuar con el estudio del método, y específicamente de las características de la gramática LL, es necesario tener la idea clara de los siguientes conceptos: PRIMERO, SIGUIENTE.

El conjunto llamado PRIMERO se refiere a todos los símbolos terminales que pueden aparecer al principio de una frase que puede derivarse de una secuencia arbitraria de símbolos. Y el conjunto SIGUIENTE, se refiere a todos los símbolos terminales que pueden aparecer después de uno no terminal.

Una gramática independiente del contexto G (N, T, P, S) se denomina gramática LL(1) si tiene las siguientes características:

Para cada producción

$$A \rightarrow \sigma_1 \mid \sigma_2 \mid \ \ldots \ \mid \ \sigma_n$$

Se requiere que

$$PRIMERO(\sigma_i) \cap PRIMERO(\sigma_j) = \Phi \text{ para todo } i \neq j$$

Si puede derivarse la cadena vacía ε de un símbolo no terminal X, se requiere que

$$PRIMERO(X) \cap SIGUIENTE(X) = \Phi$$

La primera característica expresa que, en la derivación de una cadena será obvia la alternativa de la producción que deberá aplicarse. Y la segunda característica contempla el uso de la cadena vacía dentro de las producciones.

Por todo lo anterior, se resume que es de especial interés el estudio de gramáticas LL(1), ya que éstas no requieren de retrocesos para elegir la alternativa correcta dentro de la producción para generar la cadena de entrada. Esto es porque las gramáticas LL(1) recurren a su conjunto PRIMERO (σ_i) para elegir la alternativa dentro de la producción que le lleve a la generación de la entrada. Además, estas gramáticas no aceptan ambigüedades, o recursividad por la izquierda.

Es por medio de la gramática LL(1) que se construye la gramática para el lenguaje Age2000, para ser analizado bajo el método Análisis Sintáctico Predictivo.

Una vez construida la gramática de acuerdo a las características de gramáticas LL(1), se debe contemplar la forma en que se tratarán los errores en esta fase

Se tiene que tener presente que un programador siempre cometerá errores. Así que es necesaria la implementación de modos de recuperación de errores. De estos destacan los siguientes:

- *Recuperación en modo de pánico*. Al encontrar un error, el analizador sintáctico desecha los símbolos de entrada, hasta que se encuentra uno perteneciente a un conjunto designado de componentes léxicos de sincronización, los cuales pueden ser: “punto y coma”, }, alguna palabra reservada como “end”, etc.

- *Recuperación a nivel de frase*. Al descubrir un error, el analizador sintáctico puede realizar una corrección de la entrada del analizador, sustituyendo un símbolo detectado como incorrecto por alguno que permita continuar al analizador. Por ejemplo: sustituir una “coma” por un “punto y coma”, insertar un “punto y coma” faltante, etc.

- *Producciones de error*. Cuando se conocen los errores más comunes que se encuentran en los programas fuente de un lenguaje, es posible aumentar la gramática del lenguaje con producciones que generen las construcciones erróneas. Se aumenta esta gramática al analizador sintáctico y si el analizador usa una producción de error, es posible generar diagnósticos de error apropiados de acuerdo a la entrada reconocida.

- *Corrección global*. Esta consiste en usar un algoritmo para encontrar la mínima modificación a una cadena incorrecta para hacerla permitida por la gramática del lenguaje.

Como se ha mencionado, es importante detectar los errores y poder recuperarse de ellos, ya que con esto el analizador sintáctico podrá continuar con la revisión del resto del programa. Un mecanismo de recuperación de error debe colocar al analizador sintáctico en un estado en el que el procesamiento de la entrada pueda continuar con altas posibilidades de que el análisis de los siguientes componentes léxicos será efectuado correctamente.

2.2.4 Análisis Semántico

Este análisis revisa el programa fuente para tratar de encontrar errores semánticos, además de reunir la información sobre los tipos para la fase posterior de generación de código. La identificación de los tipos de las variables es un punto importante ya que debe verificar la validez de los operandos y valores permitidos por el lenguaje.

Las Tablas de Símbolos se emplean para revisar si un identificador ya ha sido declarado y su uso puede considerarse como parte del proceso de análisis semántico. El análisis semántico puede realizarse en paralelo con el análisis sintáctico.

El análisis semántico de un compilador usa la información del análisis sintáctico en combinación con las reglas semánticas del lenguaje de programación para generar una representación intermedia del código fuente que va a compilar o el mismo código objeto. En caso de que la representación interna sea un código intermedio, deberá enviarse al generador de código.

De hecho, se puede decir que el analizador semántico es el puente entre el análisis sintáctico y la fase de generación de código, ya que reúne información acerca del uso y definición de las variables usadas en el programa, la cual puede ser usada por la fase de optimización de código. Esto es, por ejemplo, una variable que haya sido declarada y no haya tenido ningún uso en el programa.

2.3 Máquina Virtual

Hasta este punto se han expuesto los fundamentos que soportan la construcción del compilador para el lenguaje Age2000. En esta sección se trata el papel que cumple la máquina virtual dentro de la herramienta.

La elección de una máquina virtual es por el tipo de código objeto generado. Un compilador se puede distinguir de acuerdo a la clase de código objeto que genera, el cual puede ser:

a) Código máquina puro. Genera código para el conjunto de instrucciones de una máquina en particular, asumiendo que no existe sistema operativo.

b) Código máquina aumentado. Genera código para la arquitectura de una máquina aumentada con un sistema y bibliotecas del lenguaje.

c) Código de máquina virtual. El código generado se compone de instrucciones virtuales, éstas deben ser ejecutadas por otros programas que se conocen como máquina virtual o como intérprete de bajo nivel.

Una vez que se haya validado la correcta declaración de una sociedad de agentes (función que estará a cargo del compilador), el siguiente paso es simular este comportamiento, responsabilidad que cae en la máquina virtual.

Del compilador se obtendrá el código objeto intermedio que contendrá una representación a un código especial equivalente del código fuente en lenguaje Age2000. La máquina virtual traducirá el código objeto intermedio en la simulación del comportamiento de los agentes declarados.

Una computadora, para la ejecución de un programa, necesita de un conjunto de registros que indican cuál es la siguiente instrucción a ejecutar, dónde se encuentran los operandos de la siguiente operación en la memoria, etc. Una máquina virtual también debe tener una estructura que lleve un registro de su estado actual.

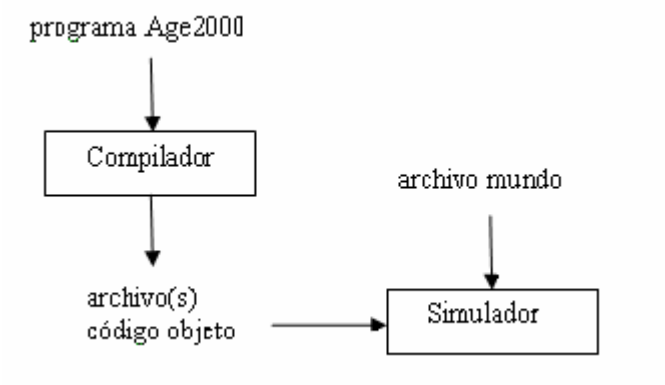


Fig. 2.4 *Flujo de datos para la construcción de la máquina virtual.*

La máquina, para el lenguaje de expresiones aritmético-lógicas, está formada por los siguientes elementos:

- *Memoria de código objeto (MC).* Ésta es un área de memoria donde se almacena el código objeto a ejecutar, y que es producido por el compilador como resultado de la traducción del archivo fuente.

El primer paso que debe hacer la máquina virtual consiste en tomar el archivo de código objeto y guardarlo en la memoria de código objeto.

- *Apuntador a la siguiente instrucción (PC – Program Counter)*. Ésta es una variable que representa la siguiente instrucción a ser ejecutada. Esta variable se puede decir que apunta a una localidad de MC. Entonces MC[PC] representa la siguiente instrucción a ser ejecutada.

- *Pila de evaluación*. En esta pila se almacenan todos los operandos de la expresión de acuerdo al orden de evaluación, y en ella se efectúan todas las operaciones.

- *Tope de la pila de evaluación*. Indica el último valor que fue colocado en la pila de evaluación.

- *Intérprete*. El intérprete no es una estructura de datos, es una función que con la memoria de código y la pila de evaluación, lleva a cabo la evaluación del programa o expresión compilada.

El intérprete desempeña el papel de la unidad central de procesos efectuando un ciclo repetitivo – Ciclo de Fetch - en el cual ejecuta instrucción por instrucción hasta encontrar la operación de fin.

Lo anterior, se puede observar en el siguiente diagrama:

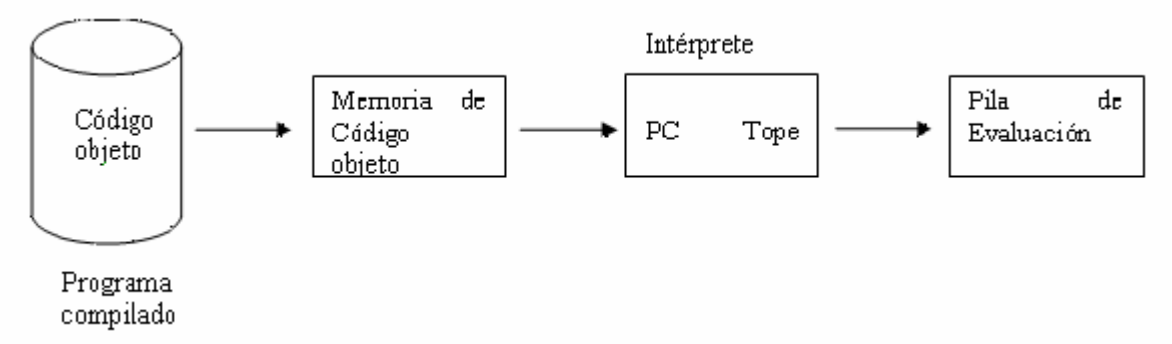


Fig. 2.5 *Elementos de una máquina virtual en expresiones aritmético-lógicas.*

El Ciclo de Fetch consiste de los siguientes 4 pasos:

1. Obtener la siguiente instrucción apuntada por el registro PC.
2. Incrementa el contenido del PC.
3. Ejecución de la operación referida por el código obtenido en el paso 1. La función puede incluir argumentos y dependiendo de su función puede afectar el estado del intérprete.
4. Si la instrucción es paro de ejecución, entonces se da fin a la ejecución, de otro modo, se continúa en el paso 1.

Para poder obtener la simulación de la sociedad de agentes se debe contar con un ambiente, que proporciona la herramienta, donde se construyan mundos virtuales en donde se desenvuelvan los agentes. Una vez construido el mundo, la máquina virtual se ejecutará dotando a los agentes de funcionalidad y reflejando su comportamiento.

2.4 Sociedad de Agentes Reactivos

Durante la vida de la Inteligencia Artificial ha emergido un cierto número de paradigmas diferentes con resultados suficientemente interesantes como para no ser descartados, y cada caso pretende ser la metodología avanzada que merece especial atención. Los principales paradigmas pueden ser clasificados en dos grupos: *aproximaciones basadas en el conocimiento*, sustentada sobre la "hipótesis del sistema físico de símbolos" de Newell y Simon; *aproximaciones basadas en el comportamiento*, basada en la duplicación de las capacidades de procesamiento de señal y control de las que disponen los animales más simples, y sus defensores son Wilson y Brooks.

En la "hipótesis del sistema físico de símbolos" en el que establece que un sistema físico de símbolos dispone de los medios necesarios y suficientes para desarrollar una actividad general inteligente. Un sistema físico de símbolos es una máquina, tal como un computador digital, que es capaz de manipular datos simbólicos. Un aspecto importante de esta hipótesis es que no importa de qué esté hecho el sistema físico de símbolos. Aunque esta hipótesis no está universalmente aceptada, en ella se basa mucho de lo que podríamos llamar IA "clásica" [Nilsson 01].

En la siguiente sección se menciona la IA basada en el comportamiento.

2.4.1 IA Basada en el Comportamiento

En la sección anterior se ha abordado la IA basada en tareas cognoscitivas, es decir, basadas en el conocimiento, pero existen tareas referentes a problemas de percepción y acción de los cuales surge otra vertiente de investigación, abordado por investigadores en el campo de la robótica [Rich & Knight 94].

Es bajo esta vertiente sobre la que está conceptualizado el lenguaje Age2000, específicamente sobre la Arquitectura Subsumption.

El enfoque de sistemas basados en el comportamiento se centra en mostrar comportamiento inteligente en la realización de tareas, más que en la representación de los aspectos relativos al razonamiento y planificación tradicional en los que se asumen características particulares del sistema y su medio ambiente [Atkinson 98].

Este segundo grupo de aproximaciones hacia la IA también se denomina aproximaciones "subsimbólicas". Éstas siguen usualmente un estilo de diseño ascendente, es decir, comenzando en el nivel más bajo y procediendo hacia los niveles superiores.

En tal definición se conciben sistemas que se centran en estudiar el comportamiento inteligente en el mundo real, cuyo cierto número de tareas se llevan a cabo de forma autónoma.

Este enfoque se fue adoptando a principios de los años 80, cuando investigadores y centros de investigación del mundo estudiaron la forma en cómo se organizaba la inteligencia bajo este enfoque. Llegaron a la opinión en común de que los sistemas inteligentes deben ser reactivos a los aspectos dinámicos del ambiente.

Los defensores de este estilo son Wilson y Brooks, quienes señalan que para conseguir máquinas inteligentes, se tienen que seguir muchos de los pasos evolutivos que sufrieron los seres humanos. Esto es, primero se concentra en la duplicación de las capacidades de procesamiento de señal y control de las que disponen los animales más simples, por ejemplo los insectos, y ascender por la escalera evolutiva en pasos sucesivos. Esto ha llevado consigo que a corto plazo se hayan alcanzado máquinas útiles y se esperan que mediante este estilo se obtenga el substrato sobre el cual deben construirse necesariamente niveles superiores de inteligencia [Nilsson 01].

Las teorías que surgen bajo esta vertiente están enfocadas a que los robots cuenten con dispositivos que les permitan percibir el mundo, sin contar con conocimiento previo de este ambiente. Para esto, a continuación se presentan los conceptos básicos que forman parte de estas teorías.

2.4.2 Agentes Inteligentes

A continuación se presenta la definición de agente [Atkinson 98]:

"Un agente se puede ver como un ente que percibe su ambiente a través de sensores y actúa sobre él a través de efectores".

Por medio de la percepción los agentes reciben información sobre el mundo que les rodea, a través de sensores. Un sensor es todo aquello capaz de percibir un cambio en el estado de un ambiente originando la modificación en el estado de cómputo de un agente.

Entre las capacidades humanas innatas de las que hemos sido dotados para percibir nuestro entorno, se encuentran nuestros sentidos (tacto, gusto, vista, oído, olfato), y es desde aquí donde parten algunos investigadores para llegar a la interpretación de estos sentidos. Mientras que, los efectores son dispositivos que producen determinados efectos en el entorno bajo el control del agente [Russell & Norvig 96].

Para que un agente pueda ser catalogado como racional, éste debe emprender todas aquellas acciones para obtener el máximo de su medida de rendimiento, basado en la secuencia de percepciones y del conocimiento [Russell & Norvig 96]. Los agentes racionales tienen como característica la autonomía, esto es porque sus acciones dependen de su propia experiencia y del conocimiento previo aprendido [Atkinson 98]. Con esto llegamos a la definición de *Agentes Autónomos*, o también llamados *Agentes Adaptativos*, que son agentes con capacidad de adaptarse al ambiente y operar en forma continua sobre éste.

El medio donde interactúan los agentes puede ser físico (robots), sistemas viviendo en el ciberespacio, o agentes de sistemas de software (softbots). En la figura 2.6 se ilustra el diseño de un robot autónomo [Rich & Knight 94], en el que el robot (ilustrado con el círculo) interactúa con un mundo físico.

Los medios en los que interactúan los agentes se pueden clasificar de acuerdo a sus propiedades [Russell & Norvig 96]:

Accesibilidad. Un ambiente es accesible cuando un agente, por medio de sus sensores, le permite tener acceso al estado total del ambiente, detectando los aspectos importantes de éste al ejecutar una acción. Su contraparte son los ambientes no accesibles.

Determinismo. Un ambiente es determinista cuando el estado siguiente del agente se determina mediante el estado actual y por las acciones elegidas. Su contraparte son los ambientes no deterministas.

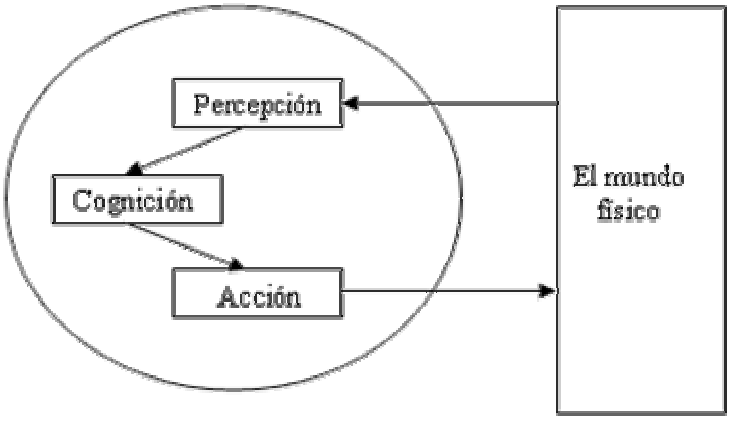


Fig. 2.6 *Diseño de un robot autónomo [Rich & Knight 94].*

Episódico. La experiencia de los agentes se divide en episodios, en donde un agente percibe y actúa. Los episodios siguientes no dependen de episodios anteriores. El agente no piensa por adelantado. Su contraparte son los ambientes no episódicos.

Dinamismo. El ambiente sufre modificaciones en momentos en que el agente delibera sus acciones, el agente considera las acciones a tomar y el paso del tiempo. Su contraparte son los ambientes estáticos.

Continuidad. Un ambiente es continuo si no existe una cantidad limitada de actuación y percepción claramente distinguibles. Su contraparte son los ambientes discretos.

Un ambiente real tiene la característica de ser no accesible, dinámico, no episódico y continuo; esto es, el peor de los casos [Atkinson 98]. Ésta es precisamente una de las razones que justifica el diseño de sistemas reactivos.

Los agentes reactivos diseñados bajo el lenguaje Age2000 actúan sobre un mundo virtual. Aunque un mundo virtual no tiene las características de un ambiente real, se puede observar el comportamiento esperado de lo agentes reaccionando en base a sus condutas.

En el lenguaje Age2000 se enlistan las conductas con las que cuenta un agente de manera jerárquica. Las conductas van decreciendo en jerarquía conforme se fueron declarando, con eso no

habrá conductas de la misma prioridad, y no se permite conductas repetidas. En el caso de existir varias conductas activas por estímulos percibidos en el ambiente es necesario coordinar las conductas.

Los métodos de coordinación tienen como objetivos principales el que sea eficiente, funcional y emergente; es decir, que el sistema reactivo alcance el objetivo con el método de coordinación, que las conductas generen las respuestas requeridas por los diseñadores, y que logren la capacidad de generar nuevos patrones a partir de la interacción de sus componentes básicos, respectivamente.

Existen básicamente dos métodos de coordinación [Atkinson 98]:

Coordinación Competitiva. Las conductas compiten por su activación y puede organizarse por prioridades (jerárquico) o por enlaces libres. Fue diseñado con el objeto de coordinar las diferentes conductas concurrentes en el sistema de control.

Coordinación Cooperativa. En ella las conductas se fusionan o suman, es decir, la respuesta global del sistema es una fusión de varias conductas que se suman vectorialmente. En este método se puede nuevas conductas que se generan.

Es necesario ahora conocer las arquitecturas desarrolladas para los sistemas reactivos, dado que, hasta este punto, hemos conocido los elementos básicos que las integran.

Para la construcción de los sistemas reactivos se han considerado los modelos y estudios arrojados por la Etología[4]. Esta ciencia ha ayudado enormemente en aspectos como diseño, operación y respuestas de los agentes.

Podemos nombrar algunas características que tienen en común las arquitecturas: el acoplamiento entre estímulos y acciones, evitan la representación de conocimiento simbólico, descomponen el sistema en conductas.

Estas arquitecturas recaen en tres grupos [Atkinson 98]:

Esquemas motores. Está formado por una red de esquemas donde cada una se comporta como si fuera un agente por sí sólo, determinando así un sistema de computación distribuido. Esto es, cada esquema percibe estímulos, los integra y genera respuestas que a su vez, en mayor o menor grado, generan la respuesta global. Su coordinación es por suma de vectores.

Redes de Conducta. También llamada Redes Compiladas. Se forman redes de conductas donde los tipos de enlaces son: sucesor, predecesor, conflicto. Lo que se busca es realizar la mejor acción para determinados estímulos, esto es formando la cadena de conductas adecuadas para ello. Cada una de las conductas posee algunas precondiciones, las cuales se cumplen sobrepasando ciertos umbrales de energía o influencia; si esto sucede, la conducta se activa. Las redes de conducta es un modelo no jerárquico.

Subsumption[5]. Esta arquitectura es de interés en este proyecto, por lo cual se desarrolla en la siguiente sección.

[4] La Etología estudia el comportamiento animal en condiciones naturales.

2.4.3 Arquitectura Subsumption

La Arquitectura Subsumption fue desarrollada por Rodney Brooks a principios de los 80, en el Laboratorio de IA en el MIT, la cual va dirigida a construir robots autónomos para navegación y manipulación en interiores. Este modelo utiliza un mecanismo de coordinación competitivo en una organización jerárquica de niveles, provee mecanismos de inhibición de entradas, supresión de salidas y otros [Atkinson 98].

En la organización jerárquica de niveles las conductas de más alto nivel representan las conductas abstractas (identificación de objetos y sus movimientos) y las de bajo nivel son las conductas básicas (comportamientos reactivos). Cada nivel se maneja independientemente, pero a la vez comunicándose con otras ya que se llegan a suprimir los niveles superiores ante los niveles inferiores – por ejemplo un comportamiento de navegación de alto nivel se suprime bruscamente cuando un obstáculo obstruye el camino de un robot [Rich & Knight 94].

En esta arquitectura los comportamientos operan de forma descentralizada y reactiva. Las principales ventajas de los sistemas reactivos es que pueden navegar en el mundo real manteniendo comportamiento complejo, no hay planes ni modelos del mundo, sólo reaccionan a las situaciones que tienen a la mano. Además son sensibles evitando la explosión combinatoria que implicaría una planificación deliberativa; estos sistemas presentan la ventaja de requerir escaso poder computacional para operar, ya que la conducta "inteligente" que pueden desempeñar emerge a partir de una programación extremadamente reducida de comportamientos sencillos [Rich & Knight 94].

Hasta el momento ya se ha hablado de sus cualidades y características de esta arquitectura, pero ¿cuáles son sus desventajas? Entre las principales se encuentran:

- Su organización jerárquica, ya que se encuentra dada bajo la apreciación del usuario, tomando en cuenta que tiene que ponderar las conductas.

- El crecimiento en el número de niveles de abstracción, esto es, una vez que el robot ha logrado alcanzar un comportamiento satisfactorio dentro del mundo, se le aumentan sus conductas en el nivel de abstracción, siendo esto aún más complejo de organizar y ponderar.

- Una topología predeterminada en tiempo de compilación.

Brooks introdujo la hipótesis de los "fundamentos físicos" en la cual expone que se puede obtener un comportamiento complejo sin usar modelos centralizados; para esto bastaría con dejar que los diversos módulos de comportamiento de un agente interactúen independientemente con el entorno. Sin embargo, Brooks acepta que para conseguir IA de nivel humano, puede ser necesaria la integración de las dos aproximaciones [Nilsson 01].

La investigación y desarrollo en el área de sistemas basados en conducta ha obtenido grandes éxitos. Cabe mencionar que los diseños basados en sistemas deliberativos tradicionales, sistemas basados en el conocimiento, no se excluyen ante los sistemas basados en conducta, estos pueden coexistir dando paso a una variedad de aplicaciones conocidos como sistemas híbridos [Atkinson 98].

[5] Algunos autores han traducido la palabra subsumption como: subducción. Para evitar caer en posibles errores de traducción, a lo largo de este texto se mencionará con su nombre en Inglés.

Capítulo 3: Compilador Age2000

3.1 Lenguaje Age2000

El lenguaje Age2000 permite la definición de una sociedad de agentes reactivos, lo cual implica que se defina un agente o más. Un agente por sí mismo, encierra cierto comportamiento, el cual implica también acciones. Lo que pretende este lenguaje es la declaración de varios agentes que convivirán en un mundo en común, y del cual se pueda observar el comportamiento definido.

El lenguaje Age2000 se divide en tres secciones, ilustrada en la Fig. 3.1, y éstas son las siguientes:

- Definición de una Sociedad de Agentes, definición de los agentes que componen la sociedad;
- Definición de Conductas, comportamiento definido por sus conductas;
- Definición de Acciones, acciones que comprende las conductas.

En la primera sección, definición de una Sociedad de Agentes, se encuentra la declaración de todos los agentes que componen la sociedad. Los agentes se declaran con sus conductas, las cuales se definen según su prioridad. La primera conducta definida es la de mayor prioridad, y en adelante la prioridad va decreciendo. Con esto se logra que no haya conductas con la misma prioridad.

Una vez definida la Sociedad de Agentes, se desarrolla la definición de la sección de Conductas. La definición de Conductas es en donde se encuentra declarado el comportamiento que corresponde a las conductas definidas por el agente en la sección anterior. Estas conductas se forman por una o más premisas de percepción y solicitudes de acción.

El lenguaje ofrece primitivas de percepción, las cuales se encuentran asociadas a dispositivos de sensado del robot. Estas primitivas están relacionadas a sensores tactiles, fotocelda, y temperatura.

La tercera y última sección de la estructura de un programa en el lenguaje Age2000 es la definición de las acciones. Las acciones contienen instrucciones formadas por estructuras de control y primitivas de acción. Las acciones son solicitadas por las distintas conductas.

Las primitivas de acción que proporciona el lenguaje son las que tienen el control del actuador del robot, tales como: avanzar hacia delante, hacia atrás, girar a la izquierda o a la derecha.

Si una acción es ejecutada o no, depende de un oráculo, ya que éste tiene la responsabilidad de asignar una sola acción de todo el conjunto de posibles solicitudes, eligiendo la de mayor prioridad. Este punto será explicado con mayor detalle en la construcción de la máquina virtual.

```
sociedad:
        agente <agente_1>:
                <conducta_1>;
                <conducta_2>;
                ...
                <conducta_n>;
        finAgente

        ...

        agente <agente_x>:
                ...
        finAgente

finSociedad

comportamiento:
        conducta <conducta_1>:
                 <instrucciones>
                 solicita <accion_1>;
                 <instrucciones>
        finConducta

        ...

        conducta <conducta_n>:
                 <instrucciones>
                 solicita <accion_m>;
                 <instrucciones>
        finConducta


finComportamiento

acciones:
        accion <accion_1>:
                 <instrucciones>
        finAccion

        ...

        accion <accion_m>:
                 <instrucciones>
        finAccion

finAcciones
```

Fig. 3.1 *Estructura de un programa en lenguaje Age2000.*

Bajo esta estructura, las acciones pueden ser utilizadas por otras conductas. Así también, las conductas, de los diferentes agentes pueden tener conductas iguales, y en tal caso, sólo se declara la conducta en cuestión. Con esto se evita repetir código.

Las estructuras de control consideradas en el lenguaje son estructuras comunes en la mayoría de los lenguajes de alto nivel, los cuales se utilizan en las secciones de Comportamiento y Acciones.

El siguiente ejemplo ilustra un programa en lenguaje Age2000, en el cual, los agentes siguen los bordes. La sociedad cuenta con dos agentes, agente1 y agente2, los cuales comparten el mismo comportamiento.

```
sociedad:

  agente agente1:
    evitar_obstaculos;
    caminar;
  finAgente

  agente agente2:
    evitar_obstaculos;
    caminar;
  finamente

finSociedad

comportamiento:

 tactil sensor_frente(SENSOR_1);

  conducta caminar:
        solicita dar_un_paso;
  finConducta

  conducta evitar_obstaculos:
      si sensor_frente > 0 entonces
         solicita vuelta;
      sino
         solicita dar_un_paso;
  finConducta

finComportamiento

acciones:

  accion dar_un_paso:
      avanza;
  finAccion

  accion vuelta:
      giraIzq;
  finAccion

finAcciones
```

3.2 Analizador Léxico

Para iniciar con la construcción del compilador, se va a partir de la construcción de su analizador léxico. Una vez conocido el lenguaje, y teniendo la idea clara de los componentes léxicos que admite el lenguaje, se procede con la definición de expresiones regulares correspondientes. En el Apéndice C se encuentran las estructuras que corresponden a la construcción del Analizador Léxico.

La definición de las expresiones regulares va a permitir la construcción del AFD para elaborar la Tabla de Transiciones.

El autómata construido es determinístico porque en las expresiones regulares no se contempla la transición de la cadena vacía, ni tampoco el que existan más de un estado próximo partiendo de un estado dado y un caracter de entrada.

Ya que la Tabla de Transiciones es reflejo del AFD, se construye colocando en las columnas los símbolos de entrada válidos por la gramática, y los renglones corresponden a cada uno de los estados del autómata.

En las expresiones regulares, y por ende el autómata y la tabla de transiciones, no se contemplan el reconocimiento de palabras reservadas. El analizador léxico reconoce una palabra (variable) a partir del autómata, esto es por medio de la siguiente expresión regular:

Palabra [a..zA..Z] [_ | [a..z A..Z 0..9]*]* [a..z A..Z 0..9]

Una vez leída la palabra, se realiza una búsqueda la cual se auxilia de la estructura de un árbol en la que se encuentran ordenadas las palabras reservadas de acuerdo a su tamaño. Si lo encuentra en ella regresa su código asociado, de otra forma regresa que ha reconocido una variable.

En caso de reconocer una variable, el analizador léxico la ingresa dentro de la Tabla de Símbolos, esto está condicionado a una revisión en donde busca si ya ha sido ingresada dicha variable. La información que contiene la tabla de símbolos es: nombre de la variable, tipo de variable, valor, uso, identificador único.

Para poder recuperarse de los errores encontrados por el Analizador Léxico, se eliminan los caracteres extraños encontrados hasta hallar uno válido, almacenando estas entradas de símbolos inválidas en la Tabla de Errores, en donde también se guarda el renglón en donde fueron halladas. Esta estrategia es la recuperación en "modo de pánico". Es importante la recuperación de errores, ya que permite al Analizador Sintáctico no enterarse de tales hallazgos y permite continuar el análisis para hallar otros errores posibles.

Hasta este punto, se ha presentado la metodología de compiladores bajo la cual está construido el analizador léxico. Falta por definir la implantación bajo la cual está realizada.

El paradigma adoptado para esta implantación es programación orientada a objetos (POO). A continuación se presenta el diagrama de clases bajo el cual está construido el analizador léxico. En el Apéndice F se encuentra el modelado del analizador léxico.

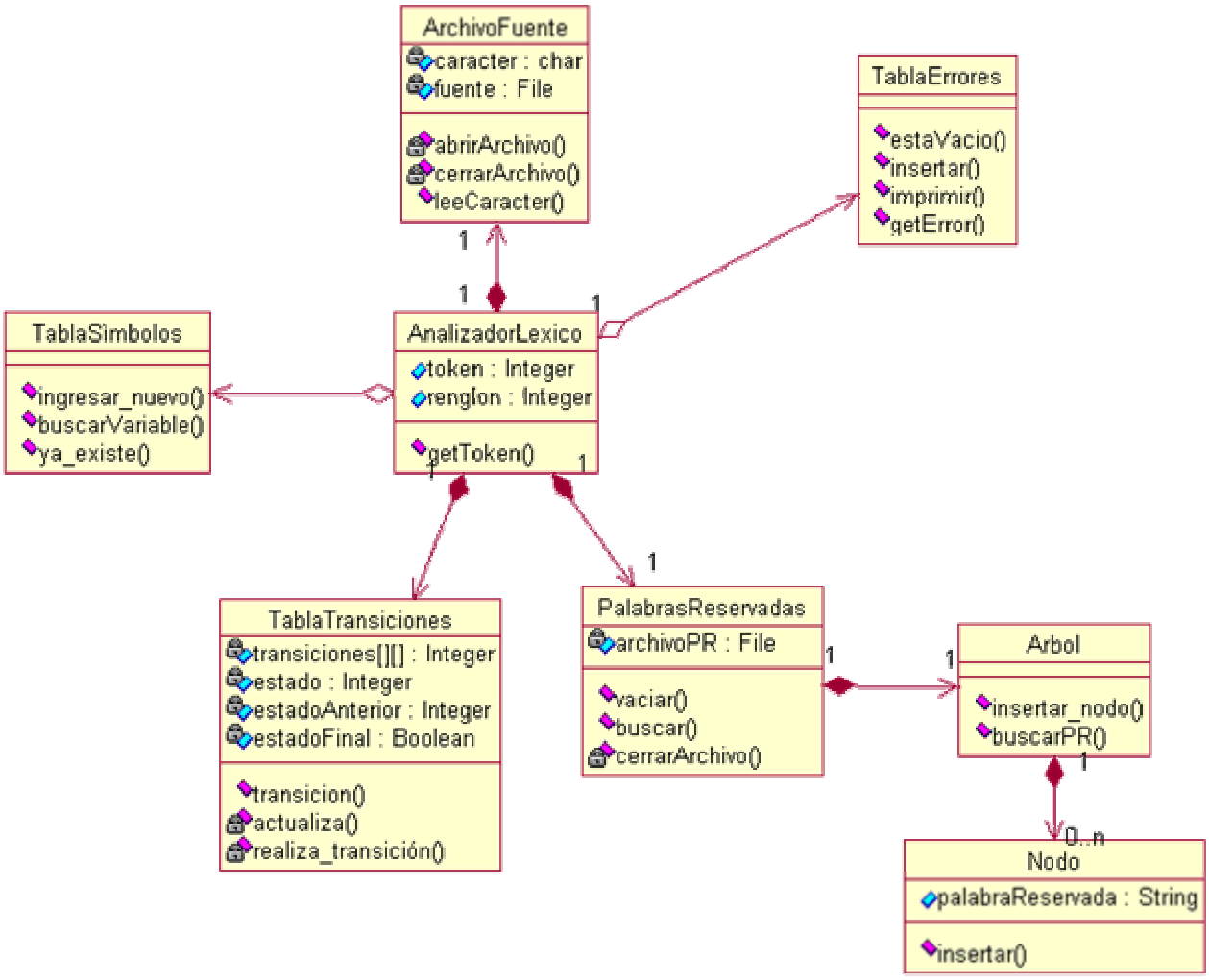


Fig. 3.2 *Diagrama de clases del Analizador Léxico.*

3.3 Analizador Sintáctico

El Analizador Sintáctico es una de las fases del compilador más complejas. En él recae la responsabilidad de reconocer un programa válido por el lenguaje.

Como ya se ha mencionado, el método bajo el cual se construye el analizador sintáctico es el Análisis Sintáctico Predictivo.

El Análisis Sintáctico Predictivo hace uso de gramáticas LL, es por esto que el lenguaje Age2000 se construye con base a las características de esta gramática.

La gramática LL que se presenta a continuación, sólo es de la sección de definición de una Sociedad de Agentes. La gramática LL del lenguaje Age2000 completa se encuentra en el Apéndice D.

G (T, N, P, S)

T = {":", ";", sociedad, finSociedad, agente, finAgente, var_conducta, EofSym}

N = {<programaAge>, <inicio>, <sociedad>, <agentes>, <agente>, <lista_robot>, <lista_conductas>, <alm_conducta>}

P = {

<programaAge>	→	<inicio> EofSym
<inicio>[6]	→	<sociedad>
<sociedad>	→	sociedad : <agentes> finSociedad
<agentes>	→	<agente> { <agente> }
<agente>	→	agente <lista_robot>: <lista_conductas> finAgente
<lista_robot>	→	var_robot
<lista_conductas>	→	<alm_conducta> { <alm_conducta> }
<alm_conducta>	→	var_conducta;

}

S = {< programaAge>}

Esta gramática tiene la característica de no ser ambigua. Aunque no existe una técnica formal para comprobar la ambigüedad, esto se puede realizar creando ejemplos de programas y a partir de ellos crear sus árboles sintácticos. De no ser ambigua, no habrá ningún caso en el que haya dos caminos para resolver la misma entrada. Se puede observar que las producciones están definidas de tal manera que no existe la oportunidad para que existan dos soluciones a una entrada.

Además, también cuenta con la característica de no tener recursividad por la izquierda. Se puede observar en la gramática del lenguaje que no existe ninguna producción en la que su símbolo inicial de la producción sea el mismo no terminal que la produce. Es decir, no hay ninguna producción de la forma:

$A \rightarrow A\ \alpha$,

donde A es un símbolo no terminal y α es una cadena de símbolos terminales y/o no terminales.

Esta gramática, además, tiene muy bien definido el camino a tomar al realizar alguna derivación. Ésta es una característica propia del método de Análisis Sintáctico Predictivo. Ya que no existen retrocesos, lo cual significa, que no hay duda de la producción a expandir en algún momento dado.

[6] La producción <inicio> ha sido modificada sólo con propósitos de ejemplificación.

En la construcción de la gramática se ha tenido cuidado de no contar con dos producciones que presenten esta clase de conflicto, y si en algún momento lo hubo, se resolvió con el uso de la factorización para que siga existiendo sólo un camino que elegir con una entrada dada.

La gramática, como bien lo especifica LL(1), se lee de izquierda a derecha, y con derivaciones por la izquierda. La lectura es de un sólo componente léxico a la vez. A continuación se presenta un ejemplo de lo anterior, aunque sólo es con la sección de una sociedad de agentes:

Entrada:

```
sociedad:

      agente robot:
            evitar;
            seguir;
      finAgente

finSociedad
```

La entrada consiste en la declaración de una sociedad formada por un solo agente llamado "robot", el cual tiene dos conductas: "evitar" y "seguir".

La derivación se lleva de la siguiente manera:

Producción	**Token actual**	**Cadena**
<programaAge>	sociedad	
<inicio>	sociedad	
<sociedad>	sociedad	
<sociedad>	:	sociedad
<sociedad>	agente	
<agentes>	agente	sociedad :
<agente>	agente	sociedad :
<agente>	robot	sociedad : agente
<lista_robot>	robot	sociedad : agente
<agente>	:	sociedad : agente robot
<agente>	evitar	sociedad : agente robot :
<lista_conductas>	evitar	sociedad : agente robot :
<alm_conducta>	evitar	sociedad : agente robot :
<alm_conducta>	;	sociedad : agente robot : evitar
<lista_conductas>	seguir	sociedad : agente robot : evitar ;
<alm_conducta>	seguir	sociedad : agente robot : evitar ;
<alm_conducta>	;	sociedad : agente robot : evitar ; seguir
<lista_conductas>	finAgente	sociedad : agente robot : evitar ; seguir ;
<agente>	finAgente	sociedad : agente robot : evitar ; seguir ;
<sociedad>	finSociedad	sociedad : agente robot : evitar ; seguir ; finAgente
<sociedad>		sociedad : agente robot : evitar ; seguir ; finAgente finSociedad

Del ejemplo anterior, se tiene el siguiente árbol sintáctico:

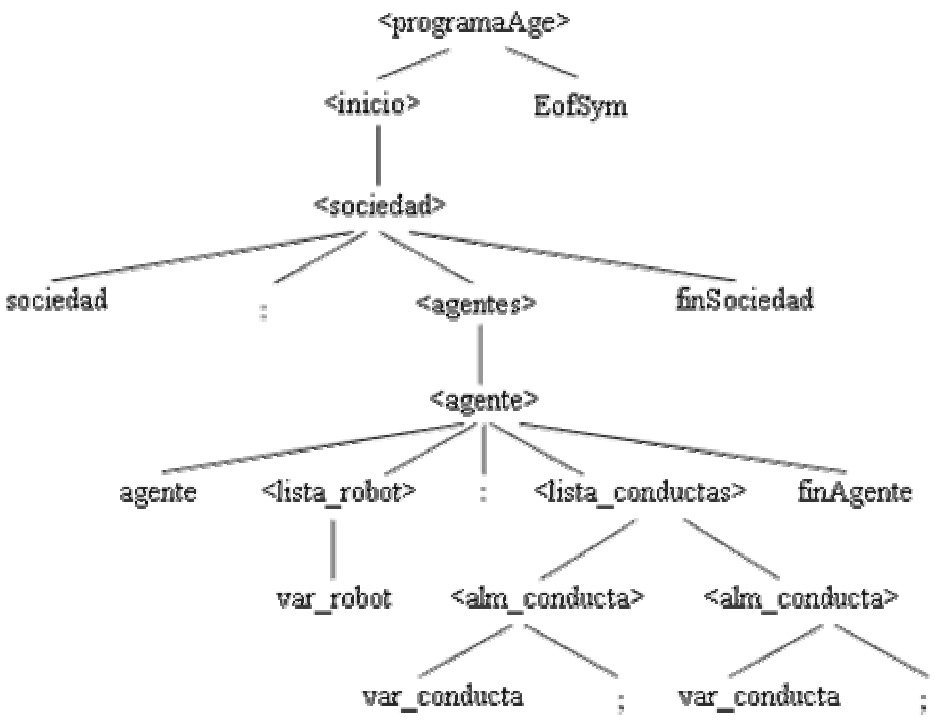


Fig. 3.3 *Árbol sintáctico de una sociedad de un solo agente*.

Como los métodos de Análisis Descendente Recursivo, el procedimiento para implantarlo es asociar a cada no terminal un procedimiento. Así que, cuando se analiza una entrada, se inicia el análisis con el símbolo inicial que es un símbolo no terminal el cual tiene asociado un procedimiento, y continúa ingresando a las reglas de producción que generen la entrada, ya sea en comparación directa del componente léxico esperado con el leído, o entrando a los procedimientos que hacen referencia a los símbolos no terminales.

La manera en que se construye el analizador sintáctico es bajo el paradigma de POO. Es decir, cada uno de los símbolos terminales de la gramática está asociado a una clase. El funcionamiento en general es que un objeto, instancia de una clase (símbolos no terminales), llama a otras objetos contenidos, hasta llegar a los objetos que contienen el caracter que se analiza (símbolos terminales). En seguida, se presenta una estructura general para el símbolo no terminal <sociedad>, el cual está contenido en el archivo llamado "Sociedad.java".

```
public class Sociedad{

    public Sociedad(){

    }

    public void analisis() throws Exception{
      try{
```

```
            Agentes agentes= new Agentes(sintactico);
            sintactico.comparar("sociedad");
            sintactico.comparar(':');
            agentes.analisis();
            sintactico.comparar("finSociedad");
        }
    }
```

Y la manera de manejar los errores es por medio de clases derivadas de excepciones, almacenando hallazgos de errores en una clase creada llamada Tabla de Errores. El modo de recuperación adoptado es recuperación en modo de pánico, en donde se han especificado como símbolos de sincronización: al "punto y coma", "coma", palabras reservadas que inician y terminan las secciones (como "agente", y "finAgente").

El trato de la Tabla de Símbolos es necesaria ya que se realizan actualizaciones en los tipos de variables halladas, así como de sus valores iniciales.

Existe la clase *AnalizadorSintactico* la cual es responsable de llevar el control de principio a fin del recorrido que se realiza con una entrada.

La construcción del analizador sintáctico bajo el método de Análisis Descendente Predictivo, es un método que facilita la implantación de analizadores sintácticos de forma manual, obtiene resultados eficientes (el lenguaje Age2000 cuenta con estructuras sencillas, que no dificultaron la construcción de su gramática LL). La implantación por medio de clases, fue una forma nueva de ver el método, de observar una nueva forma de comportamiento, de usar estructuras pertenecientes a objetos para implantar cada una de las partes que componen al analizador.

3.4 Analizador Semántico

El Analizador Semántico se encargará primordialmente de verificar la coherencia entre los tipos de datos que espera el Analizador Sintáctico. Cuando el Analizador Sintáctico espere una variable de un tipo específico, entonces el Analizador Semántico hace uso de la Tabla de Símbolos ubicando la variable y obteniendo el tipo. Así podrá detectar incoherencias entre los tipos de variables esperadas.

El analizador semántico cuenta con diversas funciones para realizar la validación de la expresión. Las acciones que realiza con apoyo de la Tabla de Símbolos son: actualizar el uso de la variable y verificar el uso correcto de los valores de las variables y sus alcances. En el análisis reporta errores encontrados por los tipos de variables, y también da avisos (warnings) de declaraciones de variables que no hayan sido utilizadas.

3.5 Generación de Código

Para cada sociedad de agentes, se generan como mínimo 4 archivos, y esto es por lo siguiente:

Un archivo generado tiene los sensores que utilizan los agentes; contiene la tabla de variables de tipo entero y decimales declaradas como globales y tiene la relación de los agentes con sus conductas. El archivo contendrá lo siguiente:

Sociedad.Age2000.obj

```
?
 & "nombre_sensor_1" id tipo
 ...
 & "nombre_sensor_x" id tipo
?

@
 $ "nombre_variable_1" id tipo #valor#
 ...
 $ "nombre_variable_y" id tipo #valor#
@

\ nombre_agente_1 ^ nombre_conducta_1 ^ ... ^ nombre_condcuta_n
\ ...
\ nombre_agente_m ^ nombre_conducta_1 ^ ... ^ nombre_condcuta_n
```

Fig. 3.4 *Archivo generado para reconocer la sociedad.*

Ya que las conductas tienen llamados a ciertas acciones, en seguida se crea un archivo donde se especifica cada relación. Este archivo también incluye la relación de las acciones asociado a su id de la tabla de las acciones.

Sociedad.Conductas.Age2000.obj

```
~
 * "nombre_accion_1" id
 ...
 * "nombre_accion_w" id
~
\ nombre_conducta_1 ^ nombre_accion_1 ^ ... ^ nombre_accion_m
\ ...
\ nombre_conducta_n ^ nombre_accion_n1 ^ ... ^ nombre_accion_nm
```

Fig. 3.5 R*elación de las conductas con sus acciones.*

Se crea para cada conducta y para cada acción declarada se crea un archivo. Por lo tanto, el mínimo número de archivos generados es de 4.

Los archivos con el código objeto de las conductas y acciones llevarán como cabecera la tabla de símbolos correspondiente a la declaración de variables de tipo entero y decimales locales. Definir el resto de la estructura de los archivos es impreciso, ya que depende de las instrucciones

que se definan en un programa específico, como las estructuras de control, solicitudes de acción, instrucciones de primitivas o instrucciones de asignación. Esto se muestra de la siguiente manera:

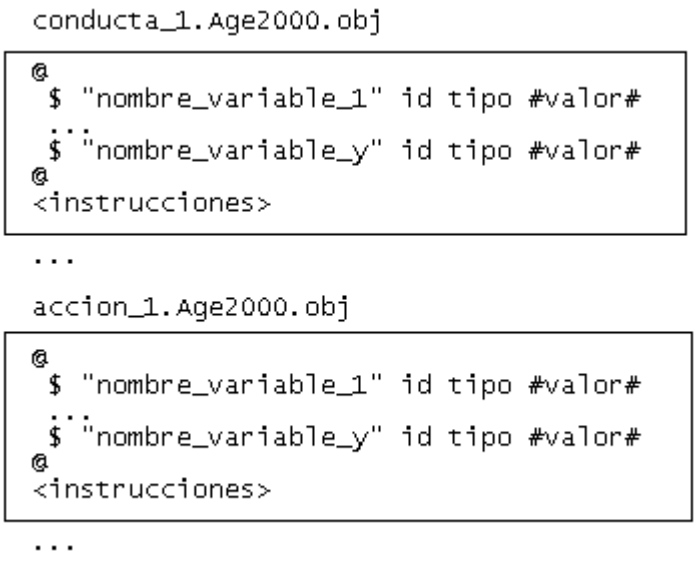


Fig. 3.6 *Archivos de conductas y acciones.*

Para la identificación de los números, se inicia y termina el número con el caracter "#", quedando de la siguiente manera: # <número> #; por ejemplo: #33.2#

El trato de las variables de los diferentes tipos cumplen con el patrón de iniciar con un símbolo que identifique su tipo (& - sensores, $ - decimal/entero), enseguida, se especifica el nombre de la variable entre comillas, el "id" que es un identificador único que tienen todas las variables, y finalmente se especifica el tipo de variable. En caso de los enteros y decimales, también se especifica el valor de la variable entre el caracteres "#", como se expuso anteriormente.

Para el caso de las estructuras de control, se asocian a un carácter en ASCII que al leerlo la identifica. Entonces, se inicia con ese identificador, enseguida, se coloca las condiciones entre corchetes y finalmente el bloque de instrucciones entre llaves. Esto es, Estructura de Control [<condicion>]{<bloque de instrucciones>}.

El código binario completo, correspondiente a las instrucciones empleadas, se encuentra en el Apéndice E.

Como se ha podido observar, el código objeto generado es de tipo código objeto intermedio, ya que se mantiene el nombre del archivo intacto sin ninguna traducción a un código especial. Esto es, para preservar los nombres de agentes, conductas y acciones utilizadas en el programa, y si alguna persona está interesada en la construcción de dicho código, se facilite entender su formato.

Capítulo 4: Máquina Virtual Age2000

Una vez que el compilador produce el código objeto, es necesario contar con una Máquina Virtual para poder ejecutar dicha aplicación. Dado que el código objeto generado no es código máquina se necesita de lo que se llama máquina virtual para poder interpretarlo.

Este proceso se divide en dos partes. La primera se enfoca a la sincronización de la conducta de los agentes, esto es, la forma en la que internamente se refleja la Arquitectura Subsumption. La segunda parte se enfoca a la construcción del mundo en el que se desenvolverá la sociedad de agentes; ya que se puede observar el comportamiento de una sociedad en más de un mundo. También, la construcción de un mundo puede servir para la ejecución de más de una sociedad de agentes distinta, esta parte se desarrollará en el siguiente capítulo.

4.1 Sincronización de conductas

A la vista del usuario, sólo se verá el comportamiento de los agentes de acuerdo a la arquitectura Subsumption. Internamente, la máquina virtual provee de un mecanismo de control la cual sincroniza las conductas. La arquitectura que se describe a continuación corresponde a la implementación interna del diseño del comportamiento del agente.

Para esto, se contempla la creación de un agente "oráculo", el cual evalúa las peticiones de acción de las conductas; y un agente “actor”, que lleva a cabo las acciones.

Para la ejecución de las conductas, el agente “oráculo” y el agente “actor” trabajan en forma concurrente como lo muestra la Fig. 4.1.

Fig. 4.1 *Componentes de sincronización [Acosta & Fernández 00].*

En caso de que haya competencia por parte de las conductas para lograr su activación, el agente "oráculo" tiene la responsabilidad de decidir cuál de ellas es a la que atenderá, mostrado en la Fig. 4.2.

Fig. 4.2 *Competencias de las conductas por lograr su activación [Acosta & Fernández 00].*

El agente "oráculo" debe decidir la conducta que ganará de acuerdo a su prioridad. Por esto mismo, se ha destacado que el orden en el que se listan las conductas es de especial cuidado por el usuario, ya que él definirá las prioridades de las conductas en forma decreciente conforme se vayan declarando las conductas del agente. De lo anterior se deduce que las conductas cuentan con una prioridad fija y no hay valores de prioridad iguales.

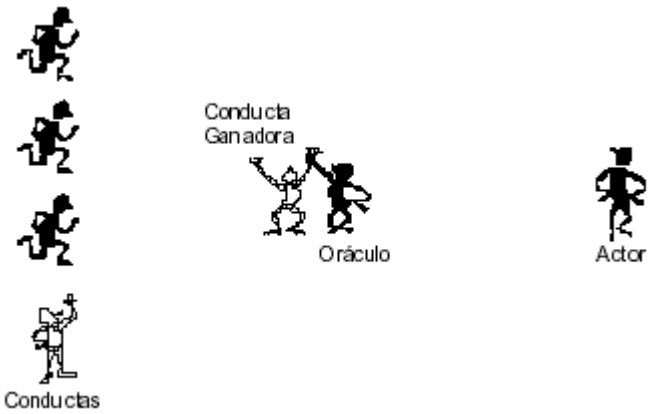


Fig. 4.3 Selección de c*onducta ganadora [Acosta & Fernández 00].*

Una vez decidida la conducta a ejecutar por parte del agente "oráculo", éste hará la petición al agente "actor" que se encargará de llevar a cabo las acciones de la conducta ganadora.

Fig. 4.4 *Ejecución de las acciones del "actor" [Acosta & Fernández 00].*

El agente "actor" ejecutará la acción hasta terminarla. En caso de que el agente "oráculo" le indique al agente "actor" la ejecución de otra acción, entonces el actor cancelará en su totalidad la ejecución actual, sin posibilidad de continuarla en otro momento y tomará la nueva petición del agente "oráculo".

El oráculo, como ya se ha mencionado, sólo actúa como un mecanismo de control para la resolución de conflictos entre las conductas de los agentes. Cada agente cuenta con su propio oráculo mostrando autonomía entre ellos, ya no existe un controlador de los agentes de la sociedad, así que no existe ningún objeto que en dado momento pueda resolver conflictos entre los agentes. El lenguaje Age2000 fue pensado para ir creciendo dependiendo de las necesidades que se fueran presentando, y no ha incluido este tipo de resolución de conflictos.

Hasta este punto, se ha dado la concepción bajo la cual se rige la ejecución de las acciones que realizará un agente dentro de una sociedad declarada en el lenguaje Age2000. A continuación se plantea la forma en que se encuentra realizada la implementación de esta concepción para lograr que los agentes actúen de este modo.

4.2 Implementación de la sincronización

La concepción bajo la cual es realizada la implementación es bajo POO, es por esto que se piensa en la creación de clases para plasmar el comportamiento de los agentes en un mundo virtual.

Como ya se ha mencionado, el compilador genera una serie de archivos, con extensión "obj", como resultado de una compilación exitosa, esto es, un programa en lenguaje Age2000 que no contiene errores.

Ahora toca el turno a la Máquina Virtual el tomar esos archivos y crear la sociedad declarada por parte del programador. Se ilustra en la Figura 4.5 el diagrama de clases que contiene el paquete creado para la Máquina Virtual.

Como primer paso, la Máquina Virtual lee el archivo objeto que contiene la información de la cantidad de agentes que contiene la sociedad, así como la relación con sus conductas y variables globales. Este archivo objeto se distingue porque su nombre está formado como sigue: el nombre del programa fuente seguido de la terminación ".Sociedad.Age2000.obj".

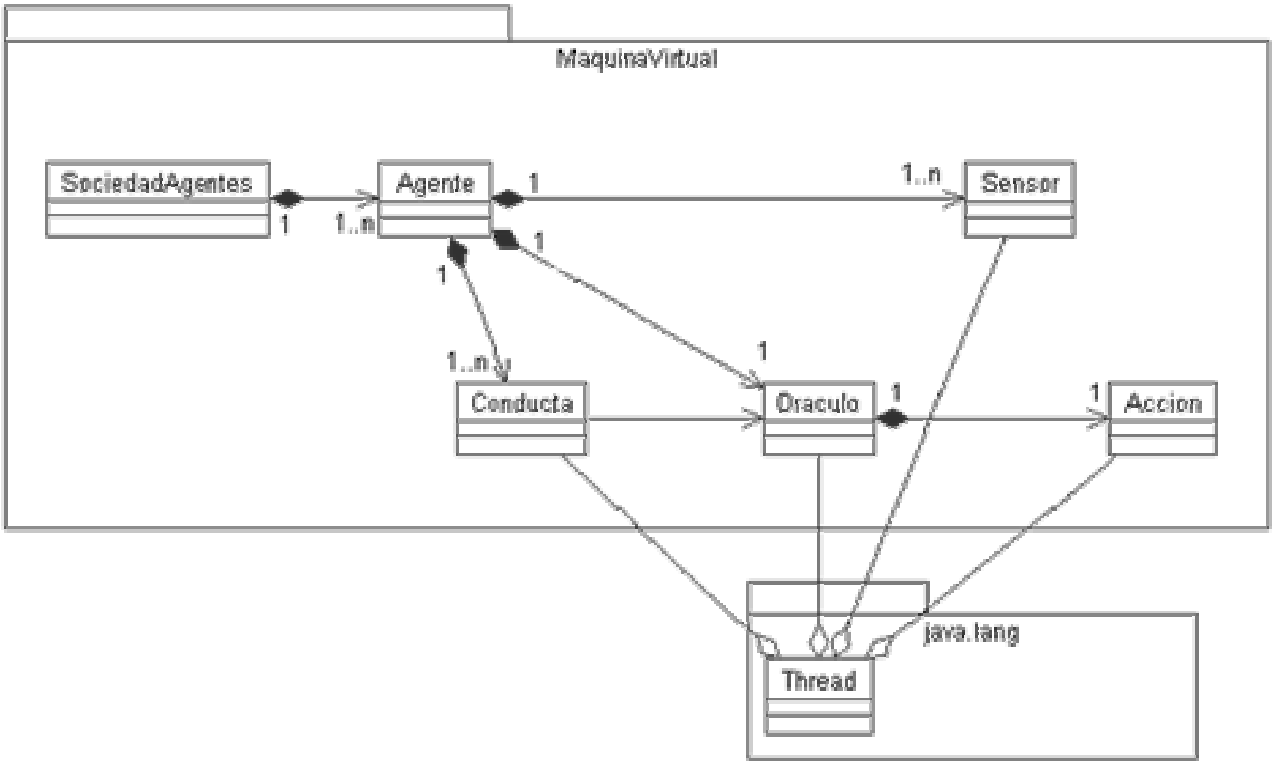


Fig. 4.5 *Clases contenidas dentro del paquete MaquinaVirtual.*

Un objeto de la clase *SociedadAgentes*, tiene como tarea iniciar a los agentes con sus conductas y acciones permitidas. De esta acción, se obtiene la creación de los objetos de tipo Agente de acuerdo al número de agentes declarados. La clase *SociedadAgentes* cuenta con una variable llamada "agente", que es un arreglo de tipo Vector, cuyo cargo es el de tener control sobre los objetos generados de la clase tipo Agente.

Los objetos de tipo Agente, a su vez, están compuestos de sensores, conductas, y un oráculo. Los sensores tienen como tarea percibir de acuerdo al tipo de sensor al que pertenece, el estado en el que se encuentra el mundo, guardan el resultado de cada sensado en una tabla de sensores, la cual es inicializada por el objeto de la clase *SociedadAgentes* y es compartida por las conductas.

La clase *Sensor* es una clase derivada de la clase *Thread*, del paquete java.lang. Un objeto de la clase *Thread* es un hilo de ejecución en un programa, esto es, especifica una porción del

programa que se puede ejecutar en paralelo con otros hilos [Deitel & Deitel 98]. Con esto se logra que hablemos de un programa que simule características concurrentes[7].

Con esta cualidad de la clase *Thread*, los objetos creados de tipo *Sensor* tienen destinado una porción del programa que realiza el sensado del mundo, en el que se desenvuelve el agente al que pertenece, de manera permanente.

Al igual que la clase *Sensor*, la clase *Conducta* deriva de la clase *Thread*. Con la característica de multihilos se designa una porción del programa la cual es la que se ejecuta en paralelo. Para los objetos de tipo *Conducta*, esa porción del programa se destina a la lectura del código que corresponde a cada conducta declarada para el agente, el cual contiene estructuras de control, expresiones de sensado, instrucciones de asignación, y también las instrucciones de solicitudes de acción.

Existen otras dos clases más que se derivan de la clase *Thread*: *Oraculo*, y *Accion*. La función del oráculo, bajo la concepción de la Arquitectura Subsumption, consiste en decidir la ejecución de alguna acción de acuerdo a las prioridades de las conductas que lo solicitan. Por esto, existe un oráculo para cada agente que se declare, para poder controlar y decidir sobre las acciones que deberá seguir el agente de acuerdo a las conductas que competen exclusivamente al agente que lo creó. La clase *Oráculo* ha sido construida bajo la concepción de un hilo ya que verifica constantemente las peticiones de acción por parte de las conductas del agente, y una vez que existan peticiones de acción entonces decidirá la ejecución de una de ellas.

Por esto mismo, un objeto de tipo *Oraculo* está compuesto de un objeto de tipo *Accion* ya que el oráculo realiza la petición de ejecución de una acción sin que éste último tenga alguna intervención en esta decisión. El objeto de tipo *Accion* inicia su ejecución cuando el oráculo así lo disponga, y así mismo terminará su ejecución. Una vez hecha la petición por parte del oráculo a realizar una acción, el objeto de la clase *Accion* debe leer el código objeto que corresponda a la acción que debe ejecutar.

Como se puede observar en la figura, existe una relación entre las clases de tipo *SociedadAgentes* y *Agente* con la clase *Suelo*. La clase Suelo es derivada de la clase *Canvas*, del paquete java.awt, la cual crea un lienzo, esto es, un componente en el que se pueden dibujar gráficos y en el que se reciben eventos del ratón; teniendo éste su propio contexto gráfico.

La relación específica de la clase *SociedadAgentes* con Suelo consiste en otorgarle al objeto de tipo *Suelo* la información sobre el número de agentes que se visualizarán en él. A su vez, la relación con la clase tipo *Agente* es para poder reflejar el comportamiento de cada uno de los agentes dentro del suelo.

Existe también una clase llmada *SueloCanvas*, es la única clase dedicada para controlar el espacio gráfico que refleje la simulación del comportamiento de los agentes. Esto significa que, la representación de los agentes en el suelo, las paredes, los obstáculos, el suelo mismo, etc, así como el cambio de orientación o los movimientos que realice el agente será labor del objeto de tipo

[7] No se puede hablar de concurrencia real, ya que solamente se cuenta con un solo procesador para la implementación de multihilos. Además, el trato de los hilos es diferente en las plataformas, por ejemplo: en las implementaciones de 32 bits de Java para Windows, los hilos se manejan por porciones de tiempo, esto significa que a cada hilo se le concede una cantidad limitada de tiempo para ejecutarse en el procesador, y cuando ese tiempo ha transcurrido el hilo tiene que esperar su turno.

SueloCanvas el ilustrarlo correctamente en su espacio gráfico. No se encuentra expresado en la figura anterior, ya que no es una clase que intervenga en la simulación, sólo es una clase para mostrar el espacio gráfico de simulación.

Dentro de la herramienta InAge se dedica un espacio para la presentación de los gráficos que reflejen el comportamiento de la sociedad de agentes declarada por el programador, y este último podrá con libertad crear mundos virtuales para observar el comportamiento de una misma sociedad en diferentes mundos virtuales. La creación de estos mundos se presenta en el siguiente capítulo.

Capítulo 5: El Entorno Integrado de Desarrollo InAge

El software llamado InAge, es la herramienta que apoya en el desarrollo de edición de nuevos programas en lenguaje Age2000, compila dichos programas, proporciona un ambiente amigable para la creación del mundo en el que se desenvuelven los agentes y simula su comportamiento. El nombre InAge surge de un compuesto de dos palabras: "Integral" que con ella se interpreta que contiene toda la funcionalidad requerida para la elaboración y simulación de programas en lenguaje Age2000, y la palabra "Age" que es la extensión de los archivos en lenguaje Age2000 que denota "agente". La implantación del software InAge está realizada mediante un lenguaje convencional de programación, Java, con la herramienta JBuilderX.

5.1 Caso de Uso InAge

La herramienta InAge comprende principalmente dos partes: edición, simulación. En la parte de edición, la herramienta cumple las características siguientes:

- Es una herramienta de edición de programas en lenguaje Age2000.
- Es una herramienta que incorpora un compilador del lenguaje Age2000 que permite visualizar mensajes de advertencia y error.

En la parte de simulación, la herramienta cumple las características siguientes:

- Es una herramienta que facilita la construcción de mundos virtuales.
- Es una herramienta que permite la simulación del comportamiento de los agentes.

El caso de uso construido a partir de los requisitos de la herramienta InAge se muestra en la Fig. 5.1.

5.2 Editor

La herramienta está elaborada bajo el paradigma de POO. El editor consta principalmente de cuatro clases: *Pantalla*, *Editor*, *Programa* y *Simulador*.

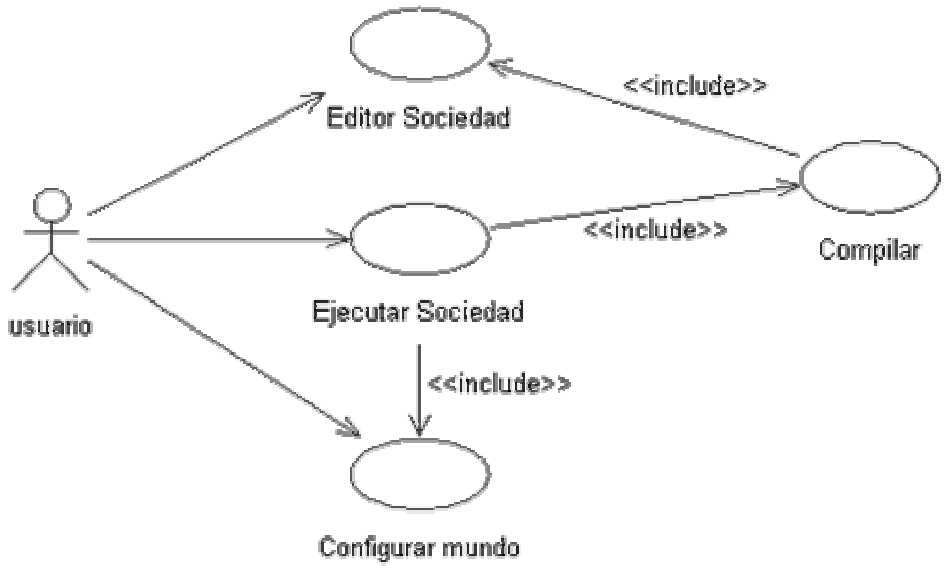


Fig. 5.1 *Caso de Uso de la herramienta InAge.*

La clase *Pantalla*, hereda de la clase *JFrame*, contiene las clases *Editor*, *Programa* y *Simulador*, las cuales se derivan de clase *JPanel*. *Pantalla* implementa el control de los botones y del menú del sistema, es decir, los eventos ya sean de edición, compilación o simulación los tiene comprendidos la clase *Pantalla*, y éste los a canaliza *Editor*, y éste a su vez al *Programa*, en caso de tratarse de la edición de algún programa o canaliza la acción a *Simulador*, en caso de tratarse de la máquina virtual.

El *Editor* está estrechamente relacionado con la clase *Programa*. *Programa* es el espacio de edición de los programas al que se le ha incluido características como la enumeración de los renglones, un color distintivo para ciertas palabras reservadas y comentarios, la localización ene l renglón en caso de un error en la compilación, la actualización del estado en el que se encuentra el sistema, entre otras tareas.

Cabe destacar que el manejo de los colores de ciertas palabras reservadas es por medio de un archivo llamado "colorPR.dat", el cual contiene una lista de las palabras reservadas que se quieren resaltar así como el color asociado de una lista de colores permitidos en el sistema. Al arrancar el sistema, se crea un árbol binario basado en la longitud de la palabra reservada para que en el momento de la escritura cambie de color según sea el caso.

Este archivo se puede editar logrando aumentar o disminuir el número de palabras reservadas que se deseen destacar al momento de la edición, sin hacer ningún cambio en el código del programa de InAge.

El editor implementa funcionalidades básicas que permiten la elaboración de programas en Age2000 de una manera amigable.

En la barra de herramientas se pueden destacar dos funcionalidades importantes: la compilación de un programa y la simulación de éste; esto mediante el menú “Compilar”, y el menú “Máquina Virtual”, respectivamente.

Al iniciarse el sistema, se muestra la pantalla con los comandos deshabilitados, sólo con la capacidad de abrir un archivo o iniciar uno nuevo.

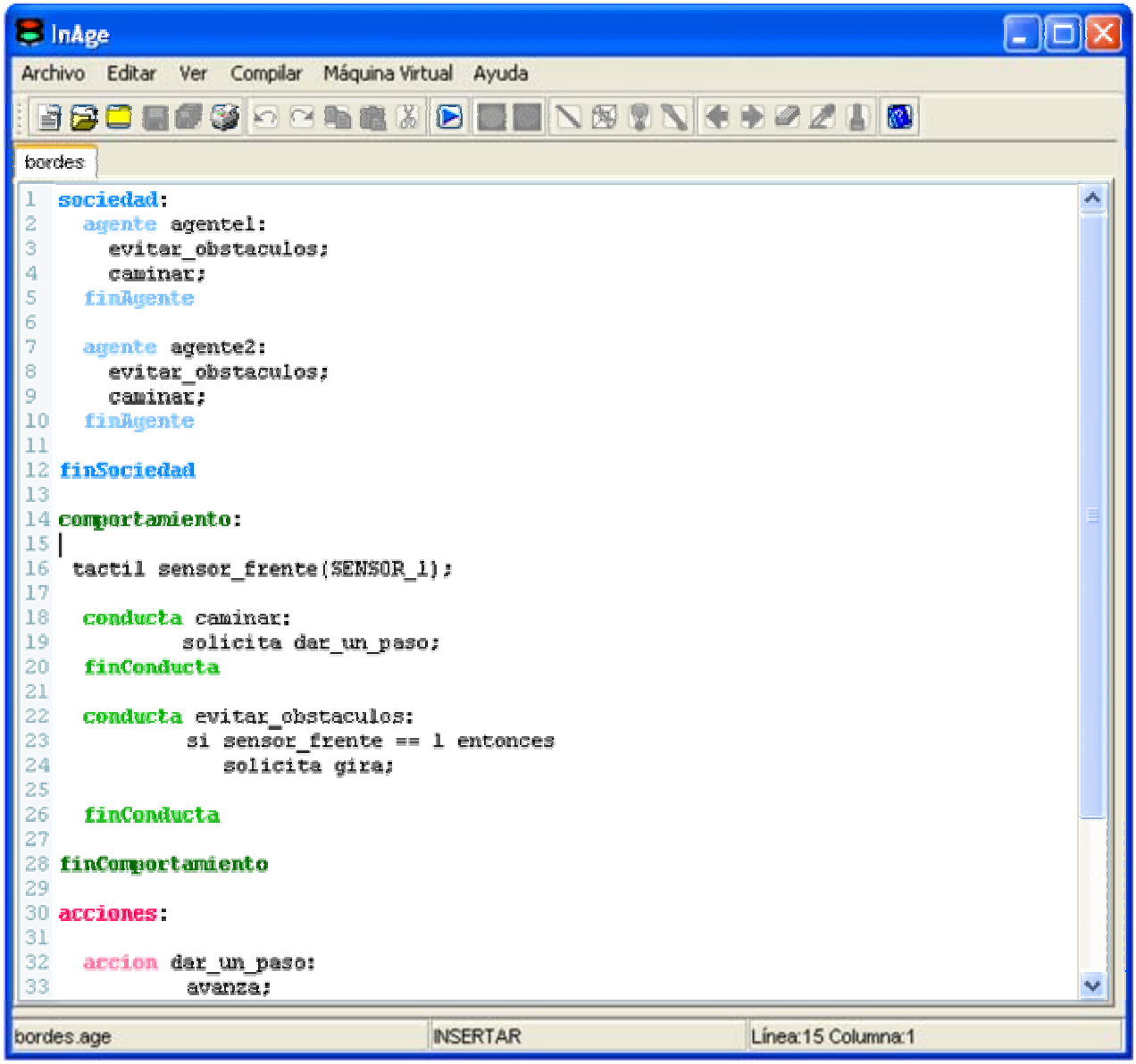


Fig. 5.2 Sis*tema InAge, con el archivo “bordes.age”.*

Al compilar un programa en lenguaje Age2000, en una ventana adicional se muestran los mensajes de error o advertencia. Si el programa está libre de errores, se generan los archivos "obj", y se puede pasar a la simulación de dicho programa.

5.3 Mundo Virtual

El mundo virtual que se construye para observar el comportamiento de los agentes, es un mundo delimitado por una matriz bidimensional de celdas a la que se llama "suelo". Cada celda puede contener objetos que tienen diversas propiedades, y pueden existir "paredes" delimitando conjuntos de celdas. Los agentes están confinados en el suelo y pueden moverse de celda a celda.

Una vez que se tiene el programa compilado exitosamente, se puede simular inmediatamente; pero se tiene una mejor apreciación de un comportamiento más complejo si se construye un mundo virtual sobre el suelo.

La herramienta InAge cuenta con una barra de herramientas de objetos que se pueden colocar sobre el suelo, los cuales son: paredes, obstáculos, un foco (dotar de iluminación), un termómetro (dotar de calor). La barra se muestra en la figura 5.3.

Fig. 5.3 *Barra de herramientas para la elaboración del mundo virtual.*

Las paredes se pueden colocar delimitando el contorno de alguna celda, o bien, atravesando la celda en forma diagonal.

Para iniciar la elaboración de un mundo, entrar al menú "Abrir Simulador", y con este comando se inicia la carga del suelo con las paredes de la periferia, como se muestra en la Figura 5.4.

Haciendo uso de la barra de herramientas, se realiza el mundo virtual bajo las condiciones que el programador desee, y una vez terminado, se puede guardar el mundo virtual en un archivo con la extensión "mdo", el cual contiene el código representativo de los gráficos en el suelo. En la Figura 5.5, se muestra en el suelo un mundo virtual en el que se desenvolverán dos agentes.

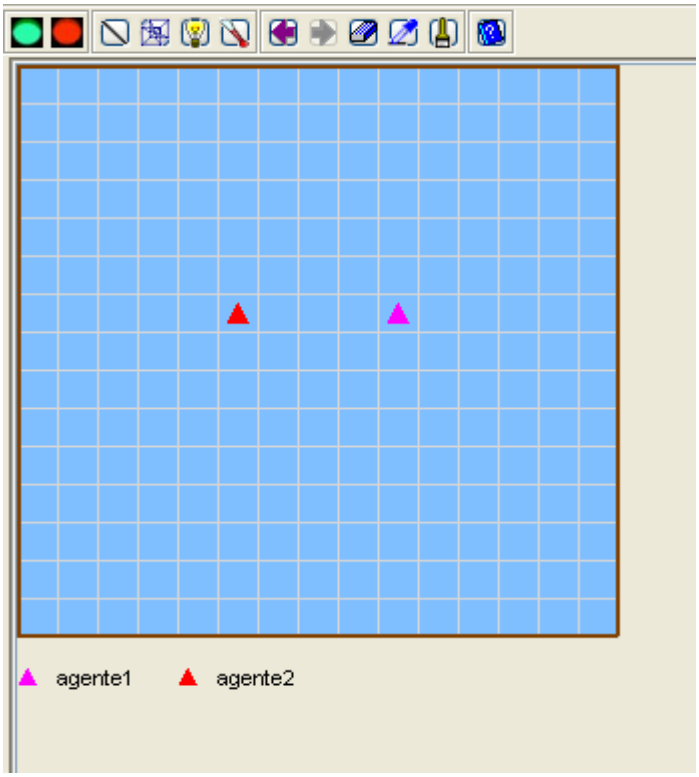


Fig. 5.4 *Suelo, con las herramientas y acciones.*

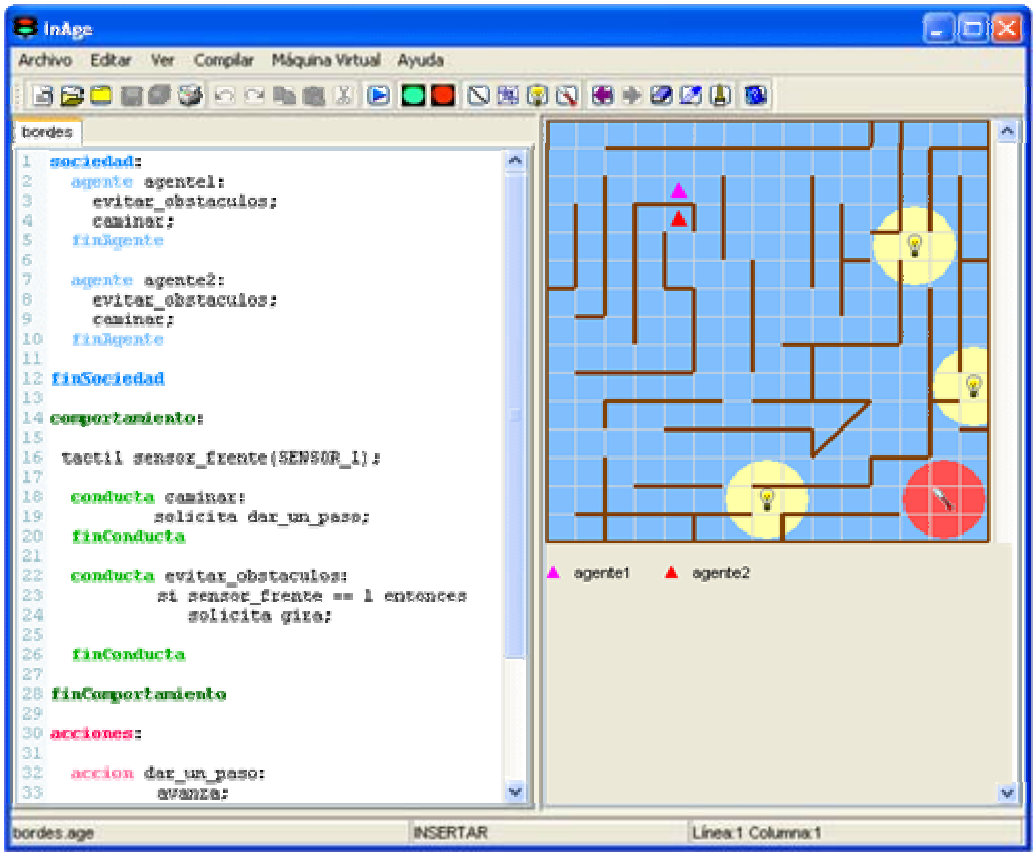


Fig. 5.5 *Programa "bordes.age" en la herramienta InAge.*

5.4 Pruebas

En la etapa de pruebas se muestra la funcionalidad de la herramienta, para ello se realizaron varios programas en lenguaje Age2000.

A continuación se muestra el código de un programa en Age2000 en el que actúan dos agente en un mundo virtual en busca de luz, y una vez hallada detienen sus acciones. El programa ha sido llamado "busca_luz.age".

```
sociedad:

   agente agente1:
      caminar;
      evitar_obstaculos;
   finAgente

   agente agente2:
      caminar;
      evitar_obstaculos;
   finAgente


finSociedad

comportamiento:

tactil sensor_frente(SENSOR_1);
luz sensor_luz(SENSOR_2);

  conducta caminar:
      si sensor_luz == 0 entonces
          solicita dar_un_paso;
  finConducta

  conducta evitar_obstaculos:
      si sensor_frente == 1 entonces
         solicita vuelta;
  finConducta

finComportamiento

acciones:

  accion dar_un_paso:
      avanza;
  finAccion

  accion vuelta:
      giraIzq;
  finAccion

finAcciones
```

Una vez copiado o abierto el programa dentro de la herramienta InAge, es el momento de compilar. En la siguiente figura se muestra la compilación exitosa de este programa.

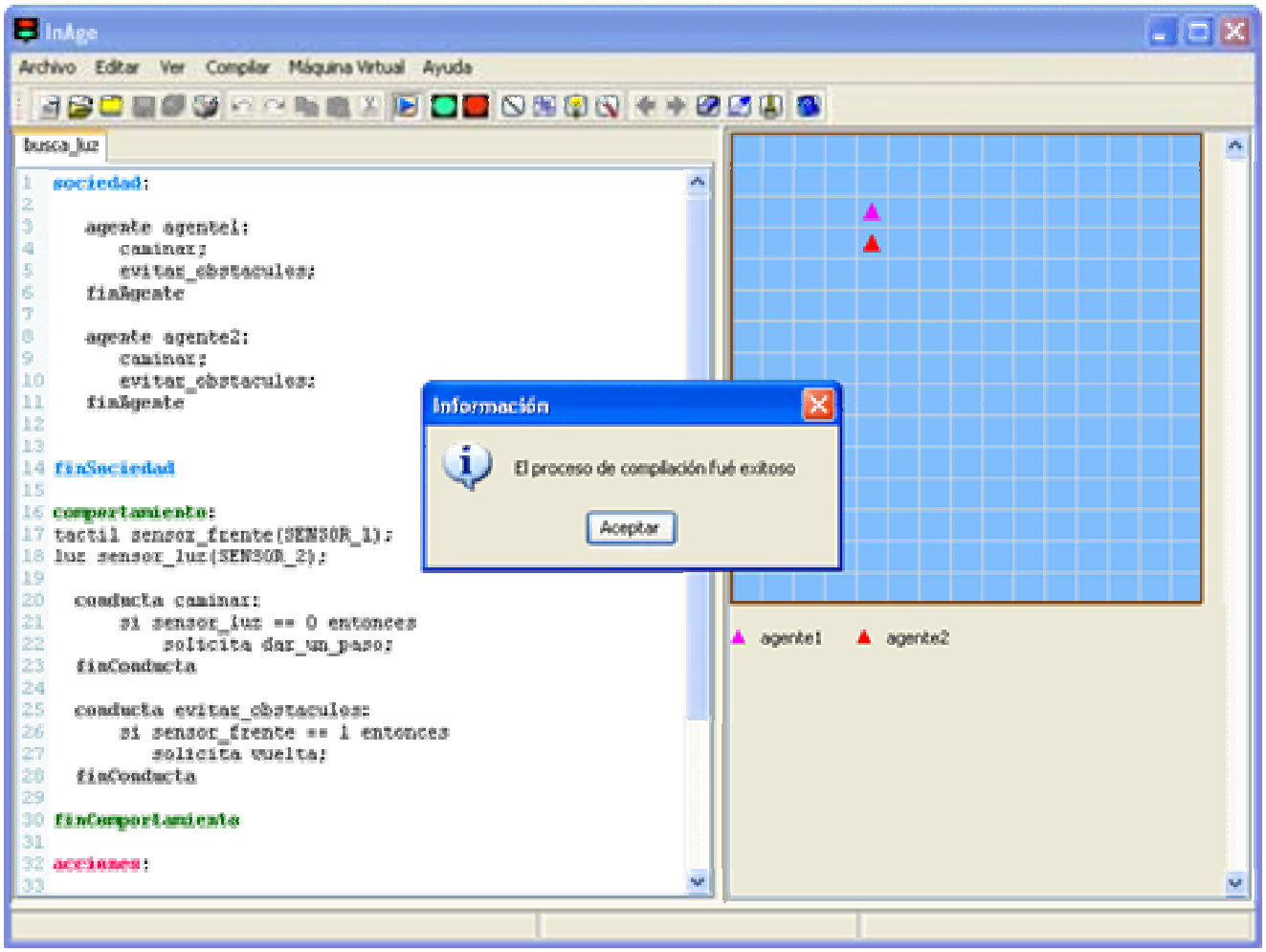


Fig. 5.6 *Compilación exitosa.*

5.5 Resultados obtenidos

Tras la compilación de "busca_luz.age" se generaron seis archivos objeto (con extensión obj) los cuales utilizará la máquina virtual para realizar la simulación.

5.5.1 Código objeto generado

El archivo "busca_luz.Sociedad.Age2000.obj" contiene la relación de todos los agentes que intervienen en la sociedad, en este caso dos: agente1 y agente2. El archivo "busca_luz.Conductas.Age2000.obj" contiene la relación de los agentes con sus conductas. Los archivos "caminar.Age2000.obj" y "evitar_obstaculos.Age2000.obj" corresponden al cuerpo de las conductas "caminar" y "evitar_obstaculos" respectivamente. Así mismo, los archivos "dar_un_paso.Age2000.obj" y "vuelta.Age2000.obj" correspondente al cuerpo de las acciones "evitar_obstaculos" y "vuelta" respectivamente.

Fig. 5.7 *Archivos objeto generados.*

A continuación se muestra el contenido de dichos archivos.

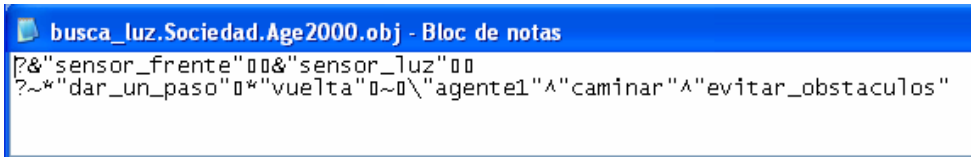


Fig. 5.8 *Archivo "busca_luz.Sociedad.Age2000.obj".*

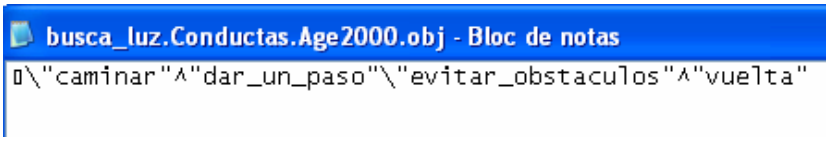


Fig. 5.9 *Archivo "busca_luz.Conductas.Age2000.obj".*

Fig. 5.10 *Archivo "caminar.Age2000.obj".*

Fig. 5.11 *Archivo "evitar_obstaculos.Age2000.obj".*

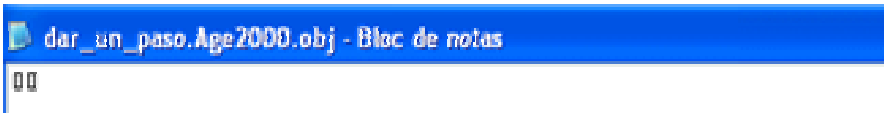


Fig. 5.12 *Archivo "dar_un_paso.Age2000.obj".*

Fig. 5.13 *Archivo "vuelta.Age2000.obj".*

5.5.2 Construcción del mundo virtual

Para poder simular el comportamiento de la sociedad declarada en el programa "busca_luz.age" se debe construir el mundo virtual. En la Figura 5.8 se muestra un ejemplo de mundo virtual en el que se simuló dicho programa. El mundo virtual se guardo con el nombre de archivo "busca_luz.mdo".

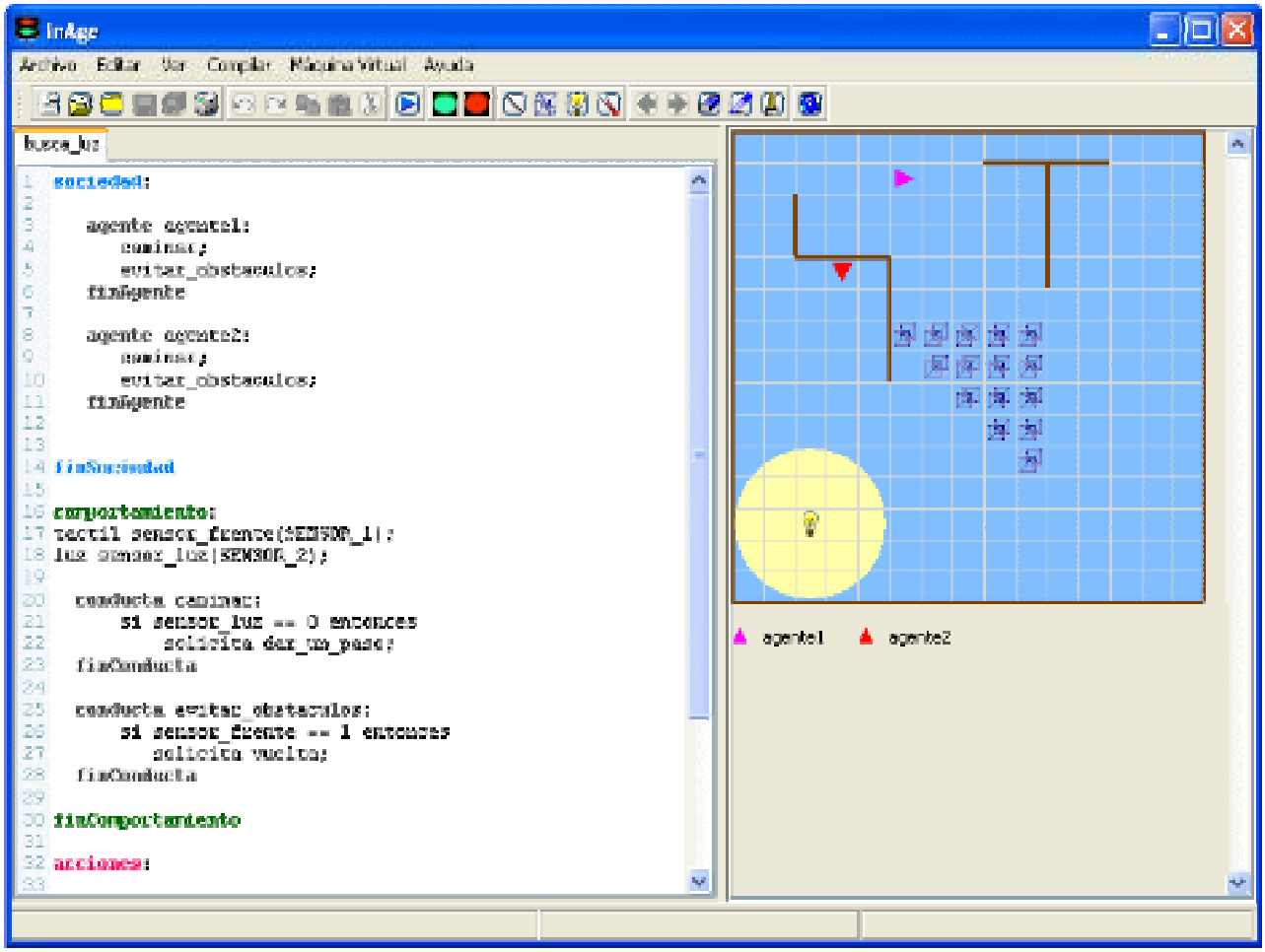


Fig. 5.14 *Mundo virtual "busca_luz.mdo".*

El contenido del archivo " busca_luz.mdo" es el siguiente:

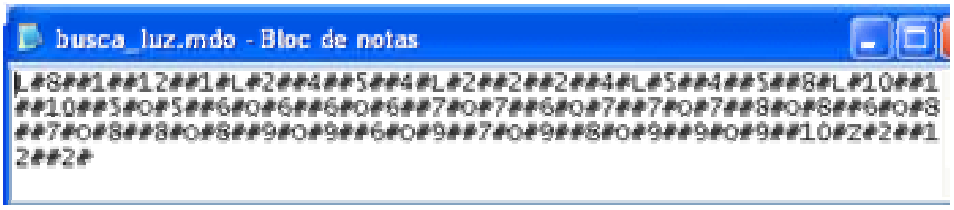


Fig. 5.15 *Archivo "busca_luz.mdo".*

Los mundos virtuales se guardan, ya que se pueden experimentar realizar la simulación de varias sociedades de agentes en un mismo mundo virtual, o una misma sociedad en diferentes mundos virtuales.

5.5.3 Resultado de la simulación

Se realizó la simulación de la sociedad de agentes “busca_luz.age” en el mundo virtual “busca_luz.mdo” con éxito, ambos agentes encontraron la luz, y una vez hallado detuvieron su actuación. Dado que consiste en observar los diferentes pasos que dieron los agentes hasta encontrar la luz, no se puede plasmar en una imagen; es por esto que se incluyó en el disco de instalación de la herramienta una carpeta llamada *Demos* en la que se puede apreciar en un video la ejecución de dicha prueba, “busca_luz.avi”.

En la simulación se observaron algunos momentos en los que no sucedía ningún movimiento por parte de los agentes, y esto es debido a que se debe de esperar la ejecución de los hilos, y una vez que toca el turno al actuador, es cuando se observa un movimiento en el suelo.

Se realizaron diferentes pruebas en la herramienta InAge, con declaración de uno, dos o más agentes en la sociedad. Se observa mejor resultado con un solo agente, ya que con más de uno implica que existen mucho más hilos (*Threads*) y el procesador tiene que dividir su tiempo de atención.

Conclusiones y trabajo futuro

El proyecto actual propone un ambiente para el lenguaje Age2000 construido conforme a la Arquitectura Subsumption. Con él se pretende la creación de programas que exhiban el comportamiento de una sociedad de agentes cuya conducta está basada en acciones simples.

La herramienta InAge es un sistema integral que comprende el compilador del lenguaje Age2000, un ambiente de edición de programas, y un simulador con un espacio para la elaboración de mundos virtuales en los que se desenvolverán los agentes de la sociedad.

Con la construcción del compilador, se ha alcanzado la validez de un programa escrito en el lenguaje Age2000. Este compilador se encuentra dotado de un mecanismo de recuperación de errores para presentarle al programador los errores cometidos en la escritura del programa, si éste fuera el caso. Pero también, hay que decir que el lenguaje no tiene estructuras complejas que propicie una mala programación, o que no sea muy clara.

El simulador ofrece un ambiente fácil de manipular por parte del programador en la construcción de los mundos y no necesita más que compilar un programa fuente del lenguaje, y si no existieran errores, la ejecución del mismo; así que es transparente para el programador la ejecución de un programa.

El ambiente de desarrollo del mundo virtual, en el que se desenvuelve el comportamiento de los agentes, proporcionado por la herramienta InAge, es una propuesta básica y limitada de lo que se puede encontrar en el mundo real. Aún con esto, es una propuesta suficiente para observar comportamiento de los agentes y por lo que se ha mencionado, se ha alcanzado este objetivo.

En el desarrollo de programas basados en Age2000, y en la ejecución de su simulación, se logra observar comportamiento emergente de acciones simples como lo son avanzar, retroceder, girar a la izquierda, girar a la derecha. El comportamiento que se experimentó con la herramienta InAge ha sido con sociedades cuyos objetivos son básicos, es decir, un comportamiento orientado básicamente a avanzar, retroceder y girar. Y es a partir de estos programas que se puede lograr la experimentación dotando a los agentes de un comportamiento más complejo.

El compilador se puede utilizar como herramienta de apoyo en la enseñanza del curso Compiladores que se imparte en la universidad. Al realizar la implementación del compilador bajo el paradigma de POO hace de su diseño un sistema por explorar y analizar en comparación con otros mecanismos y herramientas existentes generadoras de analizadores. Y a su vez, el lenguaje Age2000 y la simulación del comportamiento de una sociedad en un mundo virtual pueden apoyar en la concepción de lo que se pretende lograr en el área de Inteligencia Artificial específicamente con la Arquitectura Subsumption y pueda ser mostrado en el curso de Inteligencia Artificial que se imparte en la universidad.

Para alguien interesado en el tema de agentes reactivos experimente con InAge, primero debe de conocer en qué consiste la herramienta, y con ello hacer programas experimentales. La limitante que se tiene es que el logro de hallar comportamiento complejo, o más apegado a un robot real es muy limitado, esto es, porque el lenguaje Age2000 está diseñado con estructuras básicas de

control y tipo de datos, además de no contar con un mecanismo de resolución de conflictos entre los agentes. El mundo virtual que se construye está muy limitado a los objetos que se pueden colocar en él, y habría que dotarlo de mayores características para poder mostrar y simular lo que existe en un ambiente real.

Como trabajo futuro, queda en el interés del programador el realizar un simulador con características más especializadas apegadas a la realidad, y con esto alcanzar a observar un comportamiento aún más interesante por parte de la sociedad de agentes.

El presente proyecto de tesis se puede retomar para realizar ya sean trabajos físicos, creando robots que sean programados por medio del lenguaje Age2000, o también se puede realizar trabajos de software basados en este lenguaje en los que se persigan objetivos meramente de simulación.

Además, también se puede construir un software donde se pueda simular de manera distribuida el comportamiento de un robot, esto es, un simulador en red centralizando su comportamiento en una máquina. Es decir, hacer de este sistema centralizado un sistema distribuido pretendiendo asemejar un comportamiento concurrente de los agentes.

Por otro lado, se puede ampliar el lenguaje Age2000 con rutinas más especializadas, enriqueciendo su código para proporcionar al programador de sociedades de agentes herramientas de programación que desemboquen en comportamientos más complejos y dirigidos de los agentes.

Bibliografía

[Acosta & Fernández 00] Acosta Mesa, H. G., y Fernández y Fernández, C. A., *Propuesta de un Ambiente para el Modelado de Sociedades de Agentes Reactivos*, Instituto de Electrónica y Computación - Universidad Tecnológica de la Mixteca, Memorias del XI Simposium Nacional de Informática, México, 2000.

[Aho et.al. 98] Aho, A. V., Sethi, R., y Ullman, J.D., *Compiladores: Principios, técnicas y herramientas*, Editorial: Addison Wesley Longman de México S.A. de C.V., 1998.

[Brooks 86] Rodney A. Brooks, *A Robust Layered Control System for a Mobile Robot*, IEEE Journal of Robotics and Automation, Vol. RA-2, no. 1, Marzo 1986.

[Brooks 90] Rodney A. Brooks, Elephants Don't Play Chess, MIT Artificial Intelligence Laboratory, Journal of Robotics and Autonomous System, Cambridge, E.E.U.U., 1990.

[Brooks 91] Rodney A. Brooks, *Integrated Systems Based on Behaviors*, MIT Artificial Intelligence Laboratory, SIGART Bulletin Vol.2, No. 4, E.E.U.U., 1991.

[Deitel & Deitel 98] H. M. Deitel, P. J. Deitel, *Cómo programar en Java*, Editorial: Prentice-Hall Hispanoamericana, S. A., 1998.

[Encarta 03] Enciclopedia Microsoft Encarta 2003, *Definiciones*, Microsoft, 2003.

[Fischer & LeBlanc 88] Charles N. Fischer y Richard J. LeBlanc, Jr., *Crafting A Compiler*, Editorial: The Benjamin/Cummings Publishing Company, Inc., 1988.

[Fowler 99] Martin Fowler, *UML Gota a Gota*, Editorial: Addison Wesley Longman de México, S.A. de C.V. México 1999.

[García et.al. 01] Pedro García, Tomás Pérez, José Ruiz, Encarna Segarra, José M. Sempere, M. Vázquez de Parga, *Teoría de Autómatas y Lenguajes Formales*, Editorial: Alfaomega Grupo editor, S.A. de C.V., 2001.

[Hopcroft & Ullman 79] John E. Hopcroft, Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Editorial: Addison – Wesley Publishing Company, 1979.

[Hopcroft et.al. 01] John Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Editorial: Addison-Wesley, 2001.

[Lemone 79] Dr. Karen A. Lemone, *Design of Compilers: Techniques of programming language translation*, Editorial: Addison – Wesley Publishing Company, 1979.

[Mishkoff 88] Henry C. Mishkoff, *A fondo: Inteligencia Artificial*, Editorial: Ediciones Anaya Multimedia, S. A., Marzo 1988.

[Nilsson 01] Nils J. Nilsson, *Inteligencia Artificial: Una nueva síntesis*, Editorial: Mc Graw Hill/Interamericana de España, S.A. U., 2001.

[Rich & Knight 94] Elaine Rich y Kevin Knight, *Inteligencia Artificial*, Editorial: Mc Graw Hill/Interamericana de España, S.A.U., 1994.

[Rumbaugh et.al. 00] James Rumbaugh, Ivar Jacobson y Grady Booch, *El Lenguaje de Modelado. Manual de Referencia*, Editorial: Pearson Educación, S. A. Madrid, 2000.

[Russell & Norvig 96] Stuart J. Russell y Peter Norvig, *Inteligencia Artificial: un enfoque moderno*, Editorial: Prentice Hall Hispanoamericana S. A., 1996.

[Salas 92] Jesús Salas Parrilla, *Sistemas Operativos y Compiladores*, Editorial: McGraw-Hill / Interamericana de España, S.A., 1992.

[Silva 04] Roberto Silva, Casos *de Uso: lo más importante y menos entendido de UML*, Software Gurú: Conocimiento en Práctica, No. 0, Noviembre 2004.

[Schmuller 00] Joseph Schumller, *Aprendiendo UML en 24 horas*, Editorial: Pearson Educación, S.A. México, 2000.

[Teufel et.al. 95] Bernd Teufel, Stephanie Schmidt y Thomas Teufel, *Compiladores: Conceptos fundamentales*, Editorial: Addison-Wesley Iberoamericana, S.A., 1995.

[Ullman 76] Jeffrey D. Ullman, *Fundamental Concepts of Programming Systems*, Editorial: Addison - Wesley Publishing Company, 1976.

[Winston 94] Patrick Henry Winston, *Inteligencia Artificial*, Editorial: Addison-Wesley Iberoamericana, S.A., 1994.

Internet

[Atkinson 98] John Atkinson A., *Diseño de Agentes Autónomos Utilizando un Enfoque de Control basado en Conductas*, http://www.electa.uta.cl/~revista/control-98.pdf., Revista Facultad de Ingeniería, U.T.A. (Chile), Vol. 5, 1998.

[Cavadini 00] http://www.ucse.edu.ar/fma/compiladores/tools.html, *Herramientas para la construcción de Compiladores*, Salvador D. Cavadini, 2000.

[Generadores] http://catalog.compilertools.net/lexparse.html, *Lexer and Parser Generators*.

[Hodges 02] http://www.turing.org.uk/publications/testbook.html, *Alan Turing and the Turing Test*, Andrew Hodges, Wadham College Oxford, Octubre 2002.

[Lexico] http://www.etfpa.br/eustakio/analisadorlexicooo.html, *Analizador Léxico Orientado a Objetos*.

[Navarro 99] http://www.redcientifica.com/doc/doc199903310004.html, *IA Clásica vs Arquitectura de Subducción*, David Navarro, Red Científica: Ciencia Tecnología y Pensamiento, Marzo, 1999.

[Recursos 05] http://www.escet.urjc.es/~procesal/recursos.html, *Recursos*, 10 mayo, 2005.

Apéndice A. Glosario

Agente: Un agente se puede ver como un ente que percibe su ambiente a través de sensores y actúa sobre él a través de efectores [Atkinson 98].

Agente Autónomo (Agente Adaptativo): Son agentes con capacidad de adaptarse al ambiente y operar en forma continua sobre éste.

Árbol de análisis sintáctico: Derivación gráfica de una secuencia de símbolos a partir del inicial mediante las producciones gramaticales.

Autómata Finito (AF): Dispositivo que recibe una entrada y pasando por varios estados produce una salida apropiada.

Autómata Finito No Determinista (AFN): Dispositivo de reconocimiento en el que para cada estado y cada caracter de entrada puede existir más de un estado de transición.

Autómata Finito Determinista (AFD): Dispositivo de reconocimiento que discrimina entre palabras de entrada en un alfabeto finito. Los estados tienen un próximo estado único (para un caracter y estado dado).

BNF: Backus-Naur Form – Forma de Backus-Naur. También llamada gramática independiente del contexto, la cual es una notación para especificar la sintaxis de un lenguaje.

CDC: Compañía Control Data.

CMU: Universidad Carnegie-Mellon.

DEC: Digital Equipment Corporation.

Efectores: Dispositivo que produce determinados efectos en el entorno bajo el control de un robot [Russell & Norvig 96].

Etología: Estudio del comportamiento animal en condiciones naturales.

IDE: Integrated Development Environment, entorno de desarrollo integrado.

LISP: Representación de estructuras simbólicas.

MCC: Compañía de Tecnología de Ordenadores y Microelectrónica.

MIT: Massachusetts Institute of Technology – Instituto Tecnológico de Massachusetts.

Sensores: Es todo aquello capaz de modificar el estado de cómputo de un agente como respuesta a un cambio en el estado del mundo.

Silogismo: Esquemas de estructuras de argumentación, propuestas por Aristóteles, mediante las que se llega a conclusiones correctas si se parte de premisas correctas.

SRI: Stanford Research Institute - Instituto de Investigación de Stanford.

TI: Texas Instruments.

Tokens: Componentes léxicos que se utilizan en el analizador sintáctico para hacer el análisis.

UML: Unified Modeling Language – Lenguaje Unificado de Modelado.

Apéndice B. Lenguaje Age2000

B.1 Introducción

El lenguaje Age2000 define en su gramática la construcción de una sociedad de agentes reactivos. Cuenta con palabras reservadas, tipos de variables, operadores aritméticos, relacionales y lógicos, además de delimitadores.

Se debe de tomar en cuenta que la declaración de los sensores se cumple para todos los agentes, es decir, que todos los agentes tienen las mismas características.

B.2 Palabras Reservadas

accion
acciones
agente
avanza
conducta
conductas
comportamiento
entero
entonces
fin
finAccion
finAcciones
finAgente
finConducta
finConductas
finComportamiento
finSociedad
fotoCeldaFD
fotoCeldaFI
fotoCeldaTD
fotoCeldaTI
giraDer
giraIzq
hacer
infrarrojo
inicio
mientras
motor
MOTOR_A
MOTOR_B
MOTOR_C
otro
repite
retrocede
SENSOR_1
SENSOR_2
SENSOR_3
si
sociedad
solicita
tactilDer
tactilIzq
tactilTra
veces

B.3 Operadores Aritméticos

+	Suma
-	Resta
/	División
*	Multiplicación

B.4 Operadores Relacionales

<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
= =	Igual a
!=	Diferente

B.5 Operadores Lógicos

&	"y"
|	"o"

B.6 Signos de puntuación

(	Paréntesis que abre
)	Paréntesis que cierra
;	Punto y coma
:	Dos puntos
,	Coma
=	Asignación
//	Comentarios

B.7 Tipos de Variable

El lenguaje Age2000 contiene los siguientes tipos de variables:

Entero
Decimal
Agente
Conducta
Accion

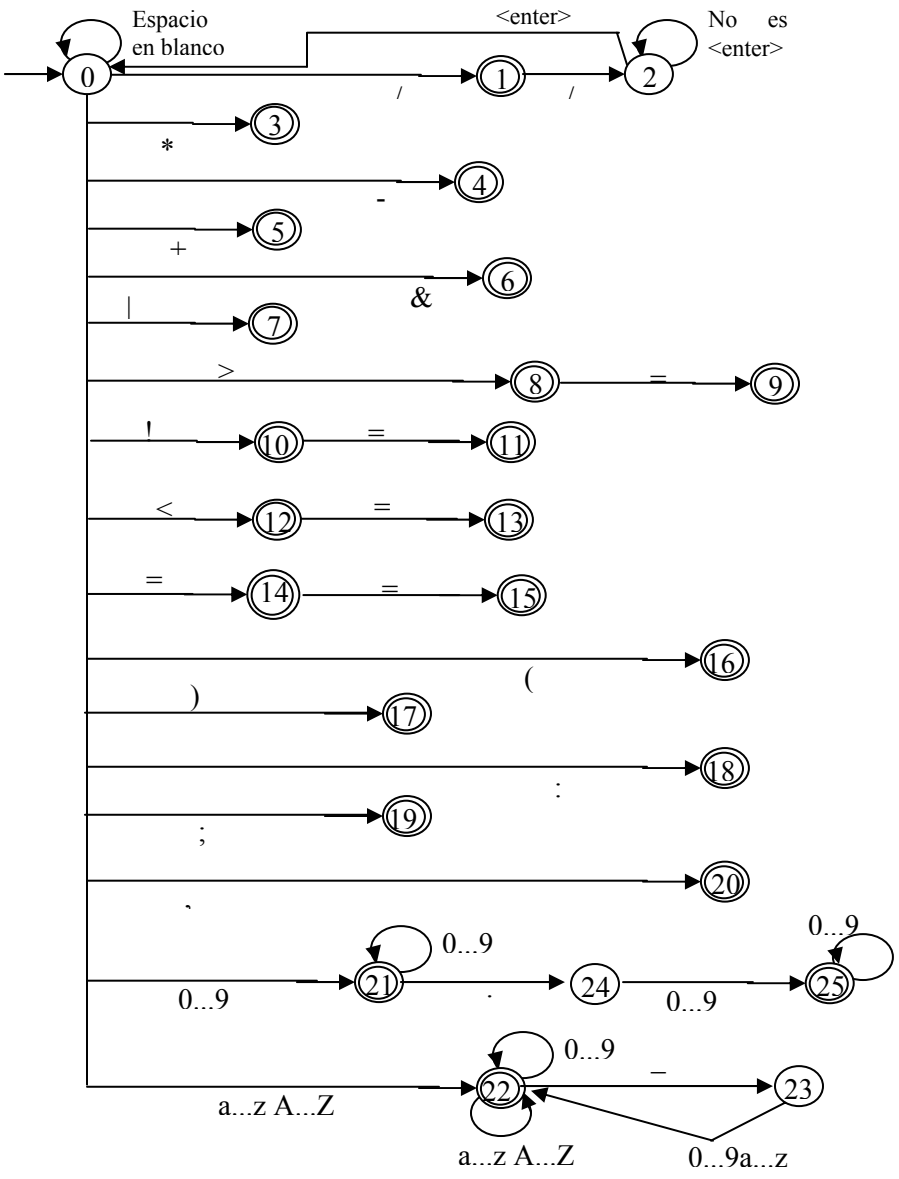
El nombre de las variables están compuestas de la combinación de letras (a..zA..Z), dígitos(0..9) y guión bajo ("_"). Las variables deben de empezar con letra y no terminar con guión bajo.

Apéndice C. Analizador Léxico

C.1 Expresiones Regulares

operadores _ aritméticos	→	**+** | **-** | ***** | **/**
operadores _ lógicos	→	**&** | | | **!**
operadores _ relacionales	→	< | > | <= | >= | = = | **!=**
asignación	→	=
comentarios	→	//
delimitador	→	**(** | **)** | **:** | **;**
Palabra	→	[a..zA..Z] [_ | [a..z A..Z 0..9]*]* [a..z A..Z 0..9]
Enteros	→	[0..9]$^{+}$
Decimales	→	[0..9]$^{+}$. [0..9]$^{+}$

C.2 Autómata Finito Determinístico



C.3 Tabla de Transiciones

Las columnas de la Tabla de Transiciones corresponden a los símbolos de entrada (caracteres válidos por la gramática), y los renglones corresponden a cada uno de los estados del autómata.

Si el estado actual es S y el carácter leído es c, entonces T(S,c) será el siguiente estado a visitar, o una bandera de error indica que c no puede ser parte de la cadena, es decir, si el siguiente carácter de entrada corresponde a una transición válida desde el estado actual, entonces se toma al nuevo estado actual como el estado al cual dicha transición apunta.

Las banderas de error que arroja la tabla de transiciones son las siguientes:

- -1 significa que no hay ninguna transición válida con esa entrada.
- -2 es que se ha incurrido en un error con esa entrada y en el estado donde se encuentra.

	Esp	/	*	-	+	&	|	>	<	=	!	(	)	:	;	,	0...9	a...z A...Z	_	.
0	0	1	3	4	5	6	7	8	12	14	10	16	17	18	19	20	21	22	-2	-2
1	1	2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
8	8	-1	-1	-1	-1	-1	-1	-1	-1	9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	10	-1	-1	-1	-1	-1	-1	-1	-1	11	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
11	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
12	12	-1	-1	-1	-1	-1	-1	-1	-1	13	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
13	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
14	14	-1	-1	-1	-1	-1	-1	-1	-1	15	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
15	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
16	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
17	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
18	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
19	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
20	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
21	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	21	-2	-2	24
22	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	22	22	23	-1
23	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	22	22	-2	-1
24	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	25	-2	-2	-2
25	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	25	-2	-2	-2

Apéndice D. Lenguaje Age2000 en notación BNF

<programaAge> → <inicio> EofSym
<inicio> → <sociedad> <comportamiento> <acciones>

<sociedad> → **sociedad** : <agentes> **finSociedad**
<agentes> → <agente> { <agente> }
<agente> → **agente** <lista_robot>: <lista_conductas> **finAgente**
<lista_robot> → var_robot
<lista_conductas> → <alm_conducta> { <alm_conducta> }
<alm_conducta> → var_conducta;

<comportamiento> → **comportamiento** : [<sensores>] {<enteros> | <decimales>} <conductas> **finComportamiento**
<sensores> → { <sensor> } $^{n=0..3}$
<sensor> → [**tactil** | **luz** | **temperatura**] <lista_sensor> ;
<lista_sensor> → <asig_sensor> { , <asig_sensor> }$^{0..2}$
<asig_sensor> → var_sensor (**SENSOR_1** | **SENSOR_2** | **SENSOR_3**)
<conductas> → <conducta> { <conducta> }
<conducta> → **conducta** <lista_conducta> : {<enteros> | <decimales>} <instruccion_ciclo> **finConducta**
<lista_conducta> → var_conducta
<instrucción_ciclo> → { <instrucción> }
<instrucciones> → <instrucción> | <bloque>
<instrucción> → <cond_si> | <cond_mientras> | <cond_repite> | <asignación> | <solicita>
<cond_si> → **si** <expresion> **entonces** <instrucciones> [**otro** <instrucciones>]
<cond_mientras> → **mientras** <expresion> **hacer** <instrucciones>
<cond_repite> → **repite** <op_entero> **veces** <instrucciones>
<bloque> → **inicio** <instrucción_ciclo> **fin**
<solicita> → **solicita** var_accion ;
<expresión> → < elec_exp > { <op_logico> < elec_exp >}
<elec_exp> → < id_sensor > | var_sensor {<op_Relac> <operacion>} | <exp_sensor>
<id_sensor> → ! var_sensor
<exp_sensor> → <operacion> <op_relac> [<operación> | var_sensor]

<acciones> → **acciones** : <motores> {<enteros> | <decimales>} <decl_accion> **finAcciones**
<motores> → **motor** <lista_motor>;
<lista_motor> → <asig_motor> { , <asig_motor> }$^{0..2}$
<asig_motor> var_motor (**MOTOR_A** | **MOTOR_B** | **MOTOR_C**)
<decl_accion> → <accion> { <accion> }
<accion> → **accion** <lista_accion> : {<enteros> | <decimales>} <instrucciones_ciclo2> **finAccion**
<lista_accion> → var_accion
<instrucción_ciclo2> → { <instrucción2> }
<instrucciones2> → <instruccion2> | <bloque2>
<instrucción2> → <cond_si2> | <cond_mientras2> | <cond_repite2> | <asignación> |

<primitivas>
<cond_si2> → **si** <exp_relacional> **entonces** <instrucciones2> [**otro** <instrucciones2>]
<cond_mientras2> → **mientras** <exp_relacional> **hacer** <instrucciones2>
<cond_repite2> → **repite** <op_entero> **veces** <instrucciones2>
<bloque2> → **inicio** {<instrucción_ciclo2>} **fin**
<primitivas> → **avanza** | **retrocede** | **giraIzq** | **giraDer**

<enteros> → **entero** <lista_entero> { , <lista_entero>} ;
<lista_entero> → var_entero [= [-] valor_entero]
<id_entero> → var_entero | valor_entero

<decimales> → **decimal** <lista_decimal> { , <lista_decimal>} ;
<lista_decimal> → var_decimal [= [-] valor_decimal]
<id_decimal> → var_decimal | valor_decimal

<numeros> → var_entero | valor_entero | var_decimal | valor_decimal
<asignacion> → <asig_decimal> | <asig_entero>
<asig_decimal> → var_decimal = <operación> ;
<asig_entero> → var_entero = <op_entero> ;
<op_entero> → (<cpo_entero>) | <cpo_entero>
<cpo_entero> → id_entero { <op_aritm> <op_entero> }

<op_relac> → < | > | <= | >= | = | !=
<op_logico> → & | '|'
<op_aritm> → + | - | * | /

<exp_relacional> → <operación> [<op_relac> <operación>]
<operación> → (<operación_decl>) | <operación_decl>
<operación_decl> → <numeros> { <op_aritm> <operación> }

Apéndice E. Generación de Código

La siguiente tabla contiene el código binario al que se traduce el lenguaje Age2000.

* Primitiva	1	☺
avanza	2	☻
retrocede	3	♥
giraIzq	4	♦
giraDer	5	♣
'<'	6	♠
'>'	7	•
'<='	8	◘
'>='	9	○
'= ='	11	♂
'! ='	12	♀
'&'	14	♫
'|'	15	☼
'+'	16	►
'-'	17	◄
'*'	18	↕
'/'	19	‼
mientras	21	§
repite	22	▬
si	23	↨
otro	24	↑
* Nombre de variable	34	"
* Inicio / Fin Número	35	#
* Variable	36	$
* Fin operación aritmética	37	%
* Asignación	61	=
* Tabla de Sensores	63	?
* Tabla de Enteros/Decimales	64	@
* Inicio condición	91	[
* Nivel 1 (agente / conducta)	92	\
* Fin condición	93	]
* Nivel 2 (conducta / acción)	94	^
Inicio bloque	123	{
Fin bloque	125	}
* Tabla de Acciones	126	~

* Son banderas que distinguen la identidad del próximo(s) valor(es) de la cadena de código objeto.

Apéndice F. Modelado

ANALIZADOR LÉXICO
VISIÓN

1. Introducción

El presente documento tiene el objetivo de presentar una visión del Analizador Léxico, es decir, una vista previa al modelado del sistema, y sus requerimientos.

2. Vista General

El Analizador Léxico es un proceso de análisis de un compilador. Su funcionamiento radica en la lectura de caracteres elaborando componentes léxicos necesarios para el analizador sintáctico.

2.1. Requisitos Funcionales

El Analizador Léxico elimina los espacios en blanco y comentarios.

El Analizador Léxico cuenta con una lista de tokens permitidos los cuales se comparan con los caracteres obtenidos del código fuente, esto es para identificar un componente léxico.

El Analizador Léxico identifica al componente léxico como palabra reservada.

El Analizador Léxico identifica al componente léxico como variable.

En respuesta a alguna petición del Analizador Sintáctico, el Analizador Léxico puede enviar el componente léxico identificado o un mensaje de fin del análisis léxico (fin de archivo del código fuente).

En el proceso de identificación de un componente léxico, el Analizador Léxico puede detectar errores, los cuales almacena para su posterior despliegue, y se recupera de éstos para continuar con el análisis. Estos errores pueden ser que se encuentren caracteres no permitidos por el lenguaje, el orden en el que se agrupan los caracteres, variables, numéricos, entre otros.

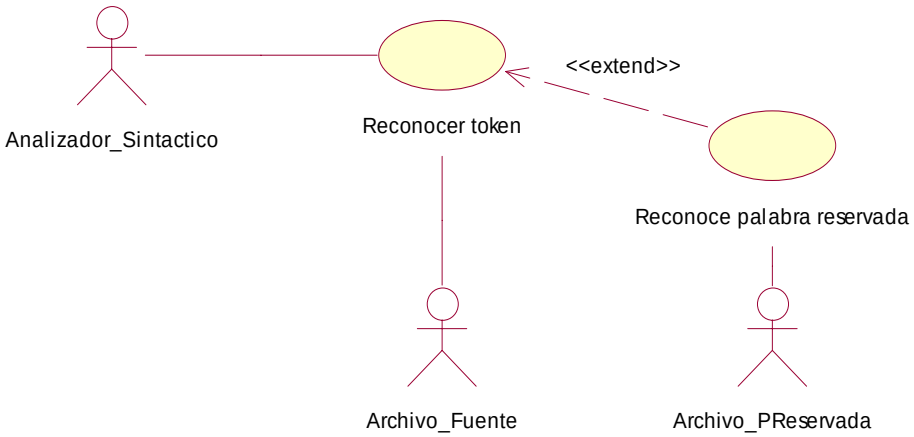
El Analizador Léxico es responsable del estado en el que se encuentra el código fuente, es decir, maneja las acciones que se aplican sobre el código fuente (abre el archivo fuente, realiza una lectura sobre el archivo fuente, retrocede un caracter, avanza un caracter, identifica el fin de archivo del código fuente, cierra el archivo fuente).

El funcionamiento del Analizador Léxico se realizará por cada petición de un componente léxico por parte del Analizador Sintáctico.

2.2. Requisitos No Funcionales

Para llevarse a cabo el funcionamiento del Analizador Léxico se necesita contar con un código fuente, el cual contiene un programa elaborado en lenguaje Age2000.

Diagrama de Caso de Uso



1. Reconocer Token

1.1. Breve descripción

Realiza el reconocimiento de tokens y envía éstos al Analizador Sintáctico para verificar que las expresiones en el código fuente cumplan con la gramática.

2. Flujo de Eventos

2.1. Flujo Básico

Acción del Actor	Respuesta del Sistema
1. El Analizador Sintáctico solicita un token.	
	2. El Analizador Léxico lee caracteres del código fuente. 3. Se realiza el reconocimiento del token. 4. Se envía el token reconocido al Analizador Sintáctico.
5. Analizador Sintáctico recibe el token.	

2.2. Flujos Alternativos

2.2.1. Fin de archivo en el código fuente.

El Analizador Léxico recibe la petición del siguiente token, al efectuar el proceso de lectura del caracter del código fuente identifica el fin de archivo. El Analizador Léxico reporta que ya no tiene más tokens que reconocer.

2.2.2. No se reconoce un token.

El Analizador Léxico identifica que el token obtenido no es parte de la gramática, así que almacena el error y se recupera de él para reconocer el siguiente token.

3. Precondiciones

El Analizador Léxico opera bajo la solicitud del Analizador Sintáctico y con un código fuente. Para almacenar los errores encontrados en el Analizador Léxico es necesario que exista la tabla de errores.

4. Poscondiciones

4.1. Se identifica un token, y el código fuente queda listo para cuando se realice una próxima lectura.

4.2. Ya no hay más tokens qué reconocer, así que culmina el funcionamiento del Analizador Léxico ya que se ha llegado al final del archivo del código fuente.

1. Reconoce Token

1.1. Propósito

La Especificación de Realización del Caso de Uso "Reconocer Token" tiene como propósito especificar a detalle el caso de uso "Reconocer Token", así como sus flujos alternos. Una vez que se especifique el caso de uso, se diseña la arquitectura del sistema creando diferentes vistas del mismo.

1.2. Alcance

La realización de este documento presenta el análisis y diseño de un caso de uso que comprende el proceso de análisis léxico en un compilador.

1.3. Vista General

El análisis y diseño empieza con la especificación del caso de uso "Reconocer Token" y sus flujos alternos que son: "Fin de archivo del código fuente" y "Token inválido". Se elabora el diagrama de clases y diagramas de secuencia.

2. Flujo de Eventos

2.1. Flujo básico

Acción del Actor	Respuesta del Sistema
1. El Analizador Sintáctico solicita un token para verificar si las expresiones se encuentran dentro de la gramática Age2000.	2. El Analizador Léxico solicita el caracter en turno del código fuente. 3. El código fuente lee el siguiente caracter. En caso de no leer ningún caracter, se ejecuta el flujo alterno "Fin de Archivo del código fuente". 4. El caracter se analiza mediante una transición del estado actual al siguiente estado al que lleva el caracter en la tabla de transiciones. (Si se trata del

	primer caracter se parte del estado 0) 5. Se conserva el caracter en una cadena temporal y así mismo se conserva el estado actual. 6. Se repite el proceso a partir del paso 2 hasta que ya no se permitan más transiciones. 7. Se reconoce un token, ya que el estado actual es un estado final de la tabla de transiciones. En caso contrario, se ejecuta el flujo alterno "Token inválido". 8. Se envía el token reconocido al Analizador Sintáctico.
9. Analizador Sintáctico recibe el token.	

2.2. Flujos Alternativos

2.2.1. Fin de Archivo del código fuente.

Paso3. Se ha llegado al fin de archivo del código fuente.

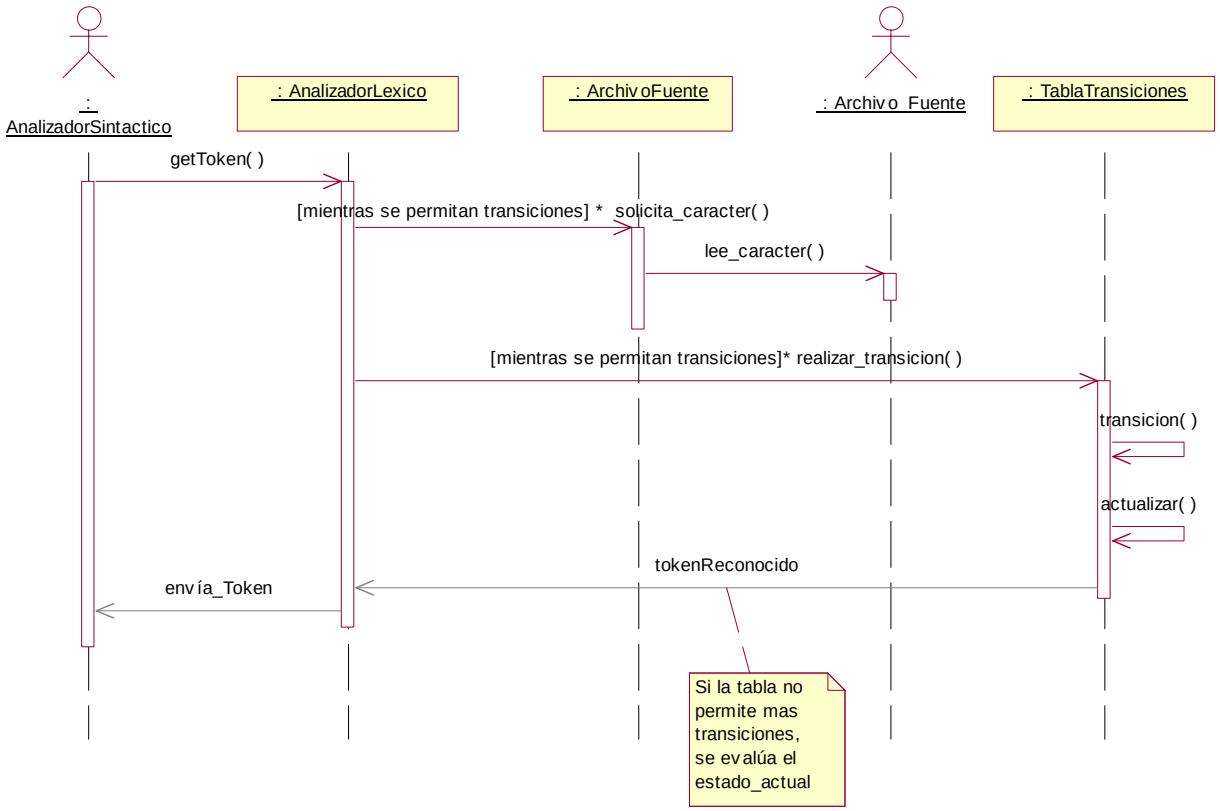
Acción del Actor	Respuesta del Sistema
	6. El Analizador Léxico no recibe ningún caracter del código fuente. 7. El Analizador Léxico reporta al Analizador Sintáctico que ya se ha llegado al final del archivo del código fuente.
5. El Analizador Sintáctico recibe el identificador del final del archivo del código fuente.	

2.2.2. Token inválido

Acción del Actor	Respuesta del Sistema
	6. No se reconoce un token, ya que el estado actual no es un estado final en la tabla de transiciones. 7. El Analizador Léxico reconoce el error y lo almacena en una tabla de errores identificados. 8. El Analizador Léxico empieza el proceso de reconocer un nuevo token, esto es, se repite el proceso de identificar un token a partir del paso 2 del flujo básico "Reconocer token".

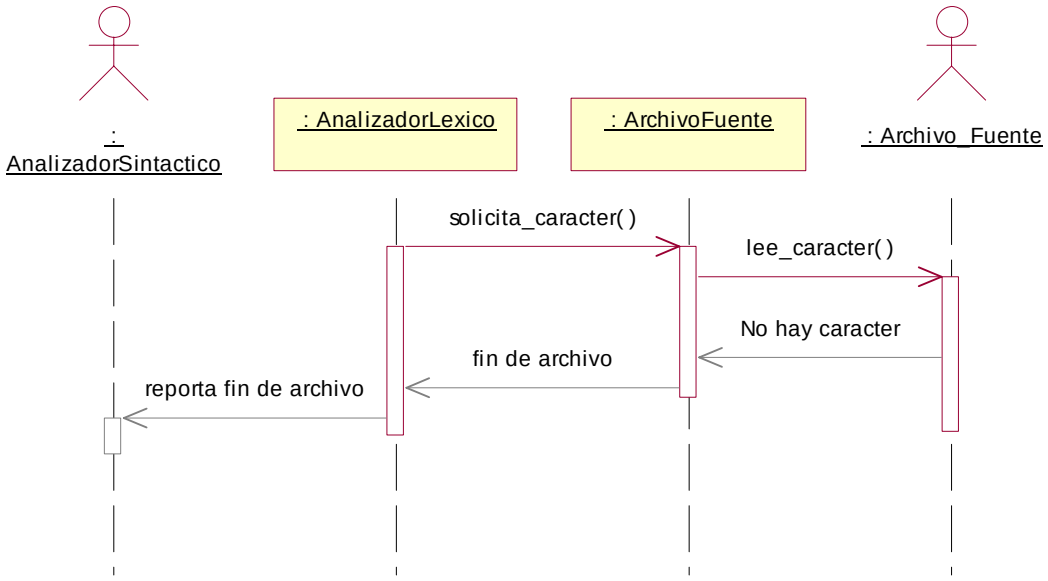
3. Diagramas de secuencia

3.1. Caso de Uso "Reconocer Token"

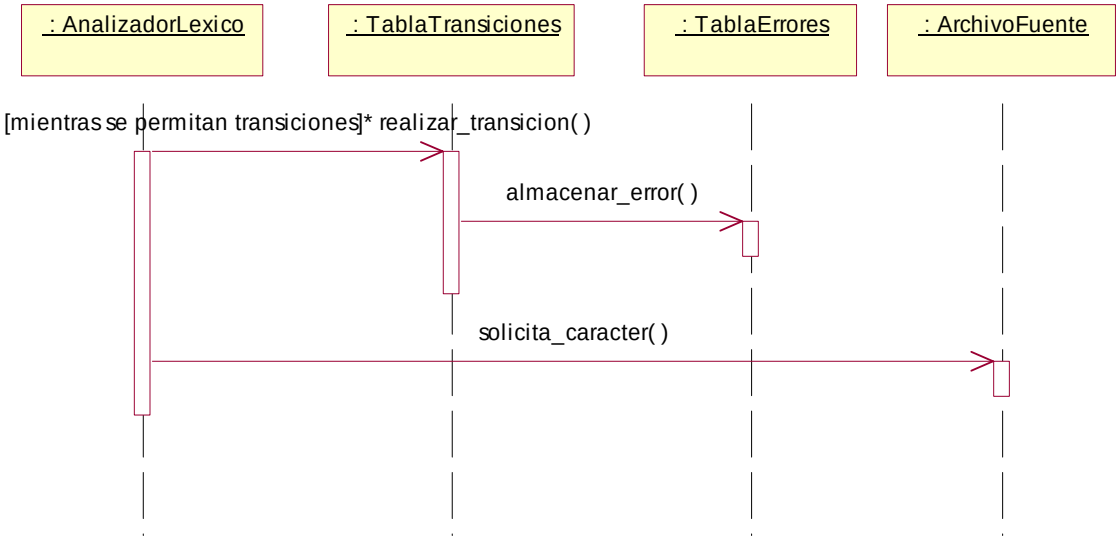


3.2. Flujos Alternativos

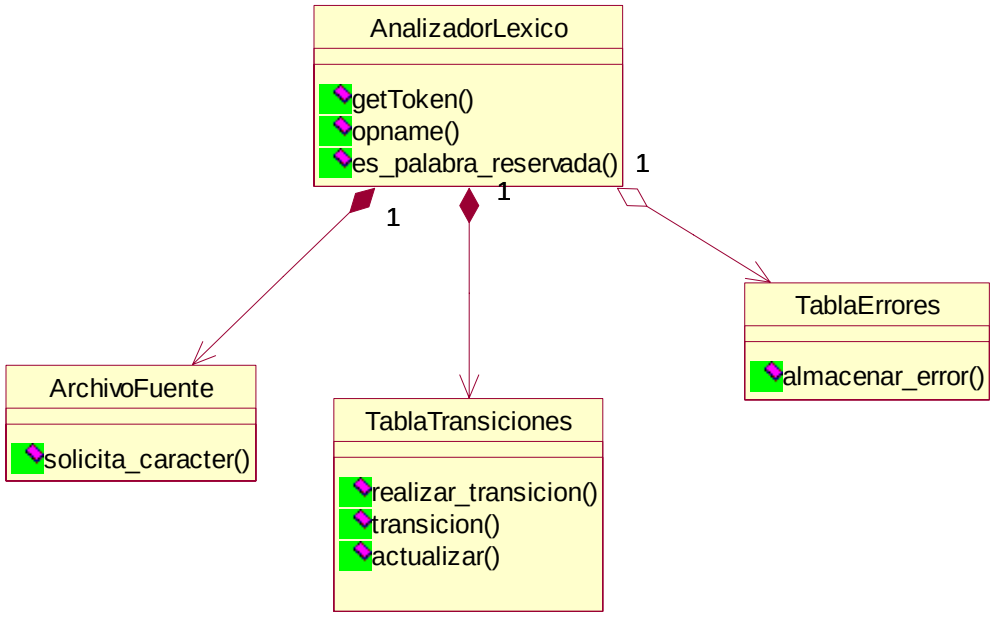
3.2.1. Caso de Uso “Fin de Archivo del código fuente”



3.2.2. Caso de Uso “Token inválido”



4. Diagrama de Clases



1. Reconoce Palabra Reservada

1.1. Breve descripción

Realiza el reconocimiento entre variable y palabra reservada. Posteriormente se envía el componente léxico apropiado al Analizador Sintáctico para continuar con la verificación de las expresiones gramaticales.

2. Flujo de Eventos

2.1. Flujo Básico

Posterior al caso de uso "Reconoce Token" y al reconocimiento de un token válido, éste último se puede reconocer como palabra reservada o variable. Para esto, se tiene que comparar el token con la tabla de palabras reservadas.

2.2. Flujos Alternativos

2.2.1. El token es una variable

Posterior al caso de uso "Reconoce Token" y al reconocimiento de un token válido, éste último se identifica como variable.

3. Precondiciones

El Analizador Léxico opera bajo la solicitud del Analizador Sintáctico y con un código fuente.

4. Poscondiciones

Se identifica un token y el código fuente queda listo para cuando se realice una próxima lectura.

1. Reconoce Token

1.1. Propósito

La Especificación de Realización del Caso de Uso "Reconoce Palabra Reservada" tiene como propósito especificar a detalle el caso de uso "Reconoce Palabra Reservada", así como su flujo alterno.

1.2. Alcance

La realización de este documento presenta el análisis y diseño de un caso de uso que comprende el proceso de análisis léxico en un compilador.

1.3. Vista General

El análisis y diseño empieza con la especificación del caso de uso "Reconoce Palabra Reservada" y su flujo alterno que es: "El token es una variable". Se elabora el diagrama de clases y diagramas de secuencia.

2. Flujo de Eventos

2.1. Flujo Básico

Acción del Actor	Respuesta del Sistema
1. Caso de Uso <<Reconoce Token>> identifica que un token puede ser una variable o palabra reservada.	2. El Analizador Léxico solicita la identificación del token. 3. Se compara la cadena temporal con la tabla de palabras reservadas. 4. Se identifica la palabra reservada. En caso de no serlo, se ejecuta el flujo alterno "El token es una variable". 5. El Analizador Léxico envía al Analizador Sintáctico el componente léxico de la palabra reservada identificada.

6. Analizador Sintáctico recibe el token.	

2.2. Flujo Alternativo

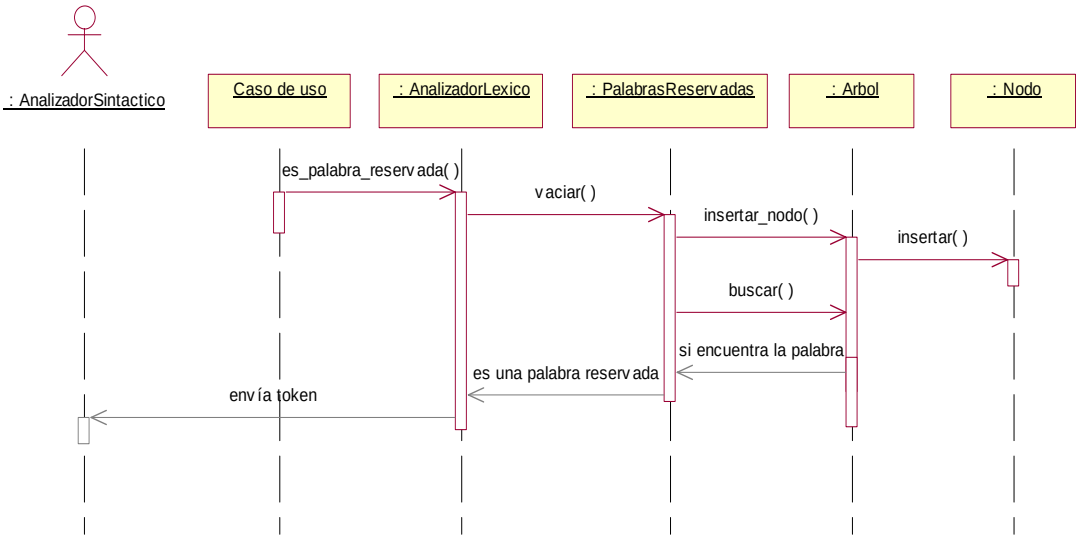
2.2.1. El token es una variable

Paso 4. Se ha llegado al fin de archivo del código fuente.

Acción del Actor	Respuesta del Sistema
	4. Se identifica una variable. 5. Ingresa la variable en la Tabla de Símbolos 6. El Analizador Léxico envía al Analizador Sintáctico el componente léxico de la variable.
7. Analizador Sintáctico recibe el token.	

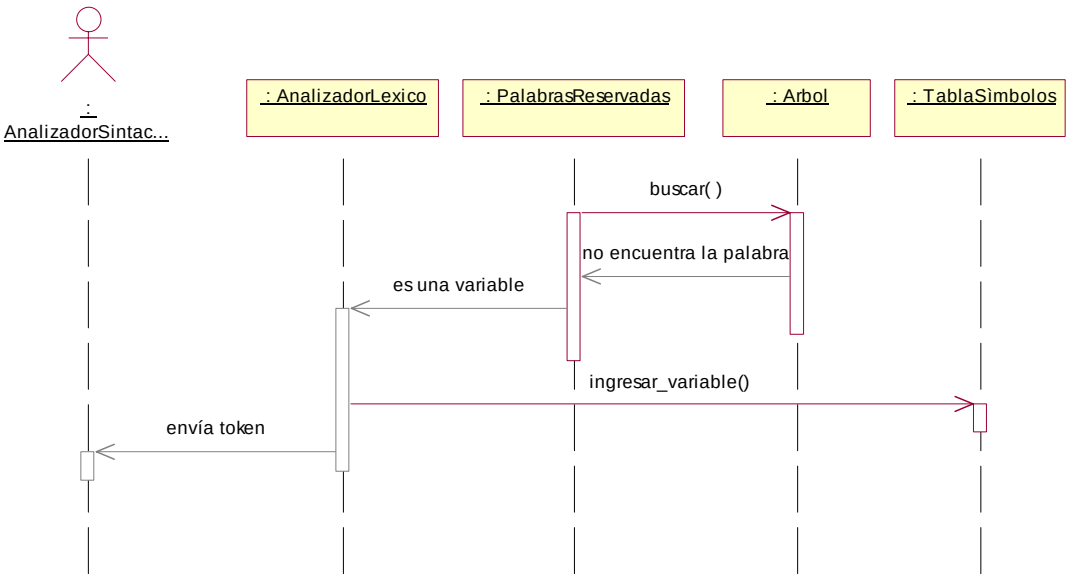
3. Diagramas de secuencia

3.1. Caso de Uso “Reconoce Palabra Reservada”



3.2. Flujo Alternativo

3.2.1. Caso de Uso “El token es una variable”



4. Diagrama de Clases