96

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

INTERFAZ PARA UN SISTEMA EXPERTO Y UNA BASE DE DATOS *SYBASE* EN UN AMBIENTE DE SISTEMA OPERATIVO *LINUX*.

T E S I S

QUE PARA OBTENER EL TITULO DE:

INGENIERO EN COMPUTACION

P R E S E N T A N :

TOLEDO PANIAGUA GPE. MIREYA

MARTINEZ MEZA EDGAR

ASESOR: DRA. ANA MARIA VAZQUEZ VARGAS

MEXICO, D. F.

2002

A mi madre, por todo el apoyo, comprensión, confianza y amor que me ha dado y por su ejemplo de superación incansable. *"Gracias"*.

Con infinito respeto a mi padre, por todo su apoyo. *"Gracias"*.

A mis hermanos (Alvaro, Roberto y Carlos) por su amistad incondicional. *"Gracias"*.

A la Dra. Ana María Vázquez V., por haberme orientado para la realización del presente trabajo. Asimismo, al DR. Jesús Savage C. por sus valiosas aportaciones al mismo. *"Gracias"*.

Con un infinito respeto y un bello recuerdo a todos mis profesores, ya que sin ellos no hubiese logrado mi formación profesional. *"Gracias"*.

A la Universidad Nacional Autónoma de México, en especial a la Facultad de Ingeniería, por existir y contribuír a la formación de profesionistas y por la oportunidad que nos brindas. *"Gracias"*.

Con mucho cariño y gratitud a Oscar, por su apoyo y comprensión, que fueron muy importantes para la culminación de esta etapa de mi vida. *"Gracias"*.

Con mucho aprecio a todos mis amigos y compañeros, que formaron parte de cada día transcurrido hasta alcanzar esta meta. *"Gracias"*.

Mireya Toledo Panlaqua.

Agradecimientos

Agradezco a DIOS quien siempre a iluminado mi camino con felicidad.

Agradezco a Mis Padres por el enorme apoyo que siempre han tenido hacia mi.

Agradezco a mi Madre Rosila Meza Zarate que siempre ha escuchado mis penas y las ha hecho mas llevaderas.

Agradezco a mi Padre Ing. Carlos Martínez Fierro quien me ha enseñado a caminar por la vida con carácter y firmeza.

Agradezco a mis Hermanas que siempre han estado a mi lado, y que siempre han apoyado mis proyectos.

Agradezco a mi Esposa Margarita Elizabeth Vallejo Gómez porque siempre ha compartido a mi lado los momentos de gloria y de sufrimiento y me ha apoyado incondicionalmente y ha llenado mi vida de Amor.

Agradezco a mis Hijos Y. NahumMartínez V. y el que esta en camino. Quienes con su sonrisa me han dado aliento y fuerza para seguiradelante en todo momento.

Agradezco a mis Profesores(a todos en general) que compartieron conmigo sus conocimientos para hacer de mi un miembro de su ilustre sociedad y que fomentaron en mi una formación de profesionista integral.

Agradezco a la Universidad Nacional Autónoma de de México y a la Facultad de Ingeniería que abrieron sus puertas para recibirme y hacer de mi un cuidadano de provecho para poder integrarme a la sociedad.

Agradezco a mis Amigos(Santiago, Mireya, Alejandro, Jorge, Lidia, Miriam, Tania, Naxielli) quienes me han dado consejos y que me han estado apoyándome en cada etapa de mi vida.

A todas las Personas que han contribuido con un granito de arena para formar la persona que ahora soy.

Edgar Martínez Meza

ÍNDICE.

Capítulo 4
Lenguajes de Programación para Sistemas Expertos.

Capítulo 5
CLIPS.

Capítulo 6
Implementación del Sistema Experto R*obot.clp*

Capítulo 7
Implementación de la Base de Datos

Capítulo 8
Implementación de la Interfaz *CLIPS-Sybase.*

INTRODUCCIÓN

Desde que nació la Inteligencia Artificial como ciencia, a principios de los 60, hasta la actualidad, se han venido realizando numerosos trabajos para el desarrollo de los diversos campos que componen la Inteligencia Artificial. De todos esos campos, quizás el que puede tener mayor número de aplicaciones prácticas sea el de los Sistemas Expertos, siendo dichas aplicaciones de utilidad en temas tan variados que pueden ir desde la medicina hasta la enseñanza pasando por el Diseño Asistido por Computadora (CAD). En las empresas, los Sistemas Expertos empiezan a tener cada vez mayor auge, hasta el punto de llegar a ser un punto de referencia importante en la toma de decisiones para la junta directiva. En realidad, incluso se podría decir que el límite de las aplicaciones objeto de los Sistemas Expertos está en la imaginación humana, siendo siempre de utilidad ahí donde se necesite un experto.

Por otro lado, las Bases de Datos (BD) han sido grandemente utilizadas en campos en donde la información es simple de utilizar y manipular. En los últimos años han aparecido nuevas necesidades de incorporar en las Bases de Datos estructuras más complejas o aplicaciones. Algunos ejemplos de estas aplicaciones son: los Sistemas Expertos (SE), la Robótica, Procesamiento Digital de Imágenes (PDI), Medicina, Diseño Asistido por Computadora (CAD), etc. Todas estas aplicaciones necesitan contar con sistemas de almacenamiento en donde se asegure la integridad, la confidencialidad y todas las demás ventajas que provee un DBMS (*Data Base Management System*) o manejador de Bases de Datos, es debido a esta razón que dichas aplicaciones se han interesado por establecer comunicación con las Bases de Datos.

Existen muchas aplicaciones aparte del procesamiento de consultas a las Bases de Datos que pueden obtener provecho de ellas para otro tipo de aplicaciones. Entre dichas aplicaciones se encuentran las *Bases de Conocimientos*, las cuales constituyen parte de un sistema experto. En los Sistema Expertos (SE) la información se representa como hechos que se expresan en forma lógica. Este conjunto de hechos puede considerarse como una base de datos que contiene conocimientos, es decir una *Base de Conocimientos*.

Los primeros Sistemas Expertos no utilizaban un sistema de Base de Datos para manejar los datos, sino que empleaban técnicas apropiadas para la aplicación específica. Esto se considera un procedimiento aceptable, ya que el tamaño de las Bases de Conocimientos les permitía tener espacio suficiente en la memoria principal. Sin embargo, en los últimos años se ha hecho necesario desarrollar técnicas más complejas para la implantación de Bases de Conocimientos debido a que la aplicación de Sistemas Expertos a problemas más grandes o más complejos requiere de Bases de Conocimientos más extensas.

Tales consideraciones apuntan a la conveniencia de que la parte del sistema experto que lleva a cabo el procesamiento de las reglas, o sea la inferencia, intercambie información con un sistema estándar de Base de Datos, permitiendo así, que el sistema experto pueda aprovechar gran parte de la eficiencia de las técnicas de manejo de memoria y procesamiento de consultas de las Bases de Datos.

La principal aportación de la presente tesis consiste en el desarrollo de la interfaz *CLIPS-Sybase*, así como la implementación del sistema experto *robot.clp* y la base de datos *house*.

La Interfaz *CLIPS-Sybase*, la cual consiste en un programa escrito en C que utiliza las funciones de librerías del servidor Sybase, tiene como objetivo fundamental establecer comunicación entre un sistema experto hecho en CLIPS y un servidor Sybase de Base de Datos, logrando así satisfacer la necesidad de comunicación entre los Sistemas Expertos con un sistema de Bases de Datos y abriendo, al mismo tiempo, la posibilidad de llevar a cabo numerosas aplicaciones de los Sistemas Expertos como es el caso de la Robótica.

La razón por la cual la interfaz se realizó en lenguaje C y bajo un sistema operativo Linux se debe a que ésta es sólo un módulo de un proyecto más grande, el cual además de estar bajo un ambiente Linux, utiliza tanto CLIPS como el lenguaje C como herramienta para su desarrollo. Es por ello que se usó CLIPS para el desarrollo del sistema experto *robot*. Pero sobretodo, al utilizar lenguaje C se ahorran recursos en cuanto a necesidades de hardware, ya que a diferencia de otros lenguajes, como por ejemplo Java, el lenguaje C no necesita consumir muchos recursos en cuanto a hardware.

En cuanto a el servidor de base de datos utilizado para hacer la interfaz se contaban com más de una posibilidad: *Oracle, Postgres, Mysql, Informix* y *Sybase*.

A pesar de que *Oracle* es una base de datos tan potente como *Sybase*, *Oracle* no permite utilizar sus librerías para hacer programas de aplicación con ellas. Otra de las desventajas que se encontró al tratar de utilizar Oracle, fue que obtención del software de *Oracle* es más costoso que *Sybase*, por esta razón se descartó la utilización de *Oracle*.

Otro manejador de base de datos que se encuentra disponible para usarse en plataforma Linux es *postgres*, el cual es relativamente nuevo por lo que la información concerniente a este manejador de base de datos no se encuentra muy difundida aún.

Mysql también puede ser instalado en un ambiente Linux y con frecuencia es utilizado en el desarrollo de sitios WEB que interactuan con un sistema de base de datos, sin embargo Mysql tiene la limitante de soportar únicamente bases de datos pequeñas.

Informix es una opción más en cuanto a manejadores de bases de datos. Existe la posibilidad de realizar una interfaz que conecte una base de datos con un sistema experto utilizando un manejador de base de datos *Informix*, sin embargo en el desarrollo de la interfaz descrita en esta tesis se optó por usar un servidor *Sybase* debido a que éste además de permitir hacer la interfaz y ser un manejador de base de datos potente, se obtiene de manera fácil y totalmente gratuita. En contraste, los productos ofrecidos en el WEB por *Informix* son únicamente de evaluación, lo cual quiere decir que sólo pueden ser utilizados durante treinta días y después deben ser adquiridos pagando el monto correspondiente por las licencias del servidor y por los clientes que deseen conectarse.

Con la finalidad de poder probar el buen funcionamiento de la Interfaz *CLIPS-Sybase*, fue necesario hacer en CLIPS un sistema experto, al cual se llamó *robot.clp,* cuyo código fuente se anexa en el apéndice B, así como también fue necesario desarrollar una base de datos en Sybase a la cual se llamó *house.*

El sistema experto *robot.clp* además de ser diseñado para probar la funcionalidad de la Interfaz *CLIPS-Sybase*, también fue diseñado para que posteriormente sea usado para dar solución a un problema que se presenta en el campo de la robótica, lo cual ya no es parte de este trabajo.

Básicamente el sistema experto *robot.clp* simula un robot que se encuentra dentro de una casa, la cual está representada por la base de datos *house.* La tarea del sistema

experto consiste en determinar la ubicación de todas y cada una de las cosas contenidas dentro de la casa en cuestión, así como también la ubicación y características tales como acción, objetos que traen en las manos o ropa que están usando las personas que habitan dicha casa. De tal manera que el usuario puede cuestionar al sistema experto *robot* acerca de cualquier cosa que desee conocer respecto al contenido de la casa, como por ejemplo el usuario podría preguntarle al sistema experto: "*¿En donde se encuentra la mamá y que está haciendo?*", para poder responder dicha pregunta el sistema experto hace uso de la Interfaz *CLIPS-Sybase*, ya que es en la Base de Datos *house* en donde se encuentra almacenada toda la información concerniente con el contenido de la casa.

La presente tesis se encuentra organizada de la siguiente manera: los primeros tres capítulos se encargan de dar a conocer las definiciones, terminología y conceptos fundamentales de los Sistemas Expertos, los cuales son esenciales para comprender qué es un sistema experto y cómo opera, pero sobretodo, fueron una parte esencial para la construcción del sistema experto *robot.clp*.

Una vez que se definen los conceptos básicos de los Sistemas Expertos que permiten tener un entendimiento claro de los mismos, se explica el proceso de extracción, análisis y representación del conocimiento se identifica la necesidad de codificar dicho conocimiento por medio de un lenguaje de programación de tal forma que pueda ser almacenado y procesado en una computadora. Es por ello, que el capítulo cuatro se encarga de discutir algunos de los lenguajes de programación de Inteligencia Artificial (o de manipulación simbólica) más importantes: PROLOG, LISP y CLIPS. Asimismo, se mencionan las ventajas que tiene CLIPS, las cuales fueron decisivas en la elección del lenguaje de programación para el desarrollo del sistema experto.

En el capítulo cinco se describe el lenguaje de programación: CLIPS (*C Language Integrated Production System*), el cual fue desarrollado en la NASA *(National Aeronautics & Space Administration)*. En dicho capítulo se describen los elementos básicos de CLIPS y sus principales características, asimismo se describen los aspectos prácticos, como son los comandos más importantes de dicha herramienta utilizados para el desarrollo del sistema *robot*.

En el capítulo seis se describe el desarrollo y la implementación del sistema experto *robot*. Uno de los objetivos del sistema experto *robot*, es aplicar tanto las funciones ya existentes de CLIPS como las nuevas funciones provistas por la interfaz *CLIPS-Sybase*, entre otras cosas para comprobar el buen funcionamiento de las mismas y para posteriormente, como ya se explicó antes, resolver un problema que se presenta en robótica.

En el capítulo siete se da una breve introducción a los conceptos fundamentales de las Bases de Datos, así como también se describe la estructura de la base de datos *house*. En la parte final de dicho capítulo se mencionan las características de Sybase que permitieron la realización de la Interfaz *CLIPS-Sybase*, y que fueron la razón por la cual se optó por utilizar Sybase en lugar de algún otro sistema manejador de base de datos.

Finalmente, en el capítulo ocho se presenta una descripción detallada del desarrollo de la Interfaz *CLIPS-Sybase*, la cual es la parte medular y más importante de la tesis , como ya se mencionó antes, consiste en establecer una conexión entre un servidor SQL Sybase y el sistema experto desarrollado en CLIPS *Robot*.

CAPÍTULO 1

SISTEMAS EXPERTOS

1.1 Generalidades.

El presente capítulo, así como los dos siguientes capítulos, tienen como principal finalidad dar a conocer las definiciones, terminología y conceptos fundamentales de los Sistemas Expertos. Dichas definiciones, terminologías y conceptos, son esenciales para comprender qué es un sistema experto y cómo opera.

1.1.1 Definición de un Sistema Experto.

Los Sistemas Expertos son utilizados para realizar una variedad de tareas extremadamente complicadas, que en el pasado sólo podían ser realizadas por un número limitado de expertos humanos altamente capacitados. A través de la aplicación de técnicas de la *Inteligencia Artificial*, tales como *la representación del conocimiento y los métodos de inferencia,* los Sistemas Expertos capturan el conocimiento básico que le permite a un humano actuar como un experto a la hora de enfrentarse con problemas complicados.

Dado que los Sistemas Expertos son una rama de la Inteligencia Artificial(IA), es importante dar respuesta a la pregunta "¿Qué es la Inteligencia Artificial?". A continuación se presenta una definición general de la Inteligencia Artificial.

> "La Inteligencia Artificial es una solución basada en la computación a problemas complejos a través de la aplicación de procesos que son análogos al proceso de razonamiento humano."[1]

[1] Rolston David W, *Principles of Artificial Intelligence and Expert Systems Development.* Edit. McGraw-Hill. 1988. pag. 15

Es posible definir a un sistema experto como "un programa de computación inteligente que usa *la representación del conocimiento* y *métodos de inferencia* para resolver problemas que son bastante difíciles y requieren de pericia humana significativa para su solución".[2]

Un sistema experto es un sistema de computación que *emula* el comportamiento y la habilidad de toma de decisiones de expertos humanos a la hora de resolver problemas. El término *emular* significa que el sistema experto está diseñado para actuar en todo como un experto humano. Una emulación es mucho más poderosa que una simulación, la cual sólo requiere actuar como el objeto real en sólo algunos aspectos, mientras que en una emulación se trata de abarcar todos los aspectos.

El término de sistema experto es con frecuencia aplicado a cualquier sistema que use tecnología de sistema experto. Esta tecnología podría incluir algunos lenguajes especiales de Sistemas Expertos, programas y hardware diseñado para ayudar en el desarrollo y ejecución de los Sistemas Expertos.

Los términos sistema experto, sistema basado en conocimientos y Sistemas Expertos basados en conocimientos son usados como sinónimos.

---

[2] Edward A. Feingenbaum, *Knowledge Engineering in the 1980's*, Dept. of. Computer Science, Stanford University, Stanford C.A, 1982.

1.1.2 Características de los Sistemas Expertos.

La principal característica de los Sistemas Expertos es su capacidad de enfrentarse a problemas del mundo real a través de procesos que reflejan tanto el razonamiento como la intuición de un humano.

Idealmente un sistema experto debe poseer las siguientes características.

- Conocimiento extensivo y específico del *dominio* de interés.
- Capacidad para inferir nuevo conocimiento a partir del conocimiento existente.
- Procesamiento *simbólico.*
- Habilidad para explicar su propio razonamiento.[3]

1.2 Papel del Conocimiento en los Sistemas Expertos.

El conocimiento, y no simplemente los hechos, es el material primario de los Sistemas Expertos, puesto que el poder de un experto humano se deriva del dominio extenso del conocimiento específico un sistema experto, debe contar de igual manera con un dominio extenso a cerca de cierto tema.

Existen muchos componentes del conocimiento que son la fuente de la habilidad de un experto humano para ejecutar determinada acción. Dichos componentes podrían resumirse de la siguiente forma:

- **Hechos.**

Los hechos son declaraciones que relacionan algún elemento de verdad observado en algún objeto. Por ejemplo:

"El cielo es azul"

---

[3] *ibidem.* Pag. 2.

- **Reglas procedurales.**

Bien definidas, las reglas invariantes describen secuencias fundamentales de eventos y relaciones. Por ejemplo:

"SI la luz está en rojo ENTONCES deténgase"

- **Reglas heurísticas.**

Son reglas generales, las cuales son aproximadas, ya que están basadas en la experiencia que un experto ha adquirido a través del tiempo. Dichas reglas sugieren los procedimientos que deben ser seguidos cuando las reglas procedurales invariantes no están disponibles. Un ejemplo de este tipo de reglas es el siguiente:

"Es mejor intentar un aterrizaje de emergencia bajo condiciones controladas que continuar el vuelo bajo condiciones desconocidas."

La presencia de la heurística contribuye grandemente al *poder* y la *flexibilidad* de los Sistemas Expertos y tiende a distinguirlos del software tradicional. Además de las formas específicas de conocimiento mostradas anteriormente un experto humano también tiene un *modelo general conceptual* del dominio de un tema y un *esquema general* para encontrar una solución.

1.3 Arquitectura de los Sistemas Expertos.

Existe más de una arquitectura para los Sistemas Expertos, la mayoría de las cuales tienen varios componentes generales en común. En la figura 1-1 se muestra una arquitectura general, la cual muestra los componentes típicos. A continuación se brinda una explicación para cada uno dichos componentes.

- **Usuario.**

El usuario, es la persona que opera en cualquiera de las siguientes formas al sistema experto:

1. *Examinador.* Es el usuario que verifica la validez del sistema
2. *Tutor.* Es el usuario que provee información adicional al sistema o modifica el conocimiento presente en el sistema.
3. *Pupilo.* Es el usuario que busca un rápido desarrollo de pericia en el área de dominio del sistema experto, mediante la extracción de conocimiento del sistema.
4. *Cliente.* Es el usuario que aplica la pericia del sistema experto para realizar tareas específicas.

- **Interfaz de usuario.**

Es el mecanismo por medio del cual el usuario se comunica con el sistema.

- **Memoria Global de Trabajo.**

La memoria global de trabajo o *base de hechos,* alberga los datos propios del problema que se desea tratar con la ayuda del sistema. Asimismo a pesar de ser la memoria de trabajo, la base de hechos puede desempeñar el papel de memoria auxiliar. La memoria de trabajo memoriza todos los resultados intermedios, permitiendo conservar el rastro de los razonamientos llevados a cabo anteriormente. Por ello, puede emplearse para explicar el origen de las conclusiones hechas por el sistema en el transcurso de una sesión de trabajo o para llevar a cabo la descripción del comportamiento del propio sistema experto. Al principio del período de trabajo, la base de hechos dispone únicamente de los datos que le ha introducido el usuario del sistema, pero, a medida que va actuando la máquina de inferencias, contiene las cadenas de inducciones y

deducciones que el sistema forma al aplicar las reglas para obtener las conclusiones buscadas.

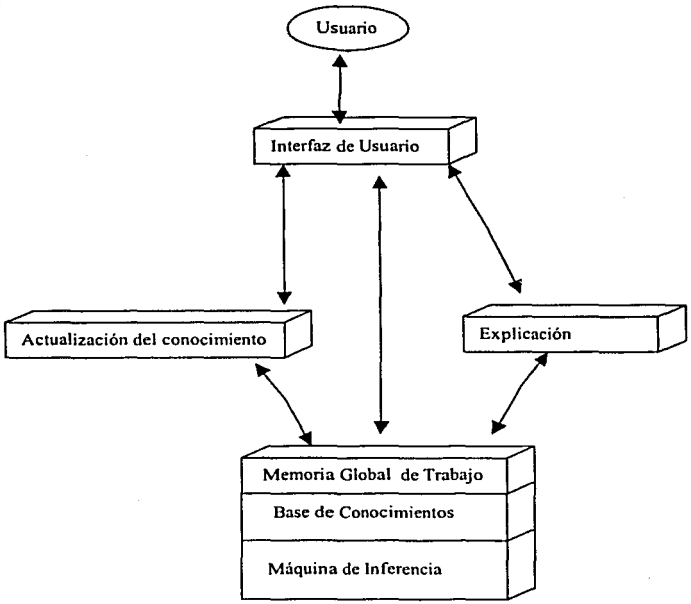

Figura 1-1. Arquitectura típica de un Sistema Experto[4]

- **Base de Conocimientos.**

La base de conocimientos representa un depósito del *conocimiento primitivo,* es decir, de las reglas procedurales que están disponibles en el sistema.

La base de conocimientos contiene el conocimiento necesario para la comprensión, formulación y solución de un problema. Dicha base de conocimientos está

[4] Figura tomada de: Rolston, David W. *Principles of Artificial Intelligence and Expert Systems Development.* Edit. McGraw Hill 1988. pag. 7

constituida por las reglas tanto procedurales como heurísticas, las cuales fueron definidas en la sección 1.2.

La información es incorporada a la base de conocimientos por medio de un programa hecho en algún lenguaje de computación por medio de un proceso llamado *representación del conocimiento*, el cual será discutido en el capítulo dos.

- **Máquina de Inferencias.**

La máquina de inferencias es un sistema de software que localiza el conocimiento que se encuentra en la base de conocimientos y hace inferencias sobre el mismo para generar nuevo conocimiento a partir del que se encuentra contenido en dicha base de conocimientos.

El *paradigma de inferencia* de la máquina es la estrategia de búsqueda usada para desarrollar el conocimiento requerido. La mayoría de los métodos usados por los Sistemas Expertos para desarrollar el conocimiento se basan en uno de los dos conceptos fundamentales:

- *Encadenamiento hacia atrás,* el cual es un proceso de deducción de tipo *top-down*, es decir, empieza a deducir a partir del objetivo o conclusión para llegar a encontrar las condiciones necesarias para dicha conclusión.

- *Encadenamiento hacia delante*, el cual es un proceso de razonamiento de tipo *bottom-up* que empieza con las condiciones iniciales y trabaja hacia el objetivo deseado o conclusión.

Para explicar la actualización del conocimiento es necesario introducir el concepto de *Ingeniería del Conocimiento.*

La *Ingeniería del Conocimiento* es el proceso mediante el cual se obtiene el conocimiento de un dominio específico y se implementa dicho conocimiento dentro de la base de conocimientos. En la figura 1-2 se ilustra la manera en que típicamente ocurre este proceso. La mayoría de las veces el conocimiento es obtenido de expertos humanos. El conocimiento que es proveído por un experto generalmente estará orientado al área de dominio.

Un *ingeniero del conocimiento* es la persona que obtiene el conocimiento de un experto humano para posteriormente trasladarlo a la una base de conocimientos.

Para adquirir el conocimiento necesario, el ingeniero del conocimiento primero debe establecer un total entendimiento del área de dominio del experto humano, formar un vocabulario esencial del área de dominio y desarrollar un entendimiento fundamental de los conceptos claves.

La función de la adquisición del conocimiento es con frecuencia el aspecto más difícil del desarrollo de un sistema experto. Esto se debe a que el proceso requiere un grado elevado de comunicación humana entre el ingeniero del conocimiento y el experto en determinada área, por lo tanto se sufren los problemas asociados a la comunicación humana.

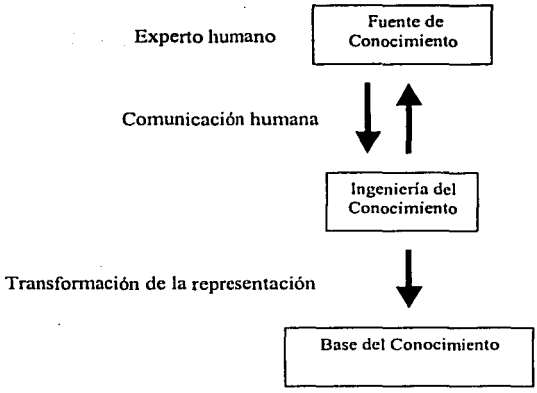

Figura 1-2. Ingeniería del Conocimiento[5]

- **Actualización del Conocimiento.**

La actualización del conocimiento es el proceso mediante el cual se hacen las modificaciones correspondientes en la base de conocimientos, ya que ésta refleja el conocimiento de un área en el tiempo en que el sistema fue puesto en servicio, debido a que el conocimiento en muchas áreas está en constante cambio y expansión, es necesario actualizar el conocimiento representado en la base de conocimientos. Dicho proceso de actualización puede ser llevado a cabo tomando una de las tres formas básicas mostradas a continuación.

1. *Actualización manual del conocimiento.* En éste caso la actualización se realiza por un ingeniero del conocimiento, quien se encarga de interpretar la información provista por el experto humano y actualiza la base de conocimientos.

[5] Tabla tomada de:Turban, Efraim, *Expert Systems and Applied Artificial Intelligence*. Edit. McMillan.1992. pag. 8

2. La segunda forma de actualización representa el estado del arte en Sistemas Expertos, ya que el experto humano introduce el conocimiento directamente en la base de conocimientos, sin la mediación del ingeniero del conocimiento. En este caso la actualización del sistema debe ser mucho más sofisticada.

3. *Actualización por aprendizaje de la máquina.* En este tipo de actualización el sistema genera nuevo conocimiento generado de conclusiones hechas a partir de experiencias pasadas. El sistema aprende de sus experiencias e idealmente se actualiza a sí mismo.

- **Explicación.**

Un sistema experto debe ser diseñado para tener la capacidad de explicar el razonamiento que lo conduce a cierta conclusión cuando se enfrenta a un problema. La explicación típicamente consiste en la identificación de los pasos del proceso de razonamiento, es decir, consiste en dar a conocer las razones por las cuales se llega a determinada conclusión, por ejemplo, en el caso de un sistema experto en vuelos, el cual podría llegar a la conclusión de que el avión debe aterrizar. Una explicación de dicha conclusión podría ser: "está encendida una luz de emergencia, la cual indica una falla en el motor y no es posible desactivar dicha luz de emergencia, por lo tanto es necesario aterrizar".

En realidad, la facilidad de explicación en muchos Sistemas Expertos está limitada a simplemente listar las reglas que fueron empleadas durante la ejecución del sistema.

1.4 Proceso de Desarrollo de los Sistemas Expertos.

El proceso de desarrollo de un sistema experto consta de varios estados básicos, los cuales son similares a los estándares de la Ingeniería de Software para el ciclo de vida del software. Dichos estados se listan a continuación.

1. Identificación del problema.
2. Construcción de prototipo.
3. Formalización.
4. Implementación
5. Evaluación.
6. Evolución.

El primer estado del desarrollo de cualquier sistema experto es establecer si el problema que se desea resolver necesita ser solucionado por un sistema experto. Si el problema en consideración puede ser descrito en términos de definiciones directas y algoritmos, es probable que deba dar una solución mediante el desarrollo de software tradicional. Si para dar solución al problema se requiere de un razonamiento humano intenso, entonces se le puede dar solución al problema por medio de un sistema experto.

El segundo paso consiste en la construcción de un pequeño prototipo para ayudar a comprender el problema y estimar la construcción de la solución completa. El siguiente paso en el desarrollo de un sistema experto es la *formalizar* la declaración del problema y diseñar el sistema experto completo. Se continúa después con la *implementación*, la cual consiste en continuar con el ciclo de adquisición del conocimiento, actualización de la base del conocimiento y la realización de pruebas.

La fase de evaluación tiene como finalidad evaluar hasta que punto el sistema se aproxima a la naturaleza del experto humano. Después de la fase de evaluación, el sistema entra en un periodo de evolución. Durante dicho período, el sistema continúa creciendo dependiendo de los cambios sufridos por el área del conocimiento dominado por el sistema experto.

1.5 Aplicaciones de Sistemas Expertos

Los Sistemas Expertos pueden ser clasificados en varias formas. Una de ellas es haciendo una clasificación genérica en la cual se diferencien las aplicaciones de los Sistemas Expertos en distintas áreas. En la tabla 1-1 se listan las categorías genéricas de los Sistemas Expertos. En algunos Sistemas Expertos pertenecen a más de una categoría. A continuación se presenta una breve descripción para cada una de dichas categorías.

- **Sistemas de Interpretación.**

Infieren una descripción de la situación partiendo de las observaciones hechas. En esta categoría se incluyen la comprensión del habla, análisis de imágenes, interpretación de señales, y muchos otros tipos de análisis de la inteligencia. Un sistema de interpretación explica los datos observados asignándoles significado simbólico a dichos datos describiendo así cierta situación.

- **Sistemas de Predicción.**

Este tipo de sistema incluye pronósticos del tiempo, predicciones demográficas, pronósticos económicos, predicciones de tráfico y pronósticos militares y financieros.

- **Sistemas de Diagnóstico.**

Se incluye software para el diagnóstico médico, electrónico y mecánico. Los sistemas de diagnóstico típicamente relacionan la naturaleza de las irregularidades observadas a causas fundamentadas.

- **Sistemas de Diseño.**

Desarrollan la configuración de objetos que satisfaga las condiciones del diseño del problema. Tales problemas incluyen diseño de construcción, distribución de circuitos y distribución de una planta. Este tipo de sistemas construyen descripciones de objetos y la forma en que éstos se relacionan con otros y también verifican que ésta configuración cumpla con las condiciones establecidas.

- **Sistemas de Planeación.**

Se especializan en problemas tales como la programación automatizada. También pueden tratar problemas relacionados en áreas tales como la administración de proyectos, comunicaciones, desarrollo de productos, aplicaciones militares y planeación financiera.

- **Sistemas de Vigilancia.**

Comparan las observaciones de la naturaleza de un sistema contra los estándares cruciales para la realización de las metas. Un sistema para el tráfico aéreo estaría dentro de esta categoría.

- **Sistemas de Supresión de Errores.**

Dependen de las capacidades de planeación, diseño y predicción para crear recomendaciones para corregir el problema diagnosticado.

- **Sistemas de Reparación.**

Desarrollan y ejecutan planes para administrar un remedio para el problema diagnosticado. Tales problemas incorporan la supresión de errores, la planeación y la ejecución de estas capacidades.

- **Sistemas de Instrucción.**

Se incorporan subsistemas de diagnóstico y de supresión de errores, los cuales son aplicados a un punto específico de interés del estudiante. Estos sistemas típicamente diagnostican las debilidades del conocimiento del estudiante e identifican las soluciones apropiadas para superar dichas deficiencias. Finalmente planean una interacción de tipo tutor con la finalidad de transmitir el conocimiento al estudiante.

- **Sistemas de Control.**

Gobiernan por completo la naturaleza del sistema. Para hacer esto, el control del sistema debe repetidamente interpretar la situación actual, predecir el futuro, diagnosticar las causas de problemas anteriores, formular un plan de soluciones y vigilar su ejecución para asegurar el éxito.

Las categorías descritas anteriormente tienen la finalidad de dar a conocer el tipo de tareas que pueden ser realizadas por los Sistemas Expertos.

Clase	Área General
Interpretación	Explicación de datos observados
Predicción	Inferir probables consecuencias a partir de situaciones dadas
Diagnóstico	Inferir mal funcionamiento en un sistema a partir de observaciones
Diseño	Ensamblar los componentes de una manera correcta.
Planeación	Desarrollar planes para alcanzar los objetivos deseados
Vigilancia	Compara datos observados de datos esperados
Supresión de Errores	Prescribir remedios a los malos funcionamientos del sistema.
Reparación	Ejecutar un plan para administrar el remedio prescrito.
Instrucción	Diagnosticar, suprimir errores y corregirlos
Control	Interpretación, predicción, reparación y vigilancia de la naturaleza del sistema

Tabla 1-1. Categorías genéricas de los Sistemas Expertos.[6]

[6] tabla tomada de Turban, Efraim, *Expert Systems and Applied Artificial Intelligence*. Edit. McMillan. 1992. pag. 92

1.5.1 Robótica.

Aunque no se menciona en la tabla 1-1, la robótica es otra aplicación de los Sistemas Expertos. Esta aplicación es de suma importancia para el presente trabajo, ya que el sistema experto *Robot.clp*, cuya descripción y desarrollo son expuestos en el capítulo seis (implementación del sistema experto *Robot.clp*), fue desarrollado para ser aplicado dentro del campo de la robótica, dado que la robótica para poder desarrollar un robot se sirve de disciplinas como la mecánica, la microelectrónica y la informática, es importante mencionar que el sistema experto desarrollado en el presente trabajo representa únicamente la parte informática del desarrollo de un robot. Por ello a continuación se muestran algunos conceptos generales relacionados con la robótica.

El término *robot* es aplicado a todos los mecanismos, accionados y controlados electrónicamente, capaces de llevar a cabo secuencias simples que permiten realizar operaciones tales como carga y descarga, accionamiento de máquinas herramienta, operaciones de ensamblaje y soldadura, etc. Hoy en día el desarrollo en este campo se dirige hacia la consecución de máquinas que sepan interactuar con el medio en el cual desarrollan su actividad (reconocimientos de formas, toma de decisiones, etc.).

La disciplina que se encarga del estudio y desarrollo de los robots es la robótica. El creciente desarrollo de los robots y su constante perfeccionamiento ha hecho que cada día se apliquen en mayor medida a los procesos industriales en sustitución de la mano de obra humana. Dicho proceso, que se inició hacia 1970, recibe el nombre de robotización y ha dado lugar a la construcción de plantas de montaje parcial o completamente robotizadas. Este proceso conlleva, según sus detractores, la destrucción masiva de

puestos de trabajo, mientras que para sus defensores supone la satisfacción de necesidades socioeconómicas de la población y lleva aparejado un aumento muy considerable de la productividad.

1.5.2 La Robótica dentro de la Inteligencia Artificial

Los robots experimentales creados en Inteligencia Artificial(IA) eran automatizaciones capaces de recibir información procedente del mundo exterior (p. ej.., sensores, cámaras de televisión, etc.), así como órdenes de un manipulador humano (expresadas en lenguaje natural). De este modo, el robot determinaba un plan y, de acuerdo con él, ejecutaba las órdenes recibidas mediante el empleo de un modelo del universo en el que se encontraba. Era incluso capaz de prever las consecuencias de sus acciones y evitar, así, aquéllas que más tarde pudieran resultarle inútiles o, en algún momento, perjudiciales. Estos primeros robots experimentales eran bastante más inteligentes que los robots industriales, y lo eran porque disponían de un grado mucho mayor de percepción del entorno que los robots empleados en las cadenas de producción.

El principal problema con el que se enfrenta la Inteligencia Artificial aplicada a los robots es el de la visión. Mientras que la información recibida a través de sensores se puede interpretar con relativa facilidad y entra a formar parte de la descripción del modelo de universo que emplea el robot para tomar decisiones, la percepción de las imágenes captadas y su interpretación correcta es una labor muy compleja. En cuanto a la interpretación de las imágenes captadas mediante cualquier sistema, se ha logrado ya el reconocimiento de formas preprogramadas o conocidas, lo que permite que ciertos robots lleven a cabo operaciones de reubicación de piezas o colocación en su posición correcta a partir de una posición arbitraria. Sin embargo, no se ha logrado aún que el sistema perciba la imagen tomada mediante una cámara de ambiente y adapte su actuación al

nuevo cúmulo de circunstancias que esto implica. Por ejemplo, la imagen de una cámara de vídeo de las que se emplea en vigilancia y sistemas de seguridad no puede ser interpretada directamente por la computadora.[7]

1.6 Desarrollo Histórico de Tecnología de Sistemas Expertos.

Como se puede observar en la tabla 1-2, en donde se muestran los avances en forma cronológica que se han logrado en desarrollo de tecnología de los Sistemas Expertos, el desarrollo de dicha tecnología empezó a darse a partir de los años 40, pero no fué sino hasta mediados de los años 70 cuando el primer sistema experto, MYCIN, llegó a funcionar con la misma calidad que un experto humano, dando a su vez explicaciones a los usuarios sobre su razonamiento. Antes del desarrollo de MYCIN (mediados de los 70), se criticaba a la Inteligencia Artificial (IA) de resolver únicamente problemas "de juguete", sin embargo, el éxito de MYCIN demostró que la tecnología de los sistemas expertos estaba suficientemente madura como para salir de los laboratorios y entrar en el mundo comercial. MYCIN es, en definitiva, un sistema de diagnóstico y prescripción en medicina, altamente especializado, diseñado para ayudar a los médicos a tratar con infecciones de meningitis y bacteriemia. Dichas infecciones pueden ser fatales y a menudo es necesario detectarlas con tiempo por lo que un sistema experto que ayude a detectar el mal por los síntomas es de gran ayuda. A continuación se describirán brevemente algunos de los avances tecnológicos más importantes de los que se muestran en la tabla 1-2.

- **1943 Sistemas de Postproducción.**

La idea básica es que cualquier sistema lógico o matemático es un conjunto simple de reglas que especifican cómo cambiar una cadena de símbolos en otro conjunto de

[7] Información tomada de: *http://www.cinefantastico.com/nexus7/ia/robots.htm.*

símbolos, es decir, a un antecedente. Una regla de producción puede realizar un consecuente. Esta idea es válida para programas y Sistemas Expertos en donde la cadena inicial de símbolos es un dato de entrada y la salida es una transformación de la entrada.

- **1954 Algoritmo de Markov.**

El siguiente avance en la aplicación de las reglas fué hecha por Markov, ya que especificó una estructura de control para sistemas de producción. Dicha estructura de control consta de un grupo ordenado de producciones que son aplicadas en orden de prioridad sobre una cadena de entrada. Si la cadena de mayor prioridad no es aplicable, entonces se aplica la siguiente y así sucesivamente.

- **1958 LISP.**

LISP (List Processor), es un lenguaje de propósito general de los más viejos, fue desarrollado en el laboratorio de Inteligencia Artificial(IA) MIT por McCarthy en 1958 y aún es utilizado. Dentro de las aplicaciones de LIPS se incluyen: Sistemas Expertos, el procesamiento del lenguaje natural, robótica y programación psicológica y educacional. LISP posee la característica de darle al programador el poder de desarrollar software que va más allá de las limitaciones de otros lenguajes de propósito general tales como COBOL y Pascal.

Para desarrollar un sistema experto usando LISP, es necesario contar con una estación de trabajo de Inteligencia Artificial llamada "máquina Lisp". A mediados de los años setenta se desarrolló una máquina especializada para soportar la ejecución de programas hechos en LISP. Desde entonces, han sido desarrollados varios sistemas comerciales.

Los Sistemas Expertos más grandes y más sofisticados tradicionalmente han sido desarrollados en el lenguaje LISP, es decir, en máquinas Lisp especializadas debido a que para poder ejecutar dichos Sistemas Expertos es necesario el poder y la capacidad del hardware y el ambiente para desarrollo de software que es provisto por las máquinas Lisp.[8]

- **1970 PROLOG**

La definición del lenguaje PROLOG (Programación Lógica), se basa en la lógica formal. PROLOG permite ejecutar estatutos que no son otra cosa que oraciones de un lenguaje lógico elemental. El tema de PROLOG será abordado nuevamente en el capítulo de lenguajes (capítulo cuatro) del presente trabajo.

- **1979 El Algoritmo RETE**

A pesar de que el algoritmo de Markov puede ser usado como la base de un sistema experto, resulta ser ineficiente para sistemas con muchas reglas. El algoritmo (RETE) resuelve el problema presentado por el algoritmo de Markov cuando se tiene un sistema experto con cientos o miles de reglas, ya que el algoritmo RETE no accede en forma secuencial a las reglas. El algoritmo RETE es más rápido que el algoritmo de Markov debido a que la información de las reglas es almacenada en la memoria de trabajo, en lugar de cotejar los hechos con los antecedentes, ya que los datos estáticos no cambian ciclo tras ciclo y por lo tanto son ignorados.

---

[8] Rolston David W, *Principles of Artificial Intelligence and Expert Systems Development.* Edit. McGraw-Hill. 1988. pag. 173-174.

- **1985 CLIPS *(C Language Integrated Production System).***

La tecnología para el desarrollo de Sistemas Expertos en los años ochenta era demasiado costosa. Este problema se ve disminuido con la introducción del poderoso software desarrollado por la NASA llamado CLIPS, el cual es descrito con mayor detalle en el capítulo 5.

CLIPS está escrito en lenguaje C por velocidad y portabilidad, además usa el poderoso reconocedor de patrones llamado Algoritmo RETE. Otra gran ventaja de CLIPS es que las 25,000 líneas de código fuente de CLIPS son totalmente gratuitas, dicho código fuente es totalmente abierto para que el programador pudiese adecuarlo a sus necesidades.

La ventaja al usar un compilador C es que sus requerimientos en cuanto hardware son muy pocos, ya que es posible instalarlo en un máquina con un procesador 386 en adelante y además puede ser instalado en computadoras con las plataformas: Windows, UNIX y Linux.

Debido a las ventajas de CLIPS de portabilidad, velocidad y bajo costo se eligió CLIPS para el desarrollo de la interfaz descrita en el capítulo 8 del presente trabajo. Además, se eligió por ser un software abierto, ya que permite modificar su código fuente.

Año	Eventos
1943	Reglas de postproducción; McCulloch y Pits con su modelo neuronal.
1954	Algoritmo de Markov para controlar ejecución de reglas.
1956	Conferencia de Darthmouth: Lógica. Búsqueda heurística.
1957	Perceptron inventado por Rosenblatt; GPS (General Problem Solver).
1958	LISP Lenguaje de inteligencia artificial (McCarthy).
1962	Principios de neurodinámica de Rosenblatt.
1965	Método de resolución automático de problemas, lógica difusa para razonar con objetos difusos. Trabajo con DENDRAL.
1968	Redes semánticas, modelo de memoria asociativa (Quilian).
1969	MACSYMA (experto de sistemas matemáticos.
1970	Trabajo empieza con PROLOG.
1971	HEARSAY I para reconocimiento de habla.
1973	MYCIN, sistema experto para medicina, líder de GUIDON Y TEIRESIAS, facilidad para explicación, así como HEARSAY II para multi-cooperación entre sistemas expertos.
1975	Representación del conocimiento por FRAMES.
1976	AM (matemático artificial, descubrimiento creativo de conceptos matemáticos (Lenat) El trabajo comienza con PROSPECTOR para la explotación mineral.
1977	La shell del sistema experto OPS usado en XCON/R1.
1978	Trabajo continuado con XCON/R1 con DEC. Trabajo para metareglas e inducción de reglas con DENDRAL.
1979	El algoritmo Rete para el reconocimiento de patrones. La comercialización de la IA empieza, se forma Inference Corp.
1980	Symbolics,LMI fundado para construir maquinas LISP.
1982	El sistema experto matemático SMP, la red neuronal de Hopfield.
1983	Herramienta de sistema experto KEE.
1985	Herramienta sistema experto CLIPS (NASA).

Tabla 1-2.
Algunos eventos importantes en la historia de los Sistema Expertos[9]

[9] Tabla tomada de Giarratano, Joseph. ed. al. *Expert Systems: principles and programmingn*. Pag. 12

CAPÍTULO 2

REPRESENTACIÓN DEL CONOCIMIENTO.

2.1 Introducción.

Como se mencionó en el capítulo anterior, el conocimiento es el elemento fundamental que determina el poder de un sistema experto. Sin embargo, es necesario que sea capaz de representar el conocimiento de manera abstracta para posteriormente poder utilizarlo en el proceso del deducción del sistema. Es por ello, que en el presente capítulo se presentan las bases de la *representación del conocimiento*. Asimismo, serán descritos la técnica de representación del conocimiento y el esquema o código de representación usados para la representación del conocimiento del sistema experto, *Robot.clp*, desarrollado en esta tesis.

La representación del conocimiento consta de dos partes: aquellas representaciones que son útiles para el análisis y aquellas que son usadas en la codificación, a su vez estos dos tipos de representación son parte del proceso de Ingeniería del Conocimiento, el cual es el proceso previo a la implementación del sistema mediante un lenguaje de programación para Sistemas Expertos. El proceso de implementación del sistema desarrollado en esta tesis se explica en el capítulo seis, mientras que el método de representación del conocimiento es presentado en el presente capítulo y el método de inferencia se explicará en el capítulo tres.

2.2 Significado de Conocimiento

El conocimiento puede ser clasificado como *conocimiento procedural* y *conocimiento declarativo.* A continuación se describe brevemente a cada uno de ellos.

- **Conocimiento Procedural.** Se refiere a el procedimiento que se debe seguir para llevar a cabo algo.
- **Conocimiento Declarativo.** Se refiere a determinar si una afirmación es verdadera o falsa.
- **Conocimiento Implícito.** Es el conocimiento que no puede ser expresado mediante algún lenguaje. Por ejemplo, el conocimiento que una persona posee para poder parpadear es algo que no puede explicar, ya que la persona simplemente sabe parpadear aunque no puede explicar cómo es que lo sabe.

El conocimiento es de primera importancia en los Sistemas Expertos. De hecho, haciendo una analogía con la clásica expresión para los programas:

Algoritmos + Estructuras de datos = Programas

Para un sistema experto es:

Base de Conocimientos + Máquina de Inferencias = Sistemas Expertos

El conocimiento es parte de una jerarquía, ilustrada en la figura 2-1. En la base de la pirámide está el ruido, que consiste de arreglos sin interés que obscurece los datos. En el siguiente nivel se encuentran los datos, que son arreglos de potencial interés. Después de ser procesados los datos se convierten en información. En seguida está el conocimiento, el cual representa información muy especializada.

El término "hecho" puede significar dato o información. Los Sistemas Expertos pueden hacer inferencias usando datos o información, separar datos del ruido, convertir datos en información o transformar datos e información en conocimientos.

La pericia es un tipo especializado de conocimiento que los expertos humanos poseen. La pericia no es comúnmente encontrada en fuentes públicas de información tales como libros y revistas. La pericia es el conocimiento implícito de los expertos, el cual debe ser extraído para posteriormente convertirlo en conocimiento explícito de tal forma que pueda ser codificado dentro de un sistema experto.

El conocimiento de los expertos humanos debe formularse de tal forma, que pueda ser almacenado en computadoras y procesado y analizado por programas. Esta imagen del conocimiento recibe el nombre de *Representación del Conocimiento*.

En el último nivel de la pirámide esta el metaconocimiento, el significado del prefijo meta quiere decir arriba. Así que el metaconocimiento es el conocimiento del conocimiento y la pericia. Un sistema experto podría ser diseñado con conocimiento sobre diferentes dominios. El metaconocimiento especificaría cuál base de conocimientos sería aplicable. También podría ser usado dentro de un dominio para decidir qué grupo de reglas en ese dominio son las más aplicables.

En un sentido filosófico, la sabiduría es la cumbre del conocimiento. De una manera análoga se podría decir que la sabiduría es el metaconocimiento para determinar las mejores metas en la vida y cómo obtenerlas.

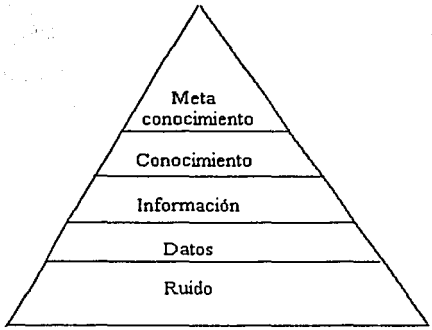

Figura 2-1. Jerarquía del conocimiento.[10]

2.3 Métodos de Representación del Conocimiento No Formales.

En general existen dos tipos de representación del conocimiento: aquellas representaciones que son útiles para el análisis y aquellas que son usadas en la codificación, las primeras serán presentadas en esta sección. En la figura 2-2 se muestra la relación entre estos dos tipos de representación y el resto del proceso de la ingeniería del conocimiento.

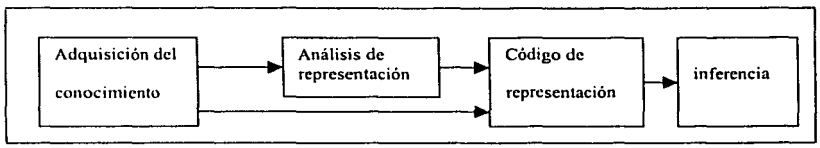

Figura 2-2. Técnicas de la representación del conocimiento.[11]

[10] Giarratano, Joseph. ed. al. *Expert Systems: principles and programming* pag. 65.
[11] Figura tomada de: Turban, Efraim, *Expert Systems and Applied Artificial Intelligence*. Edit. McMillan. 1992. Pag. 175

El código de una base de conocimientos es representado ya sea por medio de reglas de producción o por medio de *frames,* ambas formas de representar el conocimiento serán presentados en las secciones 2.5 y 2.6 respectivamente.

El conocimiento capturado a través de cualquiera de las técnicas de análisis puede ser fácilmente convertido en reglas o en *frames*, ya que existen herramientas de software que convierten las tablas o árboles de decisión en reglas, un ejemplo de dichas herramientas es CLIPS. En la siguiente sección se mencionan para qué sirven las técnicas de análisis y cuál técnica se usó para el sistema experto *robot.clp*.

2.3.1 Técnicas de Análisis.

Las técnicas de análisis del conocimiento son usadas para ayudar en la adquisición del conocimiento, ya que ayudan a establecer y reunir el conocimiento inicial. La mayoría de éstas técnicas son pictóricas y son útiles en el análisis preliminar del conocimiento. Después de que el conocimiento es organizado finalmente es codificado mediante alguna de las técnicas de codificación o esquemas de representación del conocimiento como los *sistemas de producción*, los cuales son explicados en la sección 2.5. Las técnicas de análisis más utilizadas son: redes semánticas, scripts, listas, árboles de decisión y tablas de decisión.

En el caso del sistema experto que se desarrolló en esta tesis, *Robot.clp,* se utilizó como técnica de análisis un *árbol de decisión* para llevar a cabo el análisis preliminar del conocimiento que debía contener el sistema experto. En la siguiente sección se explica qué es un *árbol de decisión* con el objeto de poder explicar la forma en que se llevó a cabo el análisis preliminar del conocimiento del sistema.

2.3.2 Redes Semánticas.

La *redes semánticas* son una de las formas de representar el conocimiento más viejas y más fáciles de entender. Las redes semánticas básicamente son gráficos del conocimiento que muestran las relaciones jerárquicas entre objetos.

Una red semántica se compone de *nodos* y *ligas*. Los nodos son interconectados por las ligas o arcos. Dichos arcos muestran las relaciones entre los diferentes objetos y proporcionan información descriptiva sobre ellos. Un objeto puede ser cualquier cosa tal como una casa, una habitación o incluso una persona. Los nodos también pueden ser eventos, acciones, o conceptos. Los atributos de un objeto tales como tamaño, color, posición, nombre, edad, etc., pueden también ser usados como nodos. De esta forma puede ser representada información detallada.

Los arcos que interconectan a los nodos además de mostrar la relación entre los objetos, describen ciertos factores. Algunos de los arcos más comunes son los de tipo *is-a (es-un)* o *has-a (tiene-un)*. Los arcos de tipo *is-a* es usado para mostrar la clase de relación, es decir, que un objeto pertenece a una categoría de objetos más grande. Las ligas de tipo *has-a* son usadas para identificar características o atributos de los objetos representados por los nodos.

Uno de los hechos más interesantes y más útiles de las redes semánticas es que por medio de ellas puede mostrarse jerarquía, ya que una red semántica es básicamente una jerarquía, las características de algún nodo son heredadas a otros nodo.

2.3.3 *Scripts.*

Un *Script* es un esquema de representación del conocimiento similar a un *frame*, pero en lugar de describir un objeto, el *script* describe una secuencia de eventos. Como el *frame*, el *script* retrata una situación estereotipada. A diferencia del *frame*, es usualmente presentado en un contexto particular. Para describir una secuencia de eventos, el *script* utiliza una serie de *slots*, los cuales contienen información acerca de personas, objetos y acciones involucradas en los eventos.

Algunos de los elementos de un *script* típico incluyen condiciones iniciales, accesorios, roles o papeles, pistas y escenas. Las *condiciones iniciales* describen las situaciones que deben ser satisfechas antes de que los eventos en el *script* puedan ocurrir o ser válidos. Los *accesorios* se refieren a los objetos que son usados en la secuencia de eventos. Los *roles* se refieren a las personas involucradas en el *script*. El resultado es la expresión de las condiciones que existen después que los eventos han ocurrido en el *script*. La *pista* se refiere a las variaciones que podrían ocurrir en un *script*. Y finalmente las escenas describen la secuencia de eventos actual.

Un *script* es útil en la predicción de lo que pasará en una situación específica, ya que le permite a la computadora predecir a quien le ocurrirá algo y cuando. Si una computadora pone a funcionar un *script*, es posible hacer preguntas y serán derivadas respuestas aproximadas con poca o ninguna entrada de conocimiento.

De igual forma que los *frames*, los *scripts* son particularmente útiles para representar el conocimiento, ya que existen muchas situaciones estereotipadas y eventos que la gente usa diariamente, es decir, son útiles para representar el conocimiento en situaciones que pueden ser totalmente predecibles.

Para utilizar un *script*, el conocimiento debe ser almacenado en la computadora de una forma simbólica. Para ello, debe ser utilizado un lenguaje tal como LISP o algún otro lenguaje simbólico. Una vez hecho lo anterior pueden preguntarse cosas acerca de personas y condiciones, como por ejemplo en el caso de la situación en la que se llega a un cine una pregunta que podría hacerse es: "¿Qué hace primero el cliente?" bueno, primero estaciona el auto y entonces entra en el cine. Es decir, un *Script* es útil para programar secuencias de eventos que deben ser seguidos para realizar una tarea en específico.

2.3.4 Listas de Decisión.

Las listas y los árboles son estructuras simples usadas para representar el conocimiento jerárquico.

Una *lista* es una serie escrita de arreglos relacionados. Puede ser una lista de nombres de personas que asisten a cierta escuela, una lista de compras, de cosas por hacer en un día, etc.

Las listas normalmente son utilizadas para representar el conocimiento en el cual los objetos son agrupados, clasificados o calificados de acuerdo a una relación. Los objetos primero son divididos en grupos o clases de arreglos similares. Entonces se muestran sus relaciones ligándolos. Cuando dos o más listas son combinadas se crea una jerarquía, tal como se muestra en la figura 2-3.

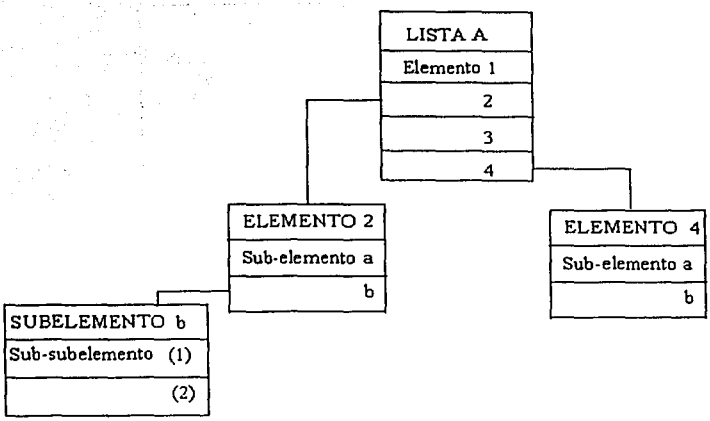

Figura 2-3. Formato general de una lista jerárquica.[12]

2.3.5 Tablas de Decisión.

Otro método para representar el conocimiento es por medio de una tabla de decisión, la cual es organizada en forma de columnas y renglones. La tabla es dividida en dos partes. Primero, es desarrollada una lista de atributos, y son listados todos los valores posibles para cada atributo. Después, se desarrolla una lista de conclusiones y finalmente se hacen corresponder las diferentes configuraciones de los atributos con las conclusiones como es mostrado en la figura 2-4.

El conocimiento almacenado en una tabla de decisión puede ser utilizado como entrada para algún otro método de representación, ya que no es posible hacer inferencias de manera directa con el contenido de las tablas.

---

[12] figura tomada de: *Ibidem. pag.*181

ATRIBUTOS			
Forma	redonda	oblicua	redonda
Color	amarillo	verde	rojo
Sabor	agrio	dulce	dulce
Textura	rugosa	suave	suave
Semillas	si	si	si
CONCLUSIONES			
Toronja	**********		
Manzana			********
Naranja china		*********	

Figura 2-4. Tabla de decisión para identificación de una fruta[13]

2.4 Árboles de Decisión.

Los árboles de decisión se componen de nodos que representan metas y ligas que representan decisiones. La raíz del árbol está a la izquierda y las hojas a la derecha. Todos los nodos terminales, excepto el nodo raíz son instancias de una meta primaria.

Al igual que las reglas, los árboles de decisión, poseen un fuerte sentido de causa y efecto. La razón por la cual se eligió utilizar un árbol de decisión en lugar de alguna otra técnica de análisis para representar el conocimiento preliminar del sistema es debido a que los árboles de decisión pueden simplificar proceso de la adquisición del conocimiento, ya que la diagramación del conocimiento es más natural que otros métodos de representación de conocimiento formales tales como las reglas. Aunque es importante señalar que una vez que el árbol de decisión es construido es posible transformarlo en reglas, las cuales son parte de un sistema de producción, como se explica en la sección 2.5. En la figura 2-5 se muestra el árbol de decisión utilizado para representar el

[13] figura tomada de: *Ibidem.* Pag.182

conocimiento preliminar del sistema experto *robot.clp*, en donde se ilustra el proceso de obtención de información sobre un objeto inanimado o un actor contenido dentro de una casa.

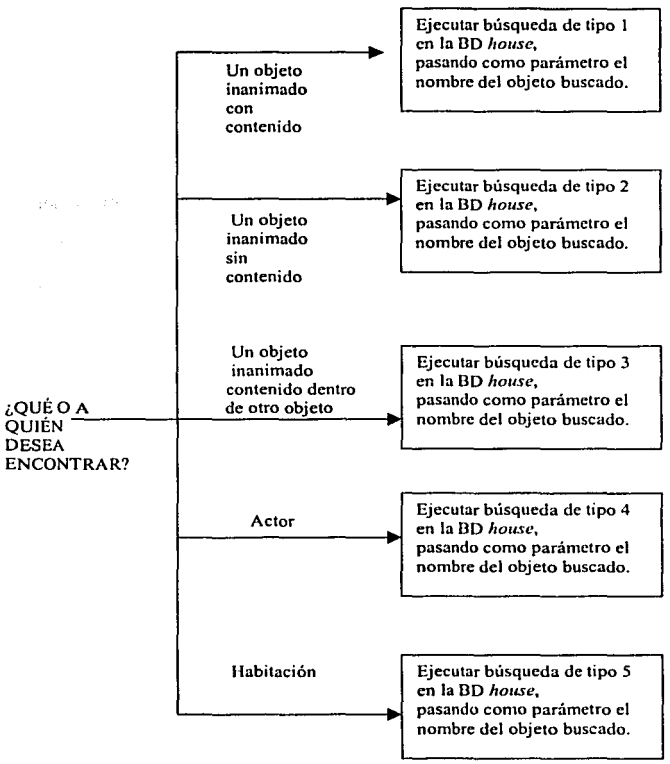

Figura 2-5. Árbol de decisión para el sistema experto *robot.clp*

Como se puede observar en la figura 2-5, el conocimiento que posee el sistema experto *robot.clp*, básicamente consiste en saber qué tipo de consulta ejecutar desde el sistema experto sobre la base de datos, ya que lo único que tiene que hacer el sistema

Robot.clp es proporcionar información concerniente al contenido de una casa. Dicha información puede ser como por ejemplo la ubicación de cierto objeto, el cual puede ser de alguno de los tres tipos mostrados en la figura 2-5 (objeto con contenido, sin contenido o contenido en otro objeto), o bien puede ser información acerca de un actor, entendiéndose por actor una persona como podría ser el papá, un robot o incluso el perro de la casa.

Es importante hacer notar que en realidad todo el conocimiento de cómo están distribuidos tanto los objetos como las personas dentro de la casa está contenido en la base de datos, la cual fue nombrada *house*. Dicha base de datos es descrita detalladamente en el capítulo 7.

Otro punto que también es importante mencionar es que para que el sistema experto pudiera llevar a cabo consultas a la base de datos fue necesario diseñar la interfaz descrita en el capítulo 8.

El conocimiento capturado a través del árbol de decisión mostrado en la figura 2-5 puede ser fácilmente convertido en reglas, ya que como fue mencionado en el párrafo final de la sección amterior, existen herramientas de software útiles para convertir un árbol de decisión en reglas, un ejemplo de dichas herramientas es CLIPS, el cual es precisamente el lenguaje que se utilizó para llevar a cabo el desarrollo del sistema experto. La descripción de dicho lenguaje es presentada en capítulo 5.

Para poder explicar cómo se transformó el conocimiento del árbol de decisión presentado en esta sección es necesario primero dar a conocer qué y cómo están constituidos los *sistemas de producción*. En la siguiente sección se muestra de manera

general la descripción general de cada uno de los elementos que constituyen un sistema de producción. Se muestra además la sintaxis general de una *regla de producción*, la cual vuelve a mencionarse en el capítulo cinco, en donde se muestra la sintaxis de una regla usada específicamente en el lenguaje CLIPS.

2.5 Sistema de Producción

El *Sistema de Producción*, es el esquema de representación del conocimiento más comúnmente usado en Sistemas Expertos. Newell y Simon[14] iniciaron el desarrollo de sistemas de producción para su modelo del conocimiento humano, ya que popularizaron el uso de reglas para representar el conocimiento humano y mostraron cómo el razonamiento podía ser representado y expresado por medio de reglas. Los psicólogos cognitivos han usado reglas para modelar el proceso de la información en el ser humano, en el cual se busca almacenar el conocimiento en memoria de largo plazo. La memoria de largo plazo consiste en muchas reglas de la estructura IF THEN. En contraste a la memoria de largo plazo, la de corto plazo se usa para almacenamiento temporal durante la resolución de un problema.

El sistema de producción es un esquema modular de representación del conocimiento. La idea básica de los sistemas de producción es que el conocimiento es representado por medio de *reglas de producción*, las cuales tienen la forma: " *SI* éstas condiciones (premisas o antecedentes) ocurren, *ENTONCES* algunas acciones (resultados, conclusiones o consecuencias) deberán ocurrir.

Además de las reglas para que un sistema de producción pueda resolver un problema necesita el procesador cognitivo, el cual trata de encontrar las reglas que serán

[14] Newell, Allen and Simon, Herbert A. *Human Problem Solving*. Prentice Hall. 1972.

activadas por un estímulo apropiado. En el caso del lenguaje de programación para Sistemas Expertos CLIPS, dicho procesador está representado por la máquina de inferencia (como se verá en el capítulo cinco), la cual debe decidir qué regla tiene la mayor prioridad, por medio del comando *salience*. Es decir, la máquina de inferencias de un sistema experto funciona como un procesador cognitivo. En el desarrollo de un sistema experto un factor importante es la cantidad de conocimiento en las reglas. Donde poca cantidad de conocimiento hace difícil de entender una regla sin referencia a otras reglas y mucha cantidad de información hace difícil la modificación del sistema experto.

2.5.1 Elementos de un Sistema de Producción.

Un sistema de producción consta de los siguientes elementos.

- Un área de *memoria* que es usada para mantener el estado actual de el universo bajo consideración.
- Un conjunto de *reglas de producción* (condición-acción).
- Un *intérprete* o *máquina de inferencia* que examine el estado actual y ejecute las reglas de producción aplicables.

A continuación se describen a cada uno de los elementos antes mencionados.

Elementos de la Memoria Global.

El área de memoria global es usada para almacenar el estado actual del sistema y se compone de una serie de elementos individuales de memoria. Conceptualmente, cada elemento de memoria describe el estado de algún aspecto en particular. Un *elemento de memoria* consta de un símbolo que identifica el elemento descrito por medio de una serie de pares de *atributos-valores*, cada uno de los cuales describe el valor actual del atributo asociado al elemento.

Reglas de Producción.

Cada regla de producción en la base de conocimiento implementa una cantidad autónoma de pericia que puede ser desarrollada y modificada independientemente de otras reglas. Cuando se combinan y alimentan a la máquina de inferencias, este conjunto de reglas se comporta en forma sinérgica, logrando mejores resultados que con la suma de resultados de las reglas individuales.

Estructura de una Regla de Producción.

Una regla de producción consta básicamente de dos partes, las cuales son descritas a continuación.

La *parte condicional* de una regla de producción; algunas veces llamada **LHS** por Left Hand Side (lado izquierdo), consiste de una serie de elementos condicionales que describen cuales serán verdaderos para que la regla pueda ser aplicable. Estas condiciones para ser identificadas son descritas como patrones requeridos de memoria global.

La *parte de acciones* de una regla de producción, algunas veces conocida como **RHS** por Right Hand Side (lado derecho), describe las acciones que serán tomadas cuando la regla se dispare. Las posibles acciones generalmente incluyen actividades como ingresar descripciones nuevas de estado en la memoria global, modificar las descripciones de estado y ejecutar una acción de usuario definida que es única para la producción específica.

Sintaxis de una Regla de Producción.

La sintaxis de una regla de producción en un lenguaje de programación de sistemas de producción típicos es:

(p <nombre de la producción>)
(<elemento condicional>)
.
.
.
(<elemento condicional>)
→
(<acción>)
.
.
.

donde <nombre de la producción> es el identificador de la producción, <elemento condicional> es un *identificador de un elemento* de memoria, un *identificador de atributo* y su *valor*.

Un *identificador de atributo* es aquel cuyo valor será comparado y *valor* es la constante o variable que identifica el valor que el atributo asociado tendrá para hacer la comparación.

La flecha → indica una implicación.

<acción> es la que será ejecutada por la producción.

Las acciones más comúnmente usadas por una regla de producción se listan de forma general a continuación:

Make	Agrega un elemento nuevo a la memoria global.
Remove	Borra un elemento de la memoria global.
Modify	Modifica el valor de un atributo específico.
Compute	Calcula el valor de las variables especificadas.
Read	Acepta un dato proporcionado por el usuario.
Write	Provee salida al usuario.
Call	Ejecuta un procedimiento específico definido por el usuario.

A pesar de no ser mencionados arriba, los elementos condicionales pueden incluir varias combinaciones AND/OR de valores de atributos y de negación de elementos condicionales.

Representación de Conocimiento Factual y por Procedimiento.

Ambos tipos de conocimiento pueden ser representados en la forma de reglas de producción. Por ejemplo, a continuación se muestra una regla de producción, la cual representa el hecho: "el pato hace quack".

```
(p el animal es un pato
      (xf (p (ANIMAL ∧ sonido quack) →
                  (modify ANIMAL ∧ nombre pato))
```

La producción anterior, llamada *el animal es un pato* indica que si el atributo *sonido* del elemento ANIMAL en la memoria global tiene el valor de "quack", entonces el atributo *nombre* del elemento ANIMAL podría ser modificado como "pato".

El siguiente conjunto de reglas de producción es requerido para ejecutar la desucción para recomendar una acción a un piloto que nota que se ha encendido una luz de alerta en el tablero de control del avión. Las reglas siguientes muestran la representación del conocimiento por procedimiento.

```
( p VERIFICANDO_LUZ
      (luz_de_alarma ∧ estado encendido)→
            (modify luz_de_alarma ∧ status (call procedimiento_verifica_luz))
( p FALSA _ALARMA
      (luz_de_alarma ∧ estado encendido ∧ status inválida)→
            (write " La luz de alarma es inválida. Continúe el vuelo normalmente"))
  ( p REVISAR_MÁQUINAS
      (luz_de_alarma ∧ estado encendido ∧ status válida)→
            (makes máquinas ∧ status (call verificando_status_de_máquinas))
( p UNA_MÁQUINA_EN_MAL_ESTADO
      (luz_de_alarma ∧ estado encendido ∧ status válida)
```

(máquinas ∧ status una en mal estado)→
(write " Ha fallado una máquina. Apagar la máquina en mal estado y continuar el vuelo"))
(p VARIAS_MÁQUINAS_EN_MAL_ESTADO
(luz_de_alarma ∧ estado encendido ∧ status válida)
(máquinas ∧ status más de una en mal estado)→
(write " Ha ocurrido un fallo múltiple en las máquinas. Efectúe un aterrizaje de emergencia inmediatamente"))

Al inicio de la ejecución, la memoria global contiene lo siguiente:

Luz_de_alarma | ∧ estado encendido

Después de la ejecución de la regla VERIFICANDO_LUZ, la memoria global contendrá alguno de los dos siguientes hechos

Luz_de_alarma | ∧ estado encendido ∧ status

ó

Luz_de_alarma | ∧ estado encendido ∧ status inválida

Si la luz es inválida, el proceso reportará el error y terminará. Pero si la luz es válida entonces el contenido de la memoria, después de haber sido ejecutada la regla REVISAR_MÁQUINAS, será alguno de los siguientes hechos.

Luz_de_alarma | ∧ estado encendido ∧ status válida
Máquinas | ∧ status una en mal estado

ó

Luz_de_alarma | ∧ estado encendido ∧ status válida
Máquinas | ∧ status más de una en mal estado

El proceso termina después de ejecutar la acción correspondiente a alguna de las reglas UNA_MÁQUINA_MAL_ESTADO o VARIAS_MÁQUINAS_MAL_ESTADO, dependiendo de cual de los hechos mostrados anteriormente se encuentre en la memoria

global. Es importante hacer notar, que el intérprete no es sensitivo al orden en el cual las reglas pudieran aparecer.

Una de las ventajas de un sistema de producción es que almacena conocimiento en una forma modular y uniforme. Cada producción es esencialmente una entidad separada e independiente y las producciones nunca llaman a otra. Esto facilita agregar, eliminar o modificar producciones.

Intérprete.

En su forma más esencial, el intérprete reconoce y ejecuta una producción cuyo lado izquierdo ha sido satisfecho. Para reconocer reglas aplicables, el intérprete compara los patrones de *atributo-valor* con el estado actual de la memoria. El proceso de razonamiento continúa debido a que la ejecución de una producción normalmente cambia el contenido de memoria global y por lo tanto activa producciones adicionales.

Proceso de correspondencia.

Durante el proceso de correspondencia, el intérprete primero otorga instancias a cada regla. Entonces compara los valores de los elementos condicionales de cada regla con los correspondientes elementos de memoria para identificar las reglas cuyas condiciones han sido satisfechas.

Resolución de Conflicto.

En un sistema de producción grande, el ciclo de correspondencia frecuentemente identificará a más de una posible correspondencia. El intérprete agrega a todas las instancias correspondientes a un grupo *llamado conjunto conflicto* y los considera como candidatos para ejecución.

La *Resolución del Conflicto* es el proceso mediante el cual se selecciona a la instancia dominante, es decir, a una instancia especifica que será ejecutada. Esta selección se basa en una *estrategia de resolución de conflicto.*

2.6 *Frames.*

Un *frame* es una estructura de datos que incluye todo el conocimiento sobre un objeto particular. Este conocimiento es organizado en una estructura jerárquica especial. Los *frames* son básicamente una aplicación de la programación orientada a objetos para SE.

Los *frames* proveen una concisa representación del conocimiento en una forma natural. En contraste con otros métodos de representación, los valores que describen un objeto son agrupados juntos en una sola unidad llamada *frame*. Por lo tanto, un *frame* abarca, objetos complejos, situaciones enteras o un problema de administración como si fuera una sola entidad. En un *frame* el conocimiento es dividido en *slots*. Un slot puede describir conocimiento declarativo (tal como el color de una casa) o conocimiento procedural (tal como "activar cierta regla si un valor excede a un nivel dado)".

Un *frame* es un bloque relativamente grande o una cantidad de conocimiento sobre un objeto, situación, evento, localización o cualquier otro elemento particular. El *frame* describe a un objeto en gran detalle. El detalle es dado en forma de *slot*, los cuales describen los diferentes atributos y características del objeto o situación.

Los *frames* son normalmente utilizados para representar conocimiento estereotipado o conocimiento construido a partir de características y experiencias muy bien conocidas. Los *frames* son útiles para representar las experiencias y el sentido común que se encuentra almacenado en el cerebro de alguna persona, la cual al enfrentarse con un problema recurre a dicho conocimiento para poder resolverlo.

La característica básica de un *frame* es que representan conocimiento relacionado acerca de un objeto que tiene mucho conocimiento por *"default"* o instanciación. Un sistema de *frame* puede ser una buena opción para describir un dispositivo mecánico como un carro. Los componentes del carro como la máquina, cuerpo, frenos y demás elementos que lo componen pueden ser relacionados para dar una vista completa de sus relaciones.

Un *frame* contiene dos elementos básicos: *slots* y *facets*. Un slot es un conjunto de atributos que describen al objeto representado por el frame. Los *facets* o *subslots* describen conocimiento o procedimientos acerca del atributo del slots. Un *facet* puede tomar alguna de las siguientes formas:

- *Valores*. Describen a los atributos tales como azul, rojo o amarillo para el caso de un slot color.
- *Default*. Este facet es usado en caso de que el slot esté vacio.
- *Rango*. Indica el tipo de información que puede aparecer en un slot.
- *If agregado*. Esta faceta contiene información procedural, la cual especifica una acción a ser tomada cuando es agregado un valor en un slot. Dichos procedimientos son llamados *demos*.
- *If necesario*. Este *slot* es usado en caso de que ningún valor sea dado al slot.

En la tabla 2-1 se presentan de manera resumida las mayores capacidades o habilidades de los frames.

Habilidad para relacionar los valores permitidos que cierto atributo puede tomar.
Modularidad de la información, permitiendo la fácil expansión y mantenimiento del sistema
Proveen una sintaxis más fácil de leer y de poder ser transformada en reglas
Plataforma para la construcción de interfaces gráficas
Acceso a un mecanismo que soporta la herencia de información a las diferentes clases

Tabla 2-1. Propiedades de un Frame.[15]

[15] Tabla tomada de: R. A. Edmonds, *The Prentice-Hall Guide to Expert Systems*. Prentice-Hall 1988.

2.7 Métodos Formales para la Representación del Conocimiento.

La forma más antigua de representar el conocimiento es mediante la lógica. La lógica es considerada una subdivisión de la filosofía, su desarrollo se atribuye a los antiguos griegos. La lógica es de gran importancia para los SE, ya que la *maquina de inferencia* parte de *hechos* para llegar a *conclusiones.*

Un proceso lógico generalmente consta de *premisas* , las cuales son declaraciones u observaciones. Las premisas son ingresadas al sistema lógico y utilizadas para generar una salida. Dicha salida consiste en las conclusiones o *inferencias.* Con este proceso los hechos que se conocen como verdaderos pueden ser utilizados para derivar nuevos hechos que igualmente serán verdaderos.

Dos formas básicas de la lógica son: la *lógica propositiva* y la *lógica predicativa.*

2.7.1 Lógica Propositiva.

La lógica propositiva, algunas veces llamada cálculo propositivo, es una lógica simbólica para manipular proposiciones. En particular, la lógica propositiva trata con la manipulación de variables lógicas las cuales representan proposiciones.

Una proposición no es más que una afirmación, la cual puede ser falsa o verdadera. Una proposición puede ser una premisa que puede ser usada para derivar nuevas proposiciones o inferencias. Dentro de un SE que usa reglas, éstas son utilizadas para determinar la veracidad o falsedad de una nueva proposición, la cual fué generada a partir de una proposición de tipo premisa. En la lógica propositiva se utilizan símbolos tales como letras del alfabeto para representar proposiciones, premisas o conclusiones.

Debido a que los problemas del mundo real involucran una gran cantidad de proposiciones interrelacionadas entre sí, se necesita formular premisas complejas, para lo cual pueden ser combinadas dos o más proposiciones mediante el uso de las conectivas lógicas. Dichas conectivas son designadas como AND, OR, NOT, implicación y equivalencia. El significado de cada uno de estas conectivas y los símbolos utilizados para representarlas se muestran en la tabla 2-2. Las conectivas son usadas para unir o modificar las proposiciones existentes para derivar nuevas. En la tabla 2-3 se muestran los valores para los conectivos binarios lógicos AND, OR, implicación y la bicondicional

CONECTIVO	SIGNIFICADO
AND o conjunción	$\wedge$
OR, disyunción	$\vee$
NOT, negación	$\sim$
Implicación (if...then)	$\rightarrow$
If and only if; bicondicional	$\leftrightarrow$
Equivalente	$\equiv$

Tabla 2-2. Símbolos de representación. [16]

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

Tabla 2-3.
Valores para los conectivos binarios lógicos AND, OR, implicación y la bicondicional.[17]

2.7.2 Lógica Predicativa.

La Inteligencia Artificial(IA) utiliza *lógica predicativa* en lugar de lógica propositiva para representar el conocimiento, ya que la habilidad de la lógica propositiva para representar conocimiento de problemas reales es muy limitada. Esto debido a que únicamente trata con declaraciones completas y se encarga de decidir si éstas son

[16] Tabla tomada de Turban, Efraim, *Expert Systems and Applied Artificial Intelligence*. Edit. McMillan. 1992. pag171.
[17] Tabla tomada de Giarratano, Joseph. ed. al. *Expert Systems: principles and programming*. Pag. 95

verdaderas o falsas, lo cual resulta no ser de mucha utilidad para la representación del conocimiento de problemas reales. Para subsanar esto se creo el predicado lógico, el cual concierne con el uso de palabras especiales llamadas cuantificadores, como son "todo","alguno" y "no".

La lógica predicativa es una forma de lógica más sofisticada que usa los mismos conceptos y reglas de la lógica propositiva, pero con la diferencia de que aumenta la habilidad para representar el conocimiento. La lógica predicativa permite descomponer una declaración o proposición en las partes que la componen, es decir, en objetos, sus características o algunas afirmaciones acerca de los objetos. El cálculo predicativo o lógica predicativa, permite descomponer una proposición en los objetos sobre los cuales algo está siendo afirmado y en la afirmación en sí misma. Además permite el uso de variables y funciones de variables en una declaración de lógica simbólica, lo cual da como resultado un esquema de representación del conocimiento más poderoso y mucho más aplicable en problemas prácticos que son resueltos por medio de la computación.

En cálculo predicativo, una proposición es dividida en dos partes: los *argumento* (u objetos) y el *predicado* o afirmaciones. Los *argumentos* son los objetos sobre los cuales una afirmación es hecha. El predicado es la afirmación hecha sobre un objeto. Una vez que el conocimiento está organizado ya sea en lógica propositiva o predicativa, entonces ya está listo para hacer inferencias con él. La lógica predicativa consta de cuatro componentes: un *alfabeto*, un *lenguaje formal*, un conjunto de declaraciones básicas llamadas *axiomas* (expresadas en el lenguaje formal) y de un conjunto de *reglas de inferencias.* Cada axioma describe un fragmento de conocimiento y las reglas de inferencia son aplicadas a los axiomas para derivar nuevas declaraciones de verdad.

El *alfabeto* en lógica predicativa consiste de símbolos de los cuales son estructuradas las declaraciones en el lenguaje formal. Un alfabeto consta de *predicado, variables, funciones, constantes, conectivas, cuantificadores* y *delimitadores* tales como paréntesis y comas. Todos los elementos mencionados anteriormente son combinados para construir fórmulas.

2.8 Ventajas y Desventajas de los Diferentes Tipos de Representación del Conocimiento.

La tabla 2-4 muestra las ventajas y desventajas de algunos de los métodos para representar el conocimiento que fueron presentados en éste capítulo.

ESQUEMA	VENTAJAS	DESVENTAJAS
Reglas de Producción	Sintaxis simple, fácil de entender, simple interpretación, modularidad alta y flexible (fácil de modificar)	Difícil seguimiento de jerarquías, ineficiente para sistemas muy grandes, no siempre el conocimiento puede ser representado por medio de reglas y es pobre en la representación del conocimiento estructural y descriptivo.
Redes Semánticas	Fácil seguimiento de jerarquías, facilidad de asociación, flexible.	El significado asociado a los nodos podría ser ambiguo y es difícil de programar.
Frames	Poder de expresión, facilidad de actualización de nuevas propiedades y relaciones contenidas en los slots, facilidad de creación de procedimientos especializados, facilidad para incluir información por default y detectar valores erróneos.	Alto grado para programar y para hacer inferencias con el conocimiento representado con los frames y el software para hacer la programación de los frames es de alto costo.
Lógica Formal	Los hechos son declarados independientemente de su uso, seguridad de que únicamente las conclusiones válidas son afirmadas o declaradas (precisión). Completez.	Separación de la representación y el procesamiento con grandes cantidades de datos ineficiente y muy lento cuando la base de conocimientos es muy grande.

Tabla 2-4. Ventajas y desventajas de algunos métodos para representar el conocimiento.[18]

[18] Tabla tomada de: Turban, Efraim, *Expert Systems and Applied Artificial Intelligence*. Edit. McMillan. 1992 pag. 194.

CAPÍTULO 3.

MÉTODOS DE INFERENCIA.

3.1 Introducción

La importancia de la inferencia para los Sistemas Expertos se debe al hecho de que el deducción es la técnica común por medio de la cual dichos sistemas resuelven los problemas. Además, cuando no es posible solucionar un problema por medio de un algoritmo, el razonamiento ofrece la única posibilidad de solución y es entonces cuando los sistema experto son utilizados.

La máquina de inferencias es el corazón de un sistema experto basado en reglas, ya que es el software que provee el mecanismo que controla algunos procesos de razonamiento y la ejecución de las reglas contenidas en la base del conocimiento. La máquina del conocimiento también es conocida como intérprete de reglas para el caso de sistemas basados en reglas.

El programa de control, o máquina de inferencias dirige la búsqueda a través de la base del conocimiento. El proceso de búsqueda o *proceso de correspondencia de patrones* involucra la aplicación de reglas de inferencia. El programa de control decide qué regla eliminar y qué atributo hace corresponder el o los patrones condicionales de una regla.

Los métodos de inferencia más usados por los Sistemas Expertos son: *Encadenamiento hacia Adelante* y *Encadenamiento hacia Atrás*, las cuales, además de otras técnicas, serán descritas en el presente capítulo.

3.2 Categorías de Razonamiento

- **Razonamiento Deductivo.**

Es un proceso en el cual son usadas *premisas generales* para obtener una inferencia específica. El razonamiento parte de un principio general para llegar a una conclusión.

- **Razonamiento Inductivo.**

El razonamiento inductivo utiliza un número de hechos o premisas establecidas para llegar a alguna conclusión general. Lo interesante de este tipo de razonamiento es que podría ser muy difícil llegar a la conclusión. Las conclusiones pueden cambiar si son descubiertos nuevos hechos. Siempre estará presente la incertidumbre en las conclusiones a menos que todos los hechos posibles sean incluidos en las premisas, lo cual es imposible. Entre más conocimiento se tenga, las inferencias pueden ser más conclusivas.

Tanto el razonamiento deductivo como el inductivo son utilizados por los sistemas basados en reglas.

- **Razonamiento Análogo.**

Este tipo de razonamiento es natural para los seres humanos, pero es difícil de realizar mecánicamente. El razonamiento análogo consiste en relacionar el pasado con el presente para lograr relacionar los objetos o conceptos que aún no han sido relacionados, es decir, el razonamiento análogo es un proceso que requiere la habilidad de reconocer experiencias que han sido vividas previamente por el individuo que esta haciendo el razonamiento. El uso de este tipo de razonamiento aún no ha sido explotado en el campo de la IA.

- **Razonamiento Formal.**

El razonamiento formal involucra la manipulación sintáctica de estructuras de datos para deducir nuevos hechos mediante reglas de inferencia prescritas. Un ejemplo de este tipo de razonamiento es la lógica matemática usada para probar teoremas geométricos.

3.3 Razonamiento Mediante el Uso de la Lógica.

Un sistema lógico puede ser utilizado para expresar hechos y conocimiento de manera simbólica, pero lo que en realidad se pretende que los Sistemas Expertos puedan hacer inferencias con el conocimiento representado mediante la lógica. Para manipular las expresiones lógicas y crear nuevas, son usadas varias reglas de inferencias.

Para ejecutar razonamiento deductivo e inductivo existen varias reglas de inferencia básicas de razonamiento, las cuales permiten la manipulación de expresiones lógicas para crear nuevas. Las reglas de inferencia más importantes son las llamadas *modus ponens y modus tollens*, las cuales serán explicadas en la siguiente sección.

3.3.1 Reglas de Inferencia.

La lógica propositiva u ontológica, en donde existen cosas que pueden ser verdaderas o falsas, ofrece significados para la descripción de argumentos. De hecho, con frecuencia son utilizadas las proposiciones lógicas, tal es el caso del siguiente argumento propositivo.

Si hay corriente eléctrica, el radio encenderá
Hay corriente eléctrica
∴ El radio encenderá

El argumento anterior puede ser expresado formalmente usando letras para representar las proposiciones.

A = Hay corriente eléctrica
B = El radio encenderá

Así que el argumento puede ser escrito de la siguiente forma.

$A \rightarrow B$
A
$\therefore B$

Argumentos como el anterior son utilizados frecuentemente. Un esquema general para representar argumentos de este tipo es el siguiente.

$p \rightarrow q$
p
$\therefore q$

Donde p y q son variables lógicas, las cuales representan cualquier declaración. El uso de variables lógicas en lógica propositiva permite la manipulación de declaraciones. El esquema de inferencia de esta forma propositiva es llamada por una variedad de nombres: *razonamiento directo, regla de asumir el antecedente y* ***modus ponens***, el cual viene del Latín *modus* que quiere decir forma o manera y *ponens* que significa afirmación.

El *modus ponens* es de gran importancia, ya que este tipo de razonamiento forma las bases o fundamentos teóricos de los sistemas expertos basados en reglas, como es el caso del sistema experto *robolt.clp*. La proposición compuesta, $p \rightarrow q$, corresponde a la regla mientras p corresponde al patrón que debe ser satisfecha por el antecedente para que la regla pueda ser aplicada. Sin embargo el condicional $p \rightarrow q$ no es exactamente equivalente a una regla, porque el condicional es una definición lógica establecida por una tabla de verdad y hay muchas definiciones posibles del condicional.

Otra notación para representar el esquema del *modus ponens* es la siguiente.

$p, p \rightarrow q; \therefore q$

donde la coma es usada para separar una premisa de la otra y el punto y coma indica el final de las premisas. La forma general de representar argumentos con más de una premisa es la siguiente.

$P_1, P_2, \ldots P_N; \therefore C$

donde P_i representa premisas tales como r, $r \rightarrow s$ y C es la conclusión.

Un argumento para las reglas de producción puede ser escrito en forma general como se muestra a continuación.

$$C_1 \wedge C_2 \wedge \ldots C_N \rightarrow A$$

lo cual quiere decir que si cada condición, C_i, de una regla es satisfecha, entonces la acción A de la regla es ejecutada. Una declaración lógica de la forma anterior no es estrictamente equivalente a una regla porque la definición lógica del condicional no es igual a la definición de regla de producción. Sin embargo está forma lógica es útil para ayudar a pensar en las reglas.

El argumento del *modus ponens*,

$p \rightarrow q$
p
$\therefore q$

es válido porque puede ser expresado como una tautología:

$$((p \rightarrow q) \wedge p) \rightarrow q.$$

La tabla de verdad para la tautología anterior está representada en la tabla 3-1. La anterior es una tautología porque todos los valores del argumento mostrados en quinta la columna son verdaderos sin importar cual sea el valor de sus premisas. En la tercera, cuarta y quinta columna, los valores verdaderos están escritos bajo ciertos operadores lógicos tales como el $\rightarrow$ (entonces) y el operador $\wedge$ (intersección). A estos operadores lógicos se les llama *conectores* y fueron presentados en la tabla 2-2.

p	q	$p \rightarrow q$	$(p \rightarrow q) \wedge p$	$(p \rightarrow q) \wedge p \rightarrow$
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

Tabla 3-1

Tabla de verdad para la tautología del ejemplo anterior (modus ponens).

Aunque este método para determinar la validez de un argumento funciona, es un método muy largo y tardado, ya que se requiere verificar cada renglón de la tabla de verdad. El número de renglones es 2^N donde N es el número de premisas, de tal forma que entre más grande sea el número de premisas más grande será el número de renglones que necesitarán ser verificados.

Un método más corto para determinar la validez de un argumento es considerar sólo aquellos renglones de la tabla de verdad en los cuales las premisas sean todas verdaderas. La definición equivalente de un argumento válido establece que será válido si y sólo si la conclusión es verdadera para cada uno de esos renglones, es decir, la conclusión es tautologicamente implicada por las premisas. En la tabla 3-1 se puede observar que en el primer renglón tanto la premisa $p \rightarrow q$ como la premisa p son ambas verdaderas, así que de ahí se puede concluir que el *modus ponens* representado por la tabla de verdad es un argumento válido. Si existiera algún otro renglón en donde las premisas fueran todas verdaderas pero la conclusión fuera falsa, entonces el argumento sería invalido. Un argumento inválido es llamado *falacia* o argumento *opuesto.*

El esquema de argumento,

$p \rightarrow q$
$\sim q$
$\therefore \sim p$

es un argumento válido, como se puede observar en la tabla 3-2. Este esquema particular es llamado por una variedad de nombres: razonamiento indirecto, ley de contraposición y ***modus tollens***, el cual viene de Latín *modus* que quiere decir manera y *tollere* que quiere decir denegar.

p	q	PREMISAS		CONCLUSIÓN
		$p \rightarrow q$	$\sim q$	$\sim p$
T	T	T	F	F
T	F	F	T	F
F	T	T	F	T
F	F	T	T	T

Tabla 3-2. Tabla de verdad para el argumento *modus tollens* del ejemplo anterior.

Tanto el *modus ponens* como el *modus tollens* son **reglas de inferencia**, algunas veces llamadas **leyes de inferencia**.

3.4 Resolución.

La regla de inferencia llamada *resolución* fue introducida en 1965, es comúnmente implementada para la comprobación de teoremas en programas de Inteligencia Artificial. A diferencia de muchas diferentes reglas de inferencia de aplicación limitada tales como *modus ponens, modus tollens, encadenamiento,* etc la regla de inferencia de resolución es de propósito general. La aplicación de la resolución hace que la comprobación de teoremas sea automática.

El procedimiento es un método automático general para determinar si un teorema es derivado de un conjunto dado de premisas. El teorema a ser probado es negado y colocado, junto con el conjunto de premisas, en una cláusula.

El método de resolución es implementado en el lenguaje de programación PROLOG, el cual permite la resolución de problemas representándolos por medio del lógica predicativa. La inferencia es hecha automáticamente por el algoritmo de resolución que es parte de PROLOG.

La importancia del proceso de resolución, el cual es usado también en sistemas de reglas de producción (como el sistema experto *robot.clp*), es que permite la derivación de

nuevos hechos a partir de reglas y hechos conocidos. En realidad este proceso es muy similar al sentido común utilizado por los humanos para resolver problemas. La simple deducción es muy natural para el cerebro humano, pero no es así para una computadora. Por ello, es necesario indicarle a la computadora la manera en que debe hacer inferencias simples. Una manera de hacer esto es por medio del método de inferencia *modus ponens*.

3.5 Inferencia por Medio de Reglas.

Hacer inferencia con reglas implica la implementación de el métodos de inferencia tales como el *modus ponens*, el cual fue descrito en la sección 3.3.1. El mecanismo de inferencia en la mayoría de Sistemas Expertos comerciales utilizan el método *modus ponens*, el cual se ve reflejado en el mecanismo de búsqueda usado por el intérprete de reglas. Para explicar lo anterior considérese el siguiente ejemplo.

REGLA 1: SI se inicia un conflicto internacional
ENTONCES el precio del oro aumentará.

Asumiendo que el sistema experto sabe que un conflicto internacional ha empezado. Esta información es almacenada en la *lista de hechos* de la base de conocimientos. Lo anterior quiere decir que el patrón condicional de la Regla 1 se satisface. Usando *modus ponens*, la conclusión es entonces aceptado como verdadero. Se dice que la Regla 1 es "disparada". Una regla se dispara sólo cuando todos los patrones condicionales de la regla son satisfechos, entonces la conclusión de dicha regla es almacenada en la *lista de hechos*. En el caso del ejemplo de la Regla 1, al ser disparada es ingresado a la lista de hechos la conclusión: (el precio del oro aumentará), la cual podría ser utilizada para satisfacer la premisa de otra regla. La comprobación de una premisa o una conclusión de una regla puede ser tan simple como la correspondencia

simbólica de patrones de una regla con un patrón similar localizado en la lista de hechos. Este proceso es llamado *correspondencia de patrones.*

El proceso por medio del cual cada regla es revisada para ver si sus premisas pueden ser satisfechas por hechos previamente introducidos a la lista de hechos puede hacerse en dirección hacia adelante o hacia atrás. Dicho proceso continúa hasta que ninguna regla pueda ser disparada.

3.6 Encadenamiento Hacia Atrás y Hacia Adelante.

Existen dos métodos para controlar la inferencia en SE basados en reglas: encadenamiento hacia adelante y encadenamiento hacia atrás.

Un encadenamiento es un grupo de inferencias múltiples que conecta un problema con su solución. El encadenamiento hacia delante es aquel en el que el encadenamiento puede guiarnos desde el problema hasta la solución del mismo. Otra forma de describir el encadenamiento hacia adelante es razonando a partir de las afirmaciones hasta llegar a la conclusión derivada de dichas afirmaciones.

El encadenamiento hacia atrás es el encadenamiento que es seguido desde una hipótesis hasta llegar a las afirmaciones que soportan dicha hipótesis. El encadenamiento hacia atrás puede ser descrito en términos de una meta, la cual puede ser alcanzada por medio de la satisfacción de sub-metas o metas intermedias.

El encadenamiento puede ser expresado en términos de inferencia. Por ejemplo, suponiendo que se tiene una regla de tipo *modus ponens* como la siguiente.

$p \rightarrow q$
p
$\therefore q$

la regla anterior forma una cadena de inferencia, y puede ser expresada de la siguiente manera.

Elefante (x) → mamífero (x)
Elefante (Clyde)
∴ mamífero (Clyde)

La regla anterior puede ser usada en un encadenamiento de inferencia causal hacia delante, el cual deduce que Clyde es un mamífero dado que es un Elefante. La cadena de inferencias para la regla anterior es ilustrada en la figura 3-4, en donde, el encadenamiento está representado por la secuencia de las líneas verticales, las cuales conectan el consecuente de una regla con el antecedente de la siguiente regla. El encadenamiento causal es en realidad una secuencia de implicaciones y unificaciones (sustituciones).

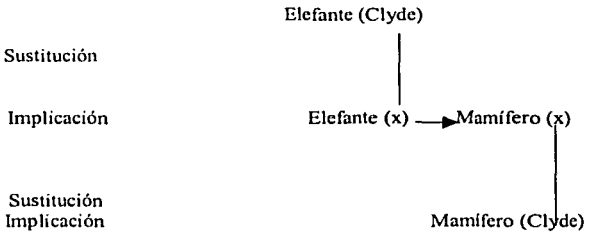

Figura 3-1
Cadena de inferencia usando encadenamiento hacia delante para la regla del ejemplo anterior. [19]

El encadenamiento hacia atrás es el proceso inverso. Retomando el ejemplo de la figura 3-1 se puede explicar el encadenamiento hacia atrás partiendo de la hipótesis de que Clyde es un mamífero. El problema principal del encadenamiento hacia atrás es encontrar una cadena que ligue una evidencia a la hipótesis. En este caso el hecho Elefante (Clyde) es llamado la *evidencia*, lo cual indica que es el hecho que soporta la hipótesis, es como la forma en que la evidencia es usada en una corte para probar la culpabilidad de el acusado.

Es útil visualizar el encadenamiento hacia atrás y hacia delante en términos de una ruta a través de un espacio de problema, en el cual los estados intermedios

[19] Giarratano, Joseph. ed. al. *Expert Systems: principles and programming* Pag 160.

corresponden a hipótesis intermedias bajo el criterio de encadenamiento hacia atrás o pueden ser conclusiones intermedias si se está llevando a cabo un encadenamiento hacia delante. La tabla 3-3 resume algunas de las características comunes del encadenamiento hacia atrás y hacia delante.

ENCADENAMIENTO HACIA ADELANTE	ENCADENAMIENTO HACIA ATRÁS
• Planeación, vigilancia, control	• Diagnosis
• Parte del presente al futuro y de antecedente a consecuente.	• Parte del presente al pasado y de consecuente con antecedente.
• Busca las soluciones que derivan de los Hechos	• Busca los hechos que soportan la hipótesis
• Se facilita la búsqueda de primero en Amplitud	• Se facilita la búsqueda de primero en Profundidad
• La búsqueda es determinada por los Antecedentes.	• La búsqueda es determinada por los consecuentes.

Tabla 3-3.
Características más importantes de los métodos de inferencia: encadenamiento hacia delante y hacia atrás.

El encadenamiento hacia delante es llamado también *razonamiento abajo-arriba,* debido a que razona a partir del nivel más bajo de evidencia, los hechos, hasta el un nivel alto, las conclusiones que son derivadas de los hechos. Los hechos son las unidades elementales, ya que no pueden ser descompuestos en unidades más pequeñas que tengan algún significado o sentido.

Así que el razonamiento que parte de una construcción de alto nivel, como una hipótesis, y llega al nivel más bajo que son los hechos que soportan la hipótesis es llamado *razonamiento arriba-abajo* o encadenamiento hacia atrás. En este tipo de razonamiento el sistema solicita información al usuario, la cual le servirá como evidencia para aprobar o desaprobar una hipótesis. Esto contrasta con los sistemas de

encadenamiento hacia delante, en los cuales todos los hechos relevantes son conocidos de ante mano.

Un aspecto importante al solicitar evidencias al usuario es hacer la pregunta correcta. Un requerimiento obvio para esto es que el sistema experto debería únicamente hacer las preguntas que guiarán a la hipótesis que se está tratando de probar. Mientras que es posible la existencia de cientos o miles de preguntas que el sistema pudiera formular, hay un costo en tiempo y en dinero para obtener la evidencia que responda una pregunta.

Idealmente un sistema experto debería permitirle al usuario dar evidencias voluntarias, aún cuando el sistema no las haya pedido, esto aumentaría la velocidad del proceso de encadenamiento hacia atrás y haría que el sistema fuera más conveniente para el usuario. Este tipo de evidencias voluntarias dejaría que el sistema se salte algunas ligas en el encadenamiento causal. La desventaja es que la programación del sistema experto se vuelve más compleja, puesto que el sistema no seguiría una cadena liga por liga.

Los métodos de inferencia mostrados en el presente capítulo no son los únicos que existen, sin embargo, son los más utilizados para el desarrollo de sistemas expertos.

CAPÍTULO 4

LENGUAJES DE PROGRAMACIÓN PARA SISTEMAS EXPERTOS.

4.1 Introducción.

Una vez que se tiene el conocimiento necesario para la construcción del sistema experto *Robot.clp*. Es necesario codificarlo con la ayuda de un *Lenguaje de Programación* de tal forma que el conocimiento pueda ser almacenado y procesado en una computadora. Es por ello, que en el presente capítulo se explicará porqué se eligió CLIPS para programar el sistema experto *Robot.clp*. También se hablará de LISP y PROLOG debido a que son los dos son los lenguajes de programación de Inteligencia Artificial (o de manipulación simbólica) más importantes. Se mencionarán las principales características de dichos lenguajes de programación, esto con la finalidad de poder identificar las diferencias entre ellos y poder justificar el uso de CLIPS para implementar el sistema experto Robot.clp.

4.2 Elección del Mejor Paradigma.

Una decisión fundamental al solucionar un problema es la elección de la mejor forma de modelarlo. Esto es escoger el mejor *paradigma* por medio del cual será solucionado dicho problema. El término *paradigma* viene de la palabra griega *paradeigma*, la cual quiere decir modelo, patrón o ejemplo. En computación, un *paradigma* es una metodología consistente y organizada para resolver un problema.[20]

A continuación se mencionan las principales diferencias entre dichos lenguajes con la finalidad de justificar el porqué se utilizó un lenguaje para Sistemas Expertos.

[20] Giarratano, Joseph C. *CLIPS User's Guide*, pag. 77

La primera diferencia fundamental entre lenguajes de sistemas expertos y lenguajes de programación estructurada es el centro de su representación. Los lenguajes estructurados se centran en proveer técnicas flexibles y robustas para representar datos, por ejemplo arreglos, cadenas, pilas y árboles; En cambio el paradigma del sistema experto provee una manera para representar el conocimiento. Por ejemplo provee la propiedad de separación de los hechos (abstracción de datos) y las reglas(abstracción del conocimiento) en un lenguaje que se basa en reglas.

Una manera de saber si es necesario seleccionar un lenguaje para sistemas expertos en lugar de un lenguaje de programación convencional es analizar si lo que se quiere es programar la pericia de un experto humano, la cual puede representarse en reglas de tipo SI-ENTONCES.

4.3 Clasificación de los Lenguajes de Programación.

Los paradigmas de programación pueden ser clasificados como procedurales y no-procedurales. En la figura 4-1 se muestra una clasificación de los paradigmas procedurales en términos de lenguajes de programación. En la figura 4-2 se muestra una clasificación para los paradigmas no-procedurales. Existen algunos lenguajes que poseen ciertas características que podrían colocarlos en más de una clasificación.

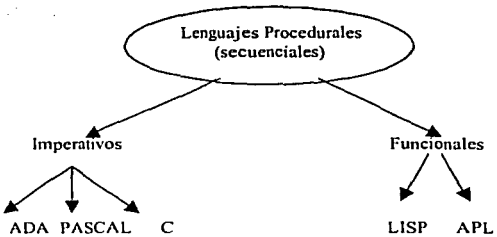

Figura 4-1. Lenguajes Procedurales[21]

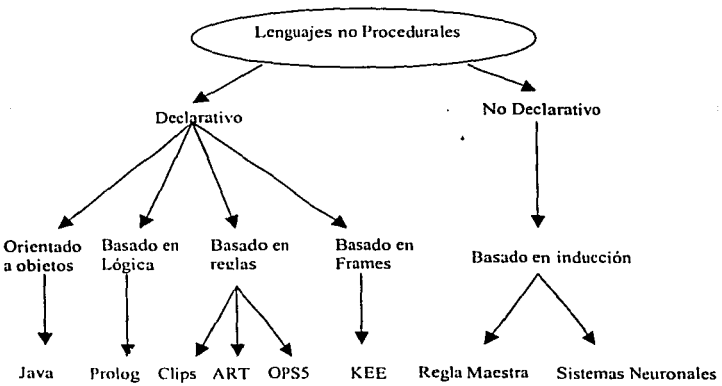

Figura 4-2. Lenguajes no Procedurales.[22]

FORTRAN 77, INSIGHT 2 en Turbo Pascal, y EXSYS, son algunos ejemplos de lenguajes que no son de Inteligencia Artificial, sin embargo dichos lenguajes pueden ser usados para desarrollar Sistema Expertos. En ocasiones se opta por utilizar alguno de los lenguajes de programación mencionados para codificar un sistema experto debido a que no se cuenta con el hardware que los lenguajes de Inteligencia Artificial requieren. La implementación de un sistema experto utilizando alguno de los lenguajes que no son de

[21] figura tomada de: Giarratano, Joseph. ed. al. *Expert Systems: principles and programming*. Pag. 36
[22] Tabla tomada de: *Ibidem*. Pag. 36.

IA requiere más memoria y no da a los programadores ese fino control que necesitan para conservar la memoria, por lo que un lenguaje que no es de IA puede ser muy restrictivo.

Los lenguajes de IA o de manipulación simbólica, proveen un efectivo camino para presentar los objetos tipo IA. Los dos lenguajes más importantes son LISP y PROLOG. Con estos lenguajes la programación y la supresión de errores comúnmente pueden hacerse mas rápidamente. En las siguientes secciones se hablará más a detalle de ellos.

4. 4 LISP

El nombre LISP es la abreviatura de **List-Processing**, ya que el LISP fue desarrollado para el procesamiento de *listas*. La lista es la estructura más importante de LISP. Una lista es una secuencia de números, cadenas de caracteres o incluso de otras listas.

LISP es uno de los más viejos lenguajes de propósito general, fue desarrollado en 1958 por McCarthy y aún es utilizado. Las aplicaciones de LISP van desde un SE, a un procesador de lenguaje natural, robótica y hasta programación educacional y psicológica. Su característica única brinda al programador el poder de desarrollar software que va más allá de las limitaciones de otros lenguajes convencionales como COBOL y Pascal.

Específicamente, LISP está orientado hacia la computación simbólica. Un programa de LISP puede manipular convenientemente tales símbolos y las relaciones entre ellos. Los programas escritos en LISP tienen la habilidad de modificarse a sí mismos. En un sentido limitado, esto significa que la computadora es capaz de programarse a sí misma para aprender a partir de su propia experiencia.

LISP permite a los programadores representar objetos tales como redes semánticas en forma de listas. Además permite hacer ciertas operaciones sobre las listas, tales como dividir una lista en varias partes y hacer nuevas listas mediante la unión de listas existentes. Convencionalmente, un programador de LISP escribe una lista como una secuencia de elementos encerrados entre paréntesis cuadrados. Frecuentemente, una lista es representada mediante un diagrama en donde los elementos están encerrados en cuadros y son unidos por flechas. En la figura 4-3 se muestra un diagrama de una lista que representa la sentencia "el cielo es azul".

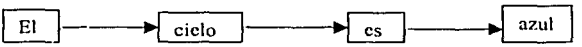

Figura 4-3
Diagrama de una lista que representa una sentencia.

En la mayoría de las situaciones de programación, los elementos de una lista son otras listas o sublistas.

Usualmente, el código de LISP es ejecutado directamente por un intérprete de LISP. En algunas versiones el código fuente del programa es compilado para aumentar la eficiencia.

LISP es un lenguaje de *programación funcional*, cuya idea fundamental es combinar funciones simples para construir funciones más poderosas. Esto es esencialmente un diseño de tipo *bottom-up* (de abajo hacia arriba), en contraste con el diseño común de tipo *top-down* (de arriba hacia abajo) de los lenguajes imperativos tales como C y ADA.

En LISP_son provistas algunas funciones primitivas. Dichas funciones primitivas son: QUOTE, CAR, CDR, CPR, CTR, CONS, EVAL, COND, LAMBDA, DEFINE, y LABEL.

Existen numerosas versiones de LISP. Algunas de ellas incluyen características para aplicaciones especiales. Las versiones más notables son: COMMON LISP, IQLISP, INTERLISP, MACLISP, ZETALISP, GOLDEN COMMON LISP y FRANZLISP (basado en UNIX).

Conceptos Básicos de LISP.

- **Listas:** La estructura más importante es la lista. Una lista es una secuencia de números, cadenas de caracteres o incluso otras listas.
- **La función:** Cada función LISP y cada programa LISP tiene estructura de lista. Los programas no pueden distinguirse de manera sintáctica de los datos. LISP ofrece sus propias funciones básicas.
- **Forma de trabajo:** LISP es un lenguaje funcional. Ofrece la posibilidad de realizar definiciones recursivas de funciones. La unión de procedimientos se realiza de forma dinámica, es decir en plena ejecución, y no como en otros lenguajes de programación. El sistema realiza automáticamente una gestión dinámica de memoria.

LISP tiene características que serán referidas posteriormente por CLIPS, por ejemplo el hecho de ser basado en reglas y el uso de paréntesis.

CLIPS no utiliza encadenamiento hacia atrás, sin embargo lo emula tal como PROLOG lo usaría. Tanto el encadenamiento hacia adelante como el encadenamiento hacia atrás son explicados en la sección 3.6.

4.5 PROLOG.

PROLOG (**PRO**gramación **LOG**ica) su idea básica es expresar declaraciones lógicas como sentencias en un lenguaje de programación. La comprobación de un teorema usando dichas declaraciones puede ser una forma para ejecutarlas. Por lo tanto la lógica en sí puede ser usada directamente como un lenguaje de programación.

PROLOG está basado en un subconjunto de la lógica de primer orden (cálculo del predicado o lógica predicativa). Además tiene la ventaja de contar con una poderosa máquina de inferencias. Más aún, el algoritmo usado por PROLOG es más poderoso que el simple algoritmo de correspondencia de patrones usado comúnmente en LISP en la representación del conocimiento a través de reglas de producción. El hecho de que PROLOG está basado en lógica lo hace diferente a los demás lenguajes. Un programa en PROLOG consta de una serie de sentencias lógicas, puede entenderse declarativamente; es decir, que puede entenderse independientemente de la forma en que será ejecutado. Los lenguajes tradicionales sólo pueden ser entendidos considerando lo que sucede cuando el programa es ejecutado en una computadora. Algunas variantes de PROLOG son: MPROLOG, ARITY PROLOG, QUINTUS PROLOG y TURBO PROLOG.

PROLOG se aplica como lenguaje de desarrollo en aplicaciones de Inteligencia Artificial. Aunque LISP es el lenguaje de Inteligencia Artificial más usado en los Estados Unidos, esto es debido a la existencia de ambientes de programación más sofisticados y estaciones de trabajo de Inteligencia Artificial especializadas.

PROLOG permite que un programa sea formulado en pequeñas unidades, cada una de las cuales cuenta con una lectura declarativa natural; en contraste el tamaño y la

anidación múltiple de funciones en un programa en LISP son barreras para la lectura de dicho programa. Además, la capacidad para correspondencia de patrones de PROLOG es un componente extremadamente útil. Aunque, PROLOG tiene algunas desventajas.

Los argumentos, a favor y en contra, de LISP y PROLOG van a continuar por algún tiempo. Mientras tanto, se han hecho algunos intentos para combinar PROLOG y LISP.

4.6 Necesidades de Hardware para Lisp y Prolog.

La elección del software con frecuencia es determinada por el hardware que este necesita para su procesamiento y poder de memoria, los cuales podrían ser una limitante significativa para el desarrollo de algunos sistemas expertos. La ejecución eficiente de LISP requiere de arquitecturas de hardware altamente especializadas. Estas arquitecturas están disponibles comercialmente desde hace algún tiempo y se conocen como *Máquinas LISP*. A pesar de que existen implementaciones de LISP para una gran variedad de computadoras convencionales, el desempeño puede ser marginal para aplicaciones comerciales complejas o de gran escala. De cualquier modo, el desarrollo de estaciones de trabajo dedicadas a la Inteligencia Artificial, tales como las Máquinas LISP, junto con el progreso de los componentes semiconductores y nuevas arquitecturas de computadoras han formado la base para una rápida adaptación al mundo real de las estaciones de trabajo de Inteligencia Artificial.

Las máquinas dedicadas a la Inteligencia Artificial tienen procesadores cuyo código de máquina está especialmente diseñado para obedecer instrucciones útiles a LISP. Este tipo de máquinas soportan un solo usuario o un pequeño grupo de ellos,

brindando un ambiente completo de programación, el cual incluye editores, depurador de errores y una interfaz de trabajo de red.

Características de las Estaciones de Trabajo de Inteligencia Artificial.

A continuación se listan las siete características más importantes de las estaciones de trabajo de Inteligencia Artificial.

1. A diferencia del tiempo compartido de una computadora regular, una estación de trabajo de Inteligencia Artificial permite ser usada por un solo usuario o por un pequeño grupo de usuarios.
2. Permiten muy altas velocidades de procesamiento de los programas de LISP.
3. Requieren una gran cantidad de memoria; ya que los programas de Inteligencia Artificial usualmente requieren más memoria RAM y más espacio de almacenamiento en disco que otros programas. Dichos requerimientos son provistos por las estaciones de trabajo de Inteligencia Artificial.
4. Proveen alta resolución que permite desplegar más texto en un tiempo.
5. Cuentan con un teclado especializado, el cual tiene cerca de dos docenas extra de teclas para mejorar la productividad del programador e incrementar la velocidad de uso.
6. Hacen uso del ratón para acelerar la comunicación sin teclado.

Una máquina LISP equipada con procesadores para LISP y UNIX pueden proveer un ambiente para PROLOG y para software basado en UNIX. Esto permite que las estaciones de trabajo sean usadas para cómputo de propósito general. Algunos programadores han cambiado a LISP o PROLOG por lenguajes convencionales como C, o han optado por desarrollar sistemas expertos en un hardware especial que soporte LISP

o PROLOG y después los ejecutan en máquinas de Inteligencia Artificial menos poderosas o en computadoras estándar. Muchos sistemas expertos pueden correr en computadoras regulares. La situación típica en donde dichas máquinas LISP muestran sus limitantes es cuando muchos usuarios corren el mismo programa al mismo tiempo, ya que no hay suficiente memoria para obtener más de cinco o diez copias de un programa en LISP o PROLOG.

Las máquinas LISP están diseñadas para correr más rápido que las máquinas convencionales. Si son usadas correctamente las máquinas LISP pueden tener muchas ventajas. Más aún, la entrada de datos, la verificación del tipo de datos y otros deberes de programación pueden mejorar dramáticamente. Sin duda, máquinas simbólicas son mejores para construcción y la ejecución de sistemas expertos.

4.7 Aspectos Representativos para la Elección del Software para el Desarrollo del Sistema Experto *Robot.clp*.

La selección del software, en general, es un asunto complicado debido a los frecuentes cambios en la tecnología y los muchos criterios contra los cuales es comparado el software a ser utilizado.

En esta sección se presentarán los aspectos más importantes que fueron tomados en cuenta a la hora de seleccionar el lenguaje de programación para llevar a cabo el desarrollo del sistema experto *Robot.clp*. Para llevar a cabo el desarrollo de dicho sistema experto se eligió el lenguaje CLIPS, cuya descripción es presentada con detalle en el capítulo seis. Pero sus características más relevantes son:

Características de CLIPS (*C Language Integrated Production System)*

1. CLIPS fue escrito en C y por eso es un código de gran rapidez y sobre todo de gran portabilidad, ya que cualquier máquina que cuente con un compilador C es capaz de soportarlo.
2. Otra ventaja de CLIPS es que ha sido diseñado para una completa integración con otros lenguajes tales como Ada y C, esto quiere decir que puede ser llamado desde un lenguaje estructurado, ejecutar su función y regresar el control al programa que lo mandó llamar. Debido a que el código estructurado puede ser definido como funciones externas y llamado desde CLIPS, cuando el código externo completa su tarea, regresa el control a CLIPS.
3. Otra gran ventaja de CLIPS es de que además de que las 25,000 líneas de código fuente de CLIPS son totalmente gratuitas, el código fuente es totalmente abierto para que el programador pueda adecuarlo a sus necesidades.
4. La ventaja de usar un compilador C es que éste puede ser instalado en computadoras que soporten las plataformas: Windows, UNIX y Linux (variante de UNIX).
5. CLIPS permite usar el esquema de representación del conocimiento por medio de reglas de producción.
6. Debido a que fue escrito en C permite hacer interfaces con otras aplicaciones, como es el caso del servidor SQL Sybase.

Debido a las ventajas que tiene CLIPS de portabilidad, velocidad y bajo costo en comparación con LISP y PROLOG se eligió CLIPS para el desarrollo de la interfaz descrita en el capítulo nueve del presente trabajo. Además de que es un software abierto, ya que permite ser modificado desde su código fuente. Además de que CLIPS provee tres paradigmas de programación: reglas de producción, Programación Orientada a Objetos y

programación estructurada, lo cual significa que es posible resolver un mismo problema de diferente forma.

CAPÍTULO 5

CLIPS.

5.1 Introducción.

En capítulos anteriores se han presentado las definiciones, la terminología y los conceptos teóricos necesarios para entender qué es un sistema experto. Aunque estos conceptos teóricos fueron esenciales para la construcción del sistema experto *robot.clp*, el cual hace uso de la interfaz CLIPS-Sybase y es discutido en el capítulo 6, fue necesario considerar algunos aspectos prácticos para llevar a cabo el desarrollo de dicho sistema.

La principal finalidad del presente capítulo es describir mediante fragmentos del código del sistema *robot.clp*, los elementos básicos de CLIPS que intervienen en el desarrollo de cualquier sistema experto que utiliza como herramienta a CLIPS, asimismo serán descritos los aspectos prácticos, como son los comandos más importantes de dicha herramienta utilizados para el desarrollo del Sistemas Expertos(SE).

5.2 CLIPS.

CLIPS, cuyas siglas en inglés significan *C Language Integrated Production System. E*s un lenguaje de computadora desarrollado por la NASA (*National Aeronautics & Space Administration*), centro espacial Lyndon B. Johnson, el cual fue diseñado para escribir aplicaciones llamadas sistemas expertos.

Se dice que CLIPS es una herramienta para los sistemas expertos porque provee un ambiente completo que incluye características tales como un editor integrado y una herramienta de eliminación de errores o depuración, dichas características facilitan el desarrollo de software que permite modelar el conocimiento o pericia humana.

Una de las ventajas de CLIPS es que debido a que fue escrito en C es un código de gran rapidez y sobre todo de gran portabilidad, ya que cualquier máquina que cuente con un compilador C puede ser utilizada para instalar CLIPS.

Otra ventaja de CLIPS es que ha sido diseñado para una completa integración con otros lenguajes tales como Ada y C, esto quiere decir que puede ser llamado desde un lenguaje procedural, ejecutar su función y regresar el control al programa que lo mando llamar. Debido a que el código procedural puede ser definido como funciones externas y llamado desde CLIPS, cuando el código externo completa su tarea, regresa el control a CLIPS.

Elementos Básicos de CLIPS.

CLIPS está compuesto por tres elementos básicos, los cuales son listados a continuación.

1. *Fact-list* o lista de hechos, es la memoria global de datos.
2. *Knowledge-base* o base de conocimientos, la cual contiene todas las reglas.
3. *Inference-engine* o máquina de inferencias, esta parte controla la ejecución de las reglas que se encuentren en la base de conocimientos.

Un programa escrito en CLIPS está compuesto por una base de conocimientos(reglas y hechos), y una máquina de inferencias que determina cuales reglas y en que momento serán ejecutadas. En las secciones 5.3, 5.4 y 5.5 son explicados respectivamente cada uno de los tres elementos listados anteriormente.

Representación del Conocimiento en CLIPS.

Existen tres maneras de representar el conocimiento en CLIPS. La primera de ellas es mediante reglas, las cuales están orientadas hacia el conocimiento heurístico basado en la experiencia.

La segunda forma de representar el conocimiento es por medio de funciones, las cuales están orientadas hacia el conocimiento procedural.

Y finalmente es posible representar el conocimiento por medio de programación orientada a objetos, dirigida también hacia el conocimiento procedural. Clases, manejadores de mensajes, abstracción, encriptación, herencia, polimorfismo y otras características son soportadas para proveer un completo ambiente orientado a objetos.

Para representar el conocimiento preliminar del sistema experto *Robot.clp* se utilizó un árbol de decisión, el cual se ilustra en la figura 2-3. Dicho esquema de representación sirvió como base para poder desarrollar el sistema experto *Robot.clp*, ya que el conocimiento capturado a través del árbol de decisión fue transformado en reglas y en funciones dentro de CLIPS. Dichas reglas y funciones, las cuales componen el código del sistema experto serán discutidas en el capítulo siguiente.

5.3 Hechos (Fact-list)

Los hechos o *facts* son uno de los elementos básicos que componen cualquier sistema experto escrito en CLIPS, como es el caso del sistema experto *robot.clp*, ya que para poder resolver el problema de saber en que parte de la casa se encuentra un objeto o una persona, el programa *robot.clp* debe contar con datos con los cuales pueda trabajar. La casa está representada por la base de datos, la cual será discutida en el

capítulo ocho, de tal forma que el sistema experto lo único que tiene que hacer es decidir que tipo de *query* o búsqueda debe efectuar sobre la base de datos dependiendo de qué o quién le haya sido solicitado encontrar. Para que el sistema experto *Robot.clp* pueda tomar una decisión sobre que tipo de búsqueda ejecutar es necesario que cuente con datos que le indiquen cuál es la opción correcta. A dichos datos en CLIPS se les conoce como *facts* o hechos. Un hecho está constituido por uno o más campos encerrados entre paréntesis como se puede observar en el hecho mostrado como ejemplo enseguida. Un hecho utilizado por el sistema experto *Robot.clp* es el que se muestra a continuación.

```
(human-choice rooms)
```

El hecho anterior consta de dos campos: *human-choice* y *rooms*. El primero de dichos campos indica que tipo de hecho es, en este caso se trata del tipo de opción elegida por el usuario. En el caso del ejemplo anterior, el hecho mostrado le indica al sistema que el usuario eligió la opción *rooms,* es decir, el usuario desea conocer información acerca de una habitación determinada como por ejemplo su ubicación dentro de la casa, las personas que se encuentran dentro o los objetos que están contenidos en ella. Enseguida se muestra la forma en que un hecho es introducido dentro de la *fact-list* o memoria global de CLIPS.

Introducción de un Hecho en la Memoria Global de Datos de CLIPS.

A continuación se muestra la forma en que un hecho es introducido mediante la función *(assert)* a la lista de hechos o memoria global de datos.

```
(assert (get-human-option))
(assert (human-choice (read))
```

Aunque CLIPS admite como un hecho cualquier combinación de campos, los hechos deben ser representados de una manera significativa para tener un buen estilo de programación. El primero de los hechos mostrados anteriormente es introducido a la lista de hechos de manera directa, es decir, el hecho que será ingresado es *get-human-option)*, el cual activará a la regla que tiene la finalidad de mostrar al usuario una lista de opciones para que le indique a *robot.clp* la opción que desee.

En el segundo caso, se muestra la manera de insertar un hecho de manera indirecta mediante la función *(read)*, la cual permite que la información que es escrita desde el teclado sea leida para que posteriormente dicha información sea insertada en la lista de hechos por medio de la función *(asser)t*. La función *(read)* no es una función de propósito general que lee todo lo que sea escrito desde el teclado, es decir, *(read)* únicamente puede leer un campo.

Por ejemplo, si el usuario teclea la letra 'a', el hecho que será introducido en la lista de hechos es *(human-choice a)*, el cual representa que la opción 'a' ha sido elegida por el usuario.
A cada hecho que es insertado dentro del *fact-list* le es asignado un identificador único. Dicho identificador consta de la letra 'f' y es seguida por un número entero, el cual es llamado el índice del hecho. CLIPS no acepta duplicados de hechos. Por lo tanto el intentar insertar en la lista de hechos un segundo hecho *get-huma-option*) no tendría resultado alguno.

Es importante notar que el identificador que es asignado en la lista de hechos no necesariamente es secuencial. Así como existe una manera de insertar un hecho, existe una forma de remover o quitar uno mediante la función *(retract)*. Cuando se remueve un

hecho se remueve también su identificador de la lista de hechos, es por eso que los identificadores de los hechos en la lista de hechos no siempre son secuenciales.

5.3.1 Construcción de *deffacts*

En ocasiones es muy conveniente tener la habilidad de insertar un conjunto de hechos, particularmente es útil para introducir aquellos hechos que se conocen desde antes de que un programa sea ejecutado, tal es el caso de los hechos que representan a las opciones que se presentan en el menú inicial del sistema *robot.clp*, es decir, el sistema conoce el significado de las letras a, b, c, d y e porque dicho significado le es asignado cuando se introduce mediante una declaración de *deffacts* los hechos que se relacionan con cada una de las letras. La declaración *deffacts* que hace lo anterior es la siguiente.

```
(deffacts information
        (valid-answer actor a)
        (valid-answer inanimate3 b)
        (valid-answer inanimate2 c)
        (valid-answer inanimate d)
        (valid-answer room e))
```

El nombre de la declaración de deffacts anterior es *information* y sirve para asignarle a un mismo campo diferentes hechos, en este caso el campo llamado *valid-answer* contiene los hechos a, b, c, d y e, los cuales sirven para indicar las respuestas válidas que puede dar el usuario. A cada letra se le relaciona con una palabra que describe el tipo de información que se puede solicitar, por ejemplo la letra 'a' está relacionada con la palabra *actor*, es decir, al seleccionar la letra 'a' el sistema entiende que lo que se desea es conocer información acerca de un actor o persona que se encuentra dentro de la casa.

Una vez que haya sido cargado por CLIPS. Después del nombre o comentario están los hechos que serán ingresados en la lista de hechos por medio de esta

declaración de *deffacts*. Los hechos en una declaración de deffacts son insertados usando el comando *reset*, cuya sintaxis es:

(reset)

una vez que sea ejecutado el comando anterior serán introducidos a la lista de hechos, todos los hechos que hayan sido declarados en la construcción *deffacts*.

El comando *reset* es la clave para iniciar o re-inicializar un sistema experto en CLIPS. Cuando un *reset* es ejecutado se remueven todas las reglas activadas que se encuentran en la agenda, así como también son removidos todos los hechos que se encuentren en la lista de hechos, una vez hecho lo anterior se introducen todos los hechos .

5.3.2 *Deftemplates*

Un *deftemplate*, es análogo a una estructura de registro en Pascal, es decir, define a un grupo de campos, los cuales se relacionan entre sí. En general con un *deftemplate* pueden ser definidos muchos tipos de *objetos*. El término objeto se refiere a algo que puede ser definido por medio de sus atributos, como por ejemplo una persona cuyos atributos podrían ser su nombre, lugar en el que se encuentra dentro de una casa, posición dentro de la habitación en la que se encuentra y acción que está ejecutando.

Los atributos que definen a un objeto son el grupo de hechos relacionados entre sí, los cuales pueden ser descritos, por medio del uso de un def*template*.

Como ejemplo de una relación def*template*, considérese el def*template* *actors_information* utilizado en el sistema experto *robot.clp*. En donde *actors_information* es el nombre de la relación, el cual indica que se trata de la información de los actores o personas que se encuentra dentro de una casa. Dicha información consta de siete

campos. El primero de ellos es *rname* que es el nombre de la habitación en donde se encuentra el actor, los campos *x, y, z* que representan las coordenadas del actor dentro de dicha habitación, el campo *action* indica la acción que está realizando el actor, el campo *wearing* indica la ropa con la que está vestido el actor y por último el campo *carry* en el cual se indica el o los objetos que tenga en las manos el actor. Para relacionar la información de un actor anteriormente mencionada, se definió un *deftemplate* de la siguiente forma.

```
(deftemplate actors_information     ; nombre de la relación de deftemplate
 (field rname                       ; nombre del campo
   (type STRING) )                  ; tipo del campo
 (field action                      ;nombre del campo
   (type STRING) )                  ; tipo del campo
 (field wearing                     ; nombre del campo
   (type STRING) )                  ; tipo del campo
 (field carry                       ; nombre del campo
   (type STRING) )                  ; tipo del campo
 (field x                           ; nombre del campo
   (type INTEGER) )                 ; tipo del campo
 (field y                           ; nombre del campo
   (type INTEGER) )                 ; tipo del campo
 (field z                           ; nombre del campo
   (type INTEGER) ))                ; tipo del campo
```

En el ejemplo anterior los componentes son estructurados de la siguiente forma:

- Un nombre de relación del *deftemplate*.
- Atributos llamados *fields* o campos
- El tipo de campo, el cual puede ser cualquiera de los siguientes cinco tipos: SYMBOL, INTEGER, NUMBER, FLOAT y STRING. NUMBER es equivalente a INTEGER y FLOAT.

Por lo tanto, un *deftemplate* es una estructura primitiva, la cual es definida por su nombre de relación y uno o más campos. El *deftemplate* mostrado en el ejemplo tiene siete campos llamados *x, y, z, rname, action, wearing* y *carry*.

Una vez que un *deftemplate* es definido es posible asignar valores a los campos de manera explícita, como se muestra a continuación.

```
CLIPS> (assert (actors_information (rname kitchen) (action eating) (wearing red dress)
       (carry a-sandwich) (x 6) (y 4) (z 0)))
```

Al ejecutarse el comando anterior son introducidos a la memoria global dentro de la estructura *deftemplate actors_information* los hechos relacionados con la información de un actor. [23]

5.4 Reglas (*Knowledge-base*)

Para que un sistema experto sea capaz de realizar un trabajo útil además de los hechos necesita tener reglas, las cuales conforman lo que se conoce como *base de conocimientos* de un sistema experto y son, además, uno de los tres elementos básicos de CLIPS.

El formato general de una regla es el mostrado a continuación.

```
(defrule <nombre-de-la-regla> [<comentario opcional>]
<<patrones>>          ; lado izquierdo (LHS) de la regla
=>
<<acciones>>)         ; lado derecho (RHS) de la regla.
```

Toda la regla debe estar encerrada entre paréntesis, así como también cada uno de los patrones y acciones de la regla deben estar encerradas entre paréntesis. Una regla puede tener múltiples patrones y acciones, si estos últimos están anidados debe tenerse cuidado de que los paréntesis estén correctamente balanceados.

Enseguida se muestra un ejemplo de una de las reglas que conforman el sistema experto *robot.clp*, el cual es capaz de brindar información referente al contenido de una casa. Dicha información se divide en cinco tipos, ya que el usuario puede solicitar información acerca de:

a) Una persona o actor.

b) Un objeto que contiene a otros objetos dentro de sí mismo (como el refrigerador).

c) Un objeto que se encuentra dentro de otro objeto (como la leche).

d) Un objeto que ni es contenido de otro ni contiene dentro de sí alguno (como la computadora).

e) Una habitación de la casa.

El pseudocódigo de la regla que se muestra a continuación sirve para desplegar el menú que es desplegado cuando se ejecuta *robot.clp* y es la encargada de leer la opción que indique que tipo de información de la presentada en el menú se desea conocer.

SI el usuario tecleó algo
ENTONCES leer lo que se tecleo e insertarlo en *human-choice*

La regla anterior puede escribirse en CLIPS como se muestra a continuación:

```
(defrule print-options  "ésta es una regla de ejemplo"
  (get-human-option)
  =>
(printout t "(a) Information about an actor " crlf "
            (b) Inanimate objects with content" crlf "
            (c) Inanimate objects inside another object " crlf "
            (d) Inanimate objects without content (and not inside in another object)"crlf"
            (e) Rooms" crlf )

                                    (assert (human-choice (read))))
```

El encabezado de una regla, como se puede observar en la regla anterior, consiste de tres partes. Una regla siempre debe comenzar con la palabra clave **defrule**, enseguida debe estar el nombre de la regla, en el caso del ejemplo, el nombre de la regla es *print-options*. El nombre puede ser cualquier palabra válida para CLIPS. Si una regla es introducida con un nombre que es igual al de una regla existente, entonces la nueva regla reemplazará a la regla que existía con anterioridad.

Después del nombre de la regla, va un comentario opcional entre comillas dobles, como se muestra en la regla *print-options* en donde el comentario es "ésta es una regla de

[21] *Cfr. Ibidem* .Pags.84-87

ejemplo". El comentario opcional normalmente es utilizado para describir el propósito de la regla, o para cualquier otra información que el programador desee.

A diferencia de los comentarios que inician con un punto y coma, el comentario que sigue al nombre de una regla no es ignorado y puede ser desplegado con el resto de la regla usando el comando *pprule* (pretty print rule).

Después del encabezado de una regla hay cero o más *patterns* o elementos condicionales. Cada elemento consta de uno o más campos. En la regla *print-options*, sólo hay un elemento condicional o patrón que es (get-human-option), el cual cuenta con un sólo campo, ya que las palabras que conforman el elemento condicional están separados por guiones lo que hace que las tres palabras sean tomadas por CLIPS como si fueran un sólo campo. CLIPS intenta hacer corresponder los patrones de las reglas con los hechos contenidos en la lista de hechos. Si se encuentra correspondencia a todos los patrones de una regla, en este caso con el patrón (get-human-option), con algunos de los hechos de la *fact-list* o lista de hechos, la regla es *activada* y colocada en la *agenda*. La agenda es la colección de reglas activadas, es posible que existan cero o más reglas en ella. Si una regla carece de patrones o elementos condicionales, entonces el patrón especial *(initial-fact)* será agregado como patrón para esa regla. El elemento condicional *(initial-fact)* es insertado de manera automática cuando el comando *(reset)* es ejecutado.

El símbolo => que sigue a los patrones en una regla es llamado una *flecha*. Se forma con un signo de igual seguido por un signo de mayor que. La flecha indica el comienzo de la parte ENTONCES de una regla SI-ENTONCES. La parte que se encuentra antes de la flecha se conoce como el lado izquierdo de la regla y la parte que viene después de la flecha es llamada el lado derecho de la regla.

La última parte de una regla es la lista de acciones que serán ejecutadas cuando la regla se *dispare* o ejecute. Es posible que una regla no contenga acciones, aunque no sería particularmente útil.

En la regla que se utilizó como ejemplo, una de las acciones es desplegar el menú que muestra los diferentes tipos de información que el usuario puede solicitar. La otra acción consiste en leer, mediante la función (read), lo que es tecleado por el usuario y lo guarda en human-choice. Un programa normalmente cesa su ejecución cuando ya no hay regla en la agenda, es decir, cuando ya no hay regla activadas.

5.5 Activación (Inference-engine)

En las secciones anteriores se han explicado dos elementos fundamentales de CLIPS para el desarrollo de un sistema experto, los hechos y las reglas. El tercer y último elemento básico de CLIPS es *la máquina de inferencias*, la cual es la encargada de controlar la ejecución de las reglas que están contenidas en la base de conocimientos de un sistema experto, es decir, se encarga de decidir qué reglas son ejecutadas basándose en los hechos disponibles en la memoria global de datos.

Un programa hecho en CLIPS puede ser ejecutado mediante el comando *run*, la sintaxis de este comando es:

(run)

Cuando se ejecuta un programa de CLIPS, si sólo hay una regla en la agenda y existe en la lista de hechos algunos hechos que tengan correspondencia con los elementos condicionales de la regla, entonces esa regla será disparada. La regla *print-options*, mostrada en la sección anterior, cuenta con un sólo patrón, el cual es satisfecho por el hecho *(get-human-option)* que se encuentra el la lista de hechos, esta regla deberá dispararse cuando el programa sea ejecutado.

Si la regla *print-options* es introducida después de que el hecho (get-human-option) sea insertado a la lista de hechos y después se intenta ejecutar la regla mediante un comando *run* la regla no será disparada. La razón de que no se dispare la regla tiene que ver con la forma en que CLIPS está diseñado. El diseño de CLIPS es tal que las reglas sólo ven los hechos que han sido introducidos después de ellas. Por lo tanto, las reglas que sean introducidas nuevamente no podrán ver los hechos que se encuentran actualmente en la lista de hechos. Sólo los nuevos hechos que sean introducidos serán vistos por dichas reglas. Esto quiere decir que una regla puede únicamente ser activada por los hechos que sean introducidos después de la regla. Es por ello que siempre primero se carga el archivo en donde están escritas las reglas que componen el sistema experto y después mediante el comando (reset) se inserta el hecho (initial-fact), el cual causa que la regla que no tenga ningún elemento condicional en la parte izquierda sea activada.

Activación de una regla.

Para activar la regla *print-options*, el hecho *(get-human-option)*, debe ser ingresado después de que la regla ha sido insertada. Si el hecho (*get-human-option*) se encuentra ya en la lista de hechos, introducir un nuevo hecho (*get-human-option*) no tendrá ningún efecto. Para poder re-insertar el hecho una vez más, la versión original de ese hecho deberá ser retractado. Una vez que esto sea realizado, un nuevo (*get-huma-option*) puede ser insertado, puesto que no ocurrirá una duplicación. Esta nueva versión del hecho tendrá un índice diferente al del hecho original y la regla *print-options* será colocada en la agenda.

La lista de reglas en la agenda puede ser desplegada con el comando *agenda*. La sintaxis de dicho comando es:

(agenda)

Si no hay activaciones en la agenda, el prompt de CLIPS reaparecerá después de que el comando *agenda* es emitido. Si la regla *print-options* ha sido activada por el hecho (get-human-options) con un índice de 2, un comando *agenda* produciría la siguiente salida:

```
CLIPS> (agenda)
0     emergencia-fuego    f-2
CLIPS>
```

El cero indica la prioridad de la regla en l agenda. Después de la prioridad viene el nombre de la regla seguido por el identificador del hecho que activó a la regla, en este caso sólo hay un identificador, f-2.

Reglas y Refracción.

Una vez que se cargue, mediante el comando (load), el archivo *robot.clp*, el cual contiene la regla *print-options*, y se ingrese el hecho *(get-human-option)* la regla será activada y colocada en la agenda. Por lo tanto, al ejecutar el comando (*run*), se producirá el despliegue en pantalla:

```
CLIPS> (load robot.clp)
CLIPS>(assert (get-human-option))
CLIPS>(run)

        (a) Information about an actor
        (b) Inanimate objects with content
        (c) Inanimate objects inside another object
        (d) Inanimate objects without content (and not inside in another object)
        (e) Rooms
```

Además de desplegarse el menú anterior, el sistema espera que el usuario teclee alguna letra para leerla y guardarla en (*human-choice)*. A pesar de que ahora ya existe una regla y un hecho, el cual satisface dicha regla, si un comando *run* es ejecutado otra vez, éste no producirá resultado alguno. Al ejecutar un comando *agenda* se podrá verificar que efectivamente, ninguna regla ha sido disparada porque no hay reglas en la agenda. La regla no se dispararía nuevamente debido a la forma en que CLIPS ha sido diseñado. CLIPS fue programado con características de una célula nerviosa (o neurona). Después

de que una neurona transmite un impulso nervioso (o se disparará), ninguna cantidad de estimulación hará que la neurona se dispare otra vez por algún tiempo. Este fenómeno es llamado *fenómeno de refracción* y es muy importante para los sistemas expertos.

Sin la refracción, los sistemas expertos podrían caer siempre en ciclos triviales. Esto es, una vez que una regla fuera disparada, se mantendría ejecutándose con el mismo hecho una y otra vez. En el mundo real, los estímulos que son la causa de un disparo, podrían eventualmente desaparecer. Por ejemplo, el fuego podría eventualmente ser extinguido por el sistema de rocío o apagarse por sí mismo. Sin embargo, en el mundo computacional, una vez que un hecho es agregado en la lista de hechos, este permanece ahí hasta que sea explícitamente removido.

Para lograr que la regla *print-options* se dispare otra vez se tendría que modificar dicha regla de tal modo que se retracte el hecho (*get-human-option*) y posteriormente sea insertado nuevamente dentro de la misma regla. Básicamente, CLIPS recuerda los identificadores de los hechos que hicieron funcionar a una regla y que no será activada otra vez con exactamente la misma combinación de identificador y hecho.

5.6 Comandos Usados con las Reglas

Despliegue de Reglas Contenidas en la Base de Conocimientos

El comando *rules* es usado para desplegar la lista actual de reglas mantenidas por CLIPS. Su sintaxis es:

```
(rules)
```

El comando *pprule (pretty print rule)* es usado para desplegar el texto de representación de una regla. Su sintaxis es:

```
(pprule <nombre-de-la-regla>)
```

CLIPS coloca las diferentes partes de una regla en renglones diferentes para facilitar la lectura de la misma.

Cargado de Reglas desde un Archivo

Un archivo de reglas creado en un editor de texto puede ser cargado en CLIPS usando el comando *load*, cuya sintaxis es:

(load <nombre-del-archivo>)

donde <nombre-del-archivo> es una cadena que contiene el nombre del archivo a ser cargado.

5.6.1 El Comando *watch*.

El comando *watch* es útil para la depuración de un programa, su sintaxis es:

(watch facts)

(watch rules)

(watch activations)

(watch all)

Los hechos, las reglas y las activaciones pueden ser vistas en cualquier combinación para proveer una cantidad apropiada de información de depuración. El comando *watch* puede ser usado más de una vez para ver más de una característica de la ejecución de CLIPS. Con la combinación (watch all) es posible ver todas las características visibles al mismo tiempo, es decir, es posible ver los hechos, las reglas y las activaciones.

Si los hechos están siendo vistos, CLIPS automáticamente envía un mensaje indicando que una actualización ha sido llevada a cabo a la lista de hecho o fact-list, indicando si los hechos han sido insertados o retractados.

Cuando se están viendo las activaciones, CLIPS envía un mensaje indicando si una activación ha sido agregada o removida de la agenda. Si lo que se está viendo son las reglas, CLIPS envía un mensaje diciendo si una regla ha sido disparada. Cuando se ven las activaciones no se causa que un mensaje sea desplegado en pantalla para indicar cuando una regla es disparada. Además si las reglas son vistas mientras se carga un programa, CLIPS desplegará un mensaje tal como el siguiente:

Compiling rule: cortar-la-electricidad +j+j+j

Para cada regla que está siendo cargada, en este momento CLIPS compila el código fuente. La cadena "+j+j+j" al final del mensaje es alguna información de CLIPS acerca de la estructura interna de las reglas compiladas.

Los efectos de un comando *watch* puden ser anulados por el correspondiente comando *unwatch*, cuya sintaxis es la siguiente:

(unwatch {hechos, reglas, activaciones, todo})

CAPÍTULO 6

IMPLEMENTACIÓN DEL SISTEMA EXPERTO ROBOT.CLP

6.1 Introducción.

Este capítulo tiene como finalidad describir el desarrollo y la implementación del sistema experto al que se denominó *robot.clp*. El objetivo del desarrollo de *robot.clp* es aplicar tanto las funciones ya existentes de CLIPS como las funciones provistas por la interfaz CLIPS-Sybase, las cuales son explicadas en el capítulo 8.

6.2 Definición del Problema.

Básicamente el problema consistió en desarrollar un sistema experto cuya finalidad fuera la de proporcionar la información concerniente a objetos contenidos en una casa, como por ejemplo ubicación y contenido del refrigerador, ubicación dentro de la casa de la sala, ubicación de una persona, etc. Dicha información está contenida dentro de una base de datos a la cual se le llamó *house* y es descrita en el capítulo 7. Para poder realizar la búsqueda de la información solicitada el sistema experto debía hacer uso de las funciones proporcionadas por la interfaz CLIPS-Sybase, las cuales son descritas en el capítulo 8.

A continuación se muestran los cinco diferentes tipos de información que el sistema tenía que ser capaz de proporcionar.

a) Información acerca de un actor, el término *actor* se refiere a que puede tratarse tanto de una persona como también puede tratarse de un robot que se encuentra dentro de la casa. La información proporcionada es el nombre de la habitación en donde se encuentra el actor, así como las coordenadas que tiene

el actor dentro de dicha habitación, la ropa que está vistiendo, el nombre de los objetos que tenga en las manos y finalmente la acción que está realizando. Por ejemplo si se quisiera saber información acerca del papá, la información proporcionada por el sistema sería:

> El papá se encuentra en la RECÁMARA, en donde sus coordenadas son: (12.06, 2.067, 0), viste un TRAJE AZUL y ZAPATOS NEGROS, tiene un SANDWICH en las manos y está COMIENDO.

En donde, las palabras escritas con letra mayúscula y las coordenadas son los datos obtenidos de la base de datos.

b) Información acerca de un objeto inanimado, cuya principal característica es guardar dentro de sí a otros objetos. La información que debía proporcionarse en este punto es el nombre de la habitación en donde se encuentra el objeto buscado, las coordenadas de dicho objeto dentro de la habitación y el nombre de todos los objetos que están guardados dentro de él. Un ejemplo de la información de este tipo proporcionada por el sistema se muestra a continuación, en donde igual que el ejemplo del inciso anterior las palabras en mayúsculas son los datos obtenidos de la base de datos. Suponiendo que se solicita información acerca del refrigerador se tendría:

> El refrigerador se encuentra en la COCINA, en donde sus coordenadas son: (9.760, 8.280,0), y contiene: HUEVOS, VEGETALES y CARNE.

c) Información acerca de un objeto inanimado que normalmente debería estar guardado dentro de otro objeto. Este tipo de información se refiere a objetos tales como ropa, zapatos, herramientas o comida, los cuales normalmente son guardados dentro de un ropero, una caja de herramientas o el refrigerador, respectivamente. La información proporcionada en este punto es: el nombre del objeto que lo contiene y el nombre de la habitación en donde se encuentra guardado dicho objeto y se proporcionan también las coordenadas que el objeto tiene dentro del objeto en que se localiza. Un objeto inanimado que sería del tipo de objetos descritos en este inciso sería por ejemplo la cebolla. Si se quisiera conocer información acerca de la carne, el sistema proporcionaría lo siguiente:

La CARNE se encuentra dentro del REFRIGERADOR, en donde sus coordenadas son: (9.500, 8.25, 0.875) y el REFRIGERADOR se encuentra en la cocina.

d) Información de un objeto inanimado, cuya principal característica es que ni esté guardado dentro de algún otro objeto ni contenga dentro de sí objeto alguno, como puede ser la computadora, la televisión, etc. La información proporcionada en este punto es el nombre de la habitación en donde se encuentra el objeto, así como sus coordenadas dentro de dicha habitación, su función y el nombre de su propietario. Un objeto del tipo de objeto descrito en este inciso podría ser por ejemplo la computadora. La información proporcionada acerca de dicho objeto sería:

La COMPUTADORA se encuentra en el CUARTO DEL NIÑO, en donde sus coordenadas son (3.636, 2.232, 0).

e) Información acerca de una habitación. Se debía proporcionar la ubicación de cierta habitación dentro de la casa, el nombre de los cuartos adyacentes, el nombre de todos los objetos contenidos dentro de la habitación y el nombre de las personas que se encuentran dentro de ella, así como también las coordenadas tanto de las personas como de los objetos contenidos dentro de la habitación. Por ejemplo si se pidiera información sobre el almacén el sistema proporcionaría la siguiente información:

El ALMACÉN se encuentra a dos metros a la derecha del balcón. Las habitaciones adyacentes al comedor son: el BALCÓN, el CORREDOR y la RECÁMARA DE LA MAMÁ.
Dentro del comedor se encuentran los siguientes objetos:
BICICLETA, CAJA DE HERRAMIENTAS, DESARMADOR y MARTILLO.
Y el/los actor(es) que se encuentra(n) en el almacén es/son: AGENTE2.

6.3 Representación del Problema en CLIPS.

CLIPS puede ser aplicado a la representación de un problema basado en estados y transiciones, ya que los estados pueden ser representados en CLIPS por medio de los hechos, mientras que las transiciones son representadas por medio de las reglas que constituyen al sistema experto.

El problema descrito en la sección anterior, se puede representar en CLIPS por medio de un *Sistema de Producción*, el cual como se definió en la sección 2.5, es un esquema de representación del conocimiento que sirve para hacer razonamiento por medio de las *Reglas de Producción*, las cuales, como se indica en la sección 2.5.1, constituyen uno de los elementos básicos de un sistema de producción

El problema parte de un estado inicial en el que se desea obtener cierta información ya sea sobre una persona u objeto contenido dentro de la casa. Para poder obtener la información deseada únicamente se cuenta con la definición de los cinco diferentes tipos de información descritos en la sección anterior que se puede proporcionar y el estado final al que se quiere llegar es la obtención de la información deseada. En los incisos de la sección anterior en donde se describen los cinco diferentes tipos de información se muestran algunos ejemplos de la forma en que el sistema proporciona la información solicitada.

6.4 Representación del Estado Inicial.

El estado inicial del problema puede ser representado en CLIPS por medio de una declaración de *deffacts*, la cual se explicó en la sección 5.3.1. Dicha declaración permite la introducción del conjunto de hechos conocidos antes de que el programa sea ejecutado, es decir, permite introducir el conjunto de hechos que representan al estado inicial del problema, en este caso dicho estado está definido por las opciones que indican los diferentes tipos de información que puede ser solicitada al sistema experto *Robot.clp*. La declaración *deffacts* que define el estado inicial se realizó de la siguiente manera.

```
(deffacts information
        (valid-answer actor a)
        (valid-answer inanimate3 b)
        (valid-answer inanimate2 c)
        (valid-answer inanimate d)
```

```
(valid-answer room e))
```

Con la declaración *deffacts* anterior, llamada *information,* le es asignado al campo *valid-answer* diferentes hechos, en este caso dicho campo contiene los hechos a, b, c, d y e, los cuales sirven para indicar el tipo de información descrita en la sección 6.2 que se desea conocer. A cada letra se le relaciona con una palabra que describe el tipo de información que se puede solicitar, por ejemplo la letra 'e' está relacionada con la palabra *room*, es decir, al seleccionar la opción representada por la letra 'e' el sistema lo interpreta como que lo que se desea es conocer información acerca de una habitación de la casa.

Además de la introducción de los hechos descritos anteriormente, debe ingresarse también a la lista de hechos de CLIPS otro hecho, el cual será el que disparará la primera regla del sistema experto. Para llevar a cabo la introducción de dicho hecho, se definió la siguiente regla.

```
*****************
* STARTUP RULE *
*****************
(defrule startup
  =>
  (printout t "Lets  make a query to SYBASE!!" crlf crlf)
  (printout t "choose  the letter of the " crlf)
  (printout t "option you desire." crlf crlf)
  (printout t "For example if you want" crlf)
  (printout t "to know information about" crlf)
  (printout t "an actor just choose the letter a. And so on." crlf crlf)
  (assert (get-human-option)))
```

Como se puede observar la regla *startup* no cuenta con ningún elemento condicional en la parte izquierda de la regla, es decir antes de la aparición de la flecha (=>), lo anterior significa que la regla será disparada por causa del hecho (*initial-fact*) y este hecho es ingresado mediante el comando *(reset)*, el cual se ejecuta antes de ejecutar

el programa. Al dispararse la regla *startup,* se producirá un mensaje en pantalla mediante el cual se invita al usuario a que elija alguna de las opciones presentadas, pero además de desplegar dicho mensaje, la regla se encarga de insertar en la memoria global de CLIPS el hecho *(get-human-option)*, el cual es la clave fundamental para iniciar la ejecución de las reglas que constituyen el sistema experto R*obot.clp*.

6.5 Transiciones del Sistema de Producción.

Una vez definido y representado en CLIPS el estado inicial del problema, fue necesario definir las transiciones por medio de las cuales se pudiera llegar al estado final. Las transiciones básicamente lo que hacen es pasar de un estado a otro, es decir, efectúan modificaciones dentro de la lista de hechos de CLIPS. Ya que los hechos son los encargados de representar los estados del problema al ser modificados los hechos se cambia de un estado a otro. A continuación se listan dichas transiciones, las cuales posteriormente serán representadas en CLIPS por medio de reglas.

TRANSICIÓN **DESCRIPCIÓN**

1. Mostrar el menú de opciones que representen los diferentes tipos de información que es posible obtener, leer la opción solicitada por medio del teclado y verificar que la opción seleccionada sea válida, en caso de no serlo volver a solicitar la selección de una opción válida hasta que sea válida.

2. Establecer una conexión con el servidor SQL y con la base de datos *house*, la cual está contenida dentro de dicho servidor. Dependiendo de la opción seleccionada, solicitar el nombre del actor, de la

habitación o del objeto inanimado y guardar dicha nombre en una variable.

3. Enviar como parámetro el contenido de la variable del punto anterior a una función que se encargue de la construcción del código SQL y al mismo tiempo guardar el resultado de dicha función en otra variable, la cual contendrá el código SQL que contendrá el parámetro que determinará sobre quien se ejecutará la búsqueda en la base de datos.

4. Enviar como parámetro el contenido de la variable que contiene el código SQL a la función *dbcmd()*, la cual es provista por la interfaz CLIPS-Sybase, quien se encargará de enviar el código al servidor SQL para que sea ejecutado. Una vez ejecutado el código SQL, llamar a la función *dbquery()*, la cual es otra función provista por la interfaz CLIPS-Sybase, y es la encargada de introducir los datos resultantes de la consulta a la base de datos dentro de la lista de hechos o memoria global de CLIPS.

5. Finalmente mandar desplegar en pantalla los resultados obtenidos de la consulta a la base de datos, es decir, desplegar la información que se deseaba conocer, llegando de esta manera al estado final del problema.

6.5.1 Representación en CLIPS de las Transiciones del Problema.

En la sección anterior fueron definidas las transiciones por medio de las cuales el sistema llega a un estado final o solución del problema, en este caso, el sistema experto R*obot.clp* llega a la obtención de la información solicitada.

Como ya ha sido mencionado con anterioridad, la manera de representar en CLIPS a una transición correspondiente a un sistema de producción es por medio de las reglas, las cuales, como se mencionó en el capítulo seis, constituyen *la base de conocimientos* del sistema experto hecho en CLIPS.

Para poder explicar la representación en CLIPS de cada una las transiciones por las cuales se tiene que pasar para llegar a un estado final se consideró un solo caso, ya que el sistema *robot.clp* es capaz de proporcionar cinco diferentes tipos de información, cada uno de los cuales debe pasar exactamente por las mismas transiciones para poder llegar al estado final. Debido a lo anterior se consideró que bastaría con la explicación de uno de los cinco posibles casos para poder comprender la forma en que se llevó a cabo el desarrollo del sistema experto *robot.clp*.

A continuación se muestran las reglas por medio de las cuales fueron representadas en CLIPS las transiciones descritas en la sección 6.5, por las cuales el sistema debe pasar para el caso en que se desee conocer información acerca de un actor

- **Primera Transición.**

La primera transición, en la cual se muestran el menú de opciones y se lee la opción seleccionada por medio del teclado verificando que la opción seleccionada sea válida y en caso de no serlo se vuelve a solicitar la elección de una opción válida hasta que se trate de una respuesta válida, está representada por las siguientes tres reglas.

```
*************************
* HUMAN CHOICE RULES  *
*************************

(defrule print-options
  (get-human-option)
  =>
(printout t "(a) Information about an actor " crlf
"(b) Inanimate objects with content" crlf
"(c) Inanimate objects inside another object " crlf
"(d) Inanimate objects without content (and not inside in another object)" crlf
"(e) Rooms" crlf )
(assert (human-choice (read))))

(defrule good-human-option
  ?f1 <- (human-choice ?choice)
  (valid-answer ?answer $? =(lowcase ?choice) $?)
 ?f2 <- (get-human-option)
  =>
  (retract ?f1 ?f2)
 (assert (human-choice ?answer)))

(defrule bad-human-option
  ?f1 <- (human-choice ?choice)
  (not (valid-answer ?answer $? =(lowcase ?choice) $?))
  ?f2 <- (get-human-option)
  =>
  (retract ?f1 ?f2)
  (assert (get-human-option)))
```

El hecho *(get-human-option)*, el cual es uno de los hechos que fueron introducidos para representar el estado inicial, es además el único elemento condicional de la regla *print-options*. Debido a que dicho elemento condicional se cumple la regla es colocada en la agenda y se ejecuta, lo cual produce que el menú sea desplegado en pantalla y sea leída desde el teclado la opción seleccionada e introducida como un hecho mediante la instrucción *(assert (human-choice (read)))*, es decir, lo que sea tecleado por el usuario será guardado en el campo llamado *human-choice*.

La regla *good-human-option* cuenta con tres elementos condicionales que deben cumplirse para que la regla pueda dispararse. Dichos elementos son los que se encuentran definidos en la regla antes de la aparición de la flecha (=>).

El primero de dichos elementos, es que dentro de lista de hechos debe encontrarse el hecho *(human-choice ?choice)*, en donde *?choice*, indica que en el campo *human-choice* debe existir algo. El segundo elemento condicional es el encargado de verificar que el contenido del campo *human-choice* sea una respuesta válida, para ello, compara dicho contenido con lo que se definió en el estado inicial como *valid-answer*, es decir contra las letras a, b, c, d y e. Si la opción tecleada por el usuario es una de las letras mencionadas anteriormente entonces este patrón será satisfecho. El tercer y último elemento condicional de la regla es que debe estar en la memoria global de CLIPS el hecho *(get-human-option)*. Una vez que los tres elementos condicionales la regla se cumplen, la regla es disparada y se ejecutan las acciones definidas en la parte derecha de la regla. En este caso primero son removidos los hechos *(get-human-option)* y *(human-choice ?choice)* de la memoria global de CLIPS, esto se hace debido a que dichos hechos son también elementos condicionales de otra regla y, como se explicó en el capítulo 5, una vez que un hecho dispara una regla, no es posible que se dispare otra con el mismo hecho. La segunda acción que es ejecutada al dispararse la regla, es que en el campo *human-option* ahora es insertado una palabra relacionada con la letra seleccionada. Por ejemplo si la letra 'a' es elegida, entonces el hecho que es insertado es: *(human-option actor)*, la introducción de este hecho en la memoria global de CLIPS significa que ha habido un cambio de estado en el sistema de producción, es decir, se pasa del estado inicial al segundo estado, ya que ahora ya se conoce el tipo de información a ser buscada.

Para que la regla *bad-human-option* se ejecute, básicamente se deben cumplir los mismos elementos condicionales de la regla *good-human-option*, excepto que ahora en lugar de verificar que la opción sea igual a alguna de las letras a, b, c, d o e, se verifica que la opción seleccionada no sea ninguna de dichas letras, lo cual significaría que se trata de una opción no válida. Al ejecutarse esta regla, igual que en la regla *good-human-option*, se retractan los hechos *(get-human-option)* y *(human-choice ?choice)* y se ingresa el hecho *(get-human-option)*, con lo que se logra que la regla *print-options* sea disparada nuevamente desplegando así, el menú de opciones para que sea elegida una opción válida, es decir, regresa al estado inicial en donde aún no se conoce que tipo de información va a ser buscada.

- **Segunda Transición.**

En la primera transición se pasó del estado inicial al segundo estado en donde ya se conoce qué tipo de información se desea conocer. Este estado está representado por el hecho *(human-option ?answer)*, en donde, como se explicó anteriormente, la variable *?answer* es sustituida por la palabra relacionada con la letra de la opción que haya sido seleccionada.

Una vez que se conoce el tipo de información a buscar, es necesario saber el nombre del actor o del objeto inanimado, según sea el caso, para poder realizar la consulta a la base de datos *house*. Lo anterior implica hacer otro cambio de estado, es decir implica la modificación de los hechos contenidos en la memoria global de CLIPS, mediante una transición o regla.

Para poder explicar cómo se realiza la segunda transición, se retomará el ejemplo mostrado en la explicación de la regla *good-human-option*, en donde se seleccionó a la

opción 'a' con lo cual se introdujo el hecho *(human-option actor)* en la lista de hechos de CLIPS. La regla *good-human-option* representa a la primera transición, ya que es quien introduce en la memoria global el hecho que representa el segundo estado. En este punto fue necesario crear otra regla que representara a una segunda transición, la cual fuera capaz de obtener el nombre del actor, para el caso particular del ejemplo en cuestión, y al mismo tiempo guardara dicho nombre en una variable para que dicha variable fuera ingresada en la memoria global de CLIPS en forma de hecho. La siguiente regla, llamada *actor_name*, es la encargada de llevar a cabo la segunda transición.

```
(defrule actor_name
(human-choice actor)
=>
(dbopen tesis tesis123 house)
(printout t  "Who are you looking for?" crlf crlf
"Remember that you have to write the actor's name between simple quotes ' ' " crlf)
(assert (parametro5 (read)))
(assert (recieve-actor-param)))
```

Como se puede observar la regla *actor_name* tiene únicamente un patrón o elemento condicional, el cual, según el ejemplo que está siendo explicado, ya ha sido ingresado a la lista de hechos y por lo tanto la regla *actor_name* es colocada en la agenda para ser ejecutada. Las acciones que realiza esta regla al ser ejecutada son: primero establece una conexión mediante la función *(dbopen tesis tesis123 house)*, la cual es provista por la interfaz CLIPS-Sybase, donde *tesis* es el login que permite establecer una conexión con el servidor SQL, *tesis123* es el password y el tercer argumento es el nombre de la base de datos que será consultada, en éste caso es *house*. La siguiente acción a ejecutarse es solicitar se proporcione mediante el teclado el nombre del actor sobre el cual se desea saber información, enseguida se lee el nombre tecleado y se guarda en la variable *parametro5*, la cual es ingresada a la lista de hechos y finalmente se introduce el hecho *(recieve-actor-param)*, el cual servirá para lograr que se ejecute la

regla que representará la tercera transición. Por ejemplo si la palabra *'father'*, es tecleada, entonces el hecho que será introducido en la memoria global es el siguiente:

(parametro5 father)

En resumen, la segunda transición guía al estado en donde además de conocer el tipo de información a ser buscada, se conoce también de manera explícita el nombre del objeto inanimado o actor, según sea el caso, acerca del cual se buscará información.

- **Tercera Transición.**

La tercera transición se encarga de enviar como parámetro el contenido de la variable que contiene el nombre del actor que va a ser buscado a una función que se encargue de la construcción del código SQL mediante la concatenación de todas y cada una de las palabras, incluyendo los espacios en blanco que forman dicho código, de manera que el contenido de la variable en cuestión sea incluida dentro del código SQL. Una vez que se forma el código SQL es necesario guardarlo en otra variable, para que ésta a su vez sea enviada a la función, provista por la interfaz CLIPS-Sybase, encargada de la ejecución de dicho código en el servidor SQL.

Lo anterior logra realizarse por medio de la regla *actor-par*, la cual se presenta a continuación.

```
(defrule actor-par
(recieve-actor-param)
(parametro5 ?param5)
=>
(bind ?x (sqlcod5 ?param5))
(assert (com5 ?x))
(assert (act_query)))
```

La regla anterior es activada debido a que en la memoria global de CLIPS se encuentran los hechos: *(recieve-actor-param)* y *(parametro5 father)*, los cuales cumplen

con los patrones condicionales de la regla *actor-par*. La primera acción que ejecuta dicha regla es la indicada por la siguiente línea.

(bind ?x (sqlcod5 ?param5))

en donde la función *sqlcod5* es llamada y le es enviado como parámetro el contenido de la variable parametro5, es decir, la función *sqlcod5* recibe la palabra 'father'. Al mismo tiempo que le es enviada una variable como parámetro, el resultado de la función es guardado en la variable 'x'. La función *sqlcod5* se presenta a continuación.

```
(deffunction sqlcod5
(?var5)
(str-cat select" "r.rname," "a.action," "a.xa," "a.ya," "a.za," "a.wear," "a.carry" "into" "#temp"
"from" "room" "r," "actors" "a" "where" "a.id_room" "=" "r.id_room" "and" "a.act_name" "="
"?var5))
```

Después de que el nombre del actor sobre el cual se hará la búsqueda, le es enviado a la función anterior, dicha función lo recibe y se encarga de concatenarlo junto con cada uno de las palabras constituyentes del código que será enviado al servidor de base de datos para que lo ejecute dará. Para el ejemplo en cuestión, se tendría el siguiente código SQL, en donde se indica que los datos resultantes de la búsqueda sean colocados en una tabla temporal, *#temp*, de la base de datos *house*.

```
select r.rname, a.action, a.xa, a.ya, a.za, a.wear, a.carry into #temp
from room r, actors a
where a.id_room = r.id_room
and a.act_name = 'father'
```

El código anterior es guardado en una variable llamada 'x', la cual a su vez es ingresada en el campo *com5* dentro de la memoria global de CLIPS en forma de hecho mediante la siguiente línea de código, la cual como puede observarse, es una de las acciones a ser ejecutada por la regla *actor-par*.

(assert (com5 ?x))

Y finalmente es insertado el hecho *(act_query)*, el cual cumple con uno de los elementos condicionales de la regla del sistema que se explica en la cuarta transición.

- **Cuarta Transición.**

La cuarta transición es la encargada de hacer que el código SQL construido en la anterior transición sea ejecutado en el servidor SQL, para lograrlo se construyó la regla *actor-query*, cuyo código es presentado a continuación.

```
(defrule actor-query
(act_query)
(com5 ?x)
=>
(dbcmd ?x)
(if (eq (dbquery "select * from #temp" actors_information) 0)
then (printout  t "The actor who you are looking for is not inside the house,
or probably you wrote incorrectly its/her/his name." crlf crlf)
(dbcmd " drop table #temp")
(assert (determine-actor-again))
else (assert (disp_act))))
```

Puesto que en la regla que representa a la tercera transición fueron insertados tanto el hecho *(act_query)* como el hecho *(com5 ?x)*, los elementos condicionales de la regla *actor_query* son entonces satisfechos y por lo tanto dicha regla es disparada.

La primera acción que se ejecuta al dispararse la regla *actor_query*, es que la variable 'x', la cual contiene el código SQL que será ejecutado en el servidor, es enviada como argumento a la función *dbcmd()*, la cual es una de las funciones provistas por la interfaz CLIPS-Sybase y es explicada con detalle en el capítulo 8. Dicha función envía el código al servidor SQL para que éste ejecute dicho código. Como se puede observar en la construcción del código SQL se indicó, mediante la instrucción *into #temp*, que los resultados obtenidos de la consulta a la base de datos fueran colocados en una tabla temporal llamada *#temp* por lo tanto es necesario llamar a otra función para que se

encargue de traer los datos almacenados en dicha tabla temporal y al mismo tiempo los introduzca dentro de la memoria global de CLIPS.

La función *dbquery()* se utilizó para lograr ejecutar un comando simple de tipo *select*, el cual hace una consulta a la tabla temporal donde fueron colocados los datos resultantes de la consulta a la base de datos *house* y a la vez introduce dichos datos en una estructura de tipo *deftemplate* de CLIPS.

Como se mencionó en el capítulo 5, un *deftemplate* define a un grupo de campos, los cuales se relacionan entre sí. En general con un deftemplate puede ser definido un *objeto*. El término objeto se refiere a algo que puede ser definido por medio de sus atributos, como es el caso de un actor y de los objetos inanimados que se encuentran dentro de la casa. Para el caso de un actor sus atributos son su nombre, nombre de la habitación en la que se encuentra dentro de la casa, posición dentro de la habitación en la que se encuentra, acción que está realizando y la ropa que está usando. El *deftemplate actors_information*, el cual se presenta a continuación, fue definido dentro del código del sistema experto *robot*.clp, para que los datos resultantes de la consulta a la base de datos fueran recibidos por dicha estructura *deftemplate*.

```
(deftemplate actors_information
  (field  rname
    (type STRING) )
  (field  action
    (type STRING) )
  (field  wear
    (type STRING) )
  (field  carry
    (type STRING) )
  (field  xa
    (type INTEGER) )
  (field  ya
    (type INTEGER) )
  (field  za
    (type INTEGER) ))
```

Las siguientes líneas de código son parte de las acciones de la regla *actor-query* y muestra como se utilizó la función *dbquery()* para introducir los datos dentro de la estructura *deftemplate actors_information.* Asimismo, se muestra que si en la tabla temporal se encontraron cero datos, es decir que si no fue encontrada la información que acerca del actor solicitado, se desplegará en pantalla un mensaje en el que será notificado al usuario que el actor solicitado no fue encontrado dentro de la casa o que probablemente el nombre de dicho actor fue tecleado incorrectamente.

```
(if (eq (dbquery "select * from #temp" actors_information) 0)
then (printout t "The actor who you are looking for is not inside the house,
or probably you wrote incorrectly its/her/his name." crlf crlf)
```

Con la acción anterior, en caso de que haber encontrado datos en la tabla temporal, como se ha mencionado, dichos datos son recibidos por la estructura *deftemplate* que se indica en la función *dbquery,* logrando así que los datos se encuentren dentro de la memoria global de CLIPS, pero para que los datos sean desplegados en pantalla es necesaria otra regla que se encargue de dicho despliegue, es por ello que el hecho *(disp_act)* es introducido a la lista de hechos de CLIPS por la regla *actor-query*, ya que este hecho satisface uno de los elementos condicionales de la regla que se encargará del despliegue de los resultados.

Pero si en la tabla temporal se encuentran cero datos el hecho *(determine-actor-again)* es insertado en la lista de hechos. Dicho hecho provocará que la siguiente regla sea disparada.

```
(defrule ask-again-for-an-actor
?f1 <- (determine-actor-again)
=>
(retract *)
(assert (human-choice actor))
(if (not (yes-or-no-p "Do you want to try again? "))
then
(halt)))
```

La regla *ask-again-for-an-actor* se encarga de preguntar si se desea volver a teclear el nombre del actor. En caso de que la respuesta sea afirmativa todos los hechos contenidos en la memoria global de CLIPS son removidos y el hecho *(human-choice actor)* es introducido nuevamente para que el proceso de consulta acerca de un actor se inicie otra vez.

- **Quinta Transición.**

Una vez que los datos resultantes de la consulta a la base de datos *house* se encuentran dentro de la memoria global de CLIPS, lo único que queda por hacer es desplegar dichos datos en pantalla para que el usuario pueda ver la información que deseaba conocer. El despliegue de los datos contenidos en la estructura *deftemplate*, que para el caso de un actor se llama *actors_information*, se lleva a cabo por medio de la siguiente regla.

```
(defrule display_actor_query_result
(disp_act)
(parametro5 ?param5)
(actors_information (xa ?cx) (ya ?cy) (za ?cz) (rname ?room)
(carry ?obj) (wear ?cloth) (action ?actn))
=>
(printout t crlf "The:" ?param5 crlf "Is:" ?actn crlf "in the: " ?room crlf
?param5 "is wearing:" ?cloth crlf "and is carrying:" ?obj crlf
" The " ?room " location is: " crlf "Coordinate X: " ?cx crlf "Coordinate Y: " ?cy crlf
"Coordinate Z: " ?cz crlf)
(assert (determine-ask-again)))
```

La regla *display_actor_query_result* cuenta con tres elementos condicionales para poder ser disparada. Al llegar a la quinta transición se encuentran en la memoria global de CLIPS tanto el hecho *(disp_act)* como el hecho *(parámetro5 ?param5)*, así como también se encuentra la estructura *deftemplate* llamada *actors_information*, por lo tanto se satisfacen los tres elementos condicionales de la regla *display_actor_query_result*..

En el *deftemplate actors_information* fueron definidos los campos: X, Y y Z, los cuales representan las coordenadas del actor dentro de la habitación. Además se definieron los campos: *rname*, que se refiere al nombre de la habitación en donde se encuentra el actor, el campo *carry* se refiere al o los objetos que lleve en las manos el actor, el campo *wearing* indica la ropa que lleva puesta el actor y finalmente el campo *actino* es para indicar la acción que está realizando el actor. En la siguiente línea de código perteneciente a la regla *display_actor_query_result*, es posible observar la forma en que el contenido de cada uno de los campos anteriormente descritos son relacionados con una nueva variable, la cual está indicada por el signo de interrogación '?'.

```
(actors_information (xa ?cx) (ya ?cy) (za ?cz) (rname ?room)
(carry ?obj) (wear ?cloth) (action ?actn))
```

Finalmente, la regla *display_actor_query_result* despliega el contenido del *deftemplate actors_information*, mediante la siguiente instrucción.

```
(printout t crlf "The:" ?param5 crlf "Is:" ?actn crlf "in the: " ?room crlf
?param5 "is wearing:" ?cloth crlf "and is carrying:" ?obj crlf
" The " ?room " location is: " crlf "Coordinate X: " ?cx crlf "Coordinate Y: " ?cy crlf
"Coordinate Z: " ?cz crlf))
```

En donde es posible observar que lo que será desplegado en pantalla es el contenido de cada una de las variables, las cuales fueron relacionadas con los campos del *deftemplate actors_información.*

Una vez que el sistema experto *robot.clp* proporciona la información solicitada se llega al estado final del sistema de producción, ya que la información que se deseaba conocer en el estado inicial ha sido finalmente proporcionada.

Para terminar la ejecución del sistema *robot.clp*, se construyó otra regla, la cual se encarga de preguntar si se desea efectuar alguna otra consulta. Es por ello que además de desplegar la información en pantalla la regla *display_actor_query_result* inserta el hecho *(determine-ask-again)* , el cual provocará que la siguiente regla sea disparada.

```
(defrule ask-again-for-an-option
  ?f1 <- (determine-ask-again)
  =>
  (retract *)
  (assert (initial-fact))
  (assert (valid-answer actor a))
  (assert (valid-answer inanimate3 b))
  (assert (valid-answer inanimate2 c))
  (assert (valid-answer inanimate d))
  (assert (valid-answer room e))
  (if (not (yes-or-no-p "Do you want to start again? "))
    then
    (halt)))
```

Al ejecutarse la regla anterior se pregunta si se desea hacer otra consulta acerca de cualquier otro tipo de información, en caso de que la respuesta sea afirmativa son retractados absolutamente todos los hechos que se encuentren en la memoria global de CLIPS y después son reinsertados nuevamente a la memoria global o lista de hechos de CLIPS los hechos iniciales, es decir los hechos que representan al estado inicial del

problema, lo cual permitirá que el *robot.clp* se ejecute nuevamente desde el principio permitiendo así que el usuario pueda consultar algún otro tipo de información.

Si la respuesta a la pregunta anterior es negativa, entonces el sistema *robot.clp* suspenderá su ejecución mediante la función *(halt)*.

CAPÍTULO 7

IMPLEMENTACIÓN DE LA BASES DE DATOS

house.

7.1 Introducción.

Las Bases de Datos aparecieron por los años 1962-63 con objetivos bien definidos, entre los que se encuentra el de lograr la independencia entre la descripción lógica de los datos y su representación física. De los tres grandes modelos que aparecieron: el jerárquico, de redes y el relacional; el *modelo relacional* es el que mejor logra dicho objetivo, en el curso de este trabajo nos referiremos sólo a este modelo.

Definido por Codd [9], el modelo relacional está basado sobre conceptos de relaciones matemáticas. El modelo relacional ofrece una visión de los datos en forma de tablas.

El software que vigila el buen funcionamiento de una Base de Datos es el DBMS *(Data Base Management System)* las ventajas de utilizarlo son las siguientes:

- Mejorar la independencia lógica y física de los datos.
- Mejorar la integridad de los datos.
- Vigilar que la confidencialidad se cumpla.
- Asegurar que los caminos de acceso a los datos sean los mejores.
- Proveer alguna metodología para construir el esquema conceptual.

Las bases de datos han sido grandemente utilizadas con éxito en campos en donde la información es simple de utilizar y manipular. En los últimos años han aparecido nuevas necesidades de incorporar a las bases de datos estructuras más complejas o aplicaciones

más específicas. Algunos ejemplos de estas aplicaciones son: los Sistemas Expertos, la Robótica, Tratamiento de Imágenes, Medicina, Diseño Asistido por Computadora (CAD), etc. Todas estas aplicaciones necesitan de sistemas de almacenamiento de información en donde se asegure la integridad, la confidencialidad y todas las demás ventajas que provee un DBMS (*Data Base Management System*). Debido a esta razón a dichas aplicaciones les interesa establecer comunicación con sistemas de bases de datos.

7.2 Base de Datos *house.*

El diseño de la base de datos constituye una parte fundamental del trabajo desarrollado en la presente tesis. El cual, consta de tres partes:

1) Un Sistema Experto que simula a un robot que obedece órdenes del tipo: "*dime la localización de un objeto determinado*", por citar un ejemplo. El capítulo seis se encarga de la explicación del Sistema Experto.
2) Una Base de Datos en donde se almacena la información concerniente al contenido de una casa específica, cuya vista superior se muestra en las figuras 7-2 y 7-3.
3) Y una interfaz encargada de establecer comunicación entre la base de datos y el sistema experto, la cual se explica en el capítulo ocho.

La base de datos *house* está constituida por cinco entidades, las cuales se describen en la siguiente tabla. (tabla 7-1)

ENTIDAD	ATRIBUTOS
Actors	• *#id_act:* identificador de actor • *id_room:* identificador de habitación donde se encuentra el actor. • *act_name:* nombre del actor • *action:* acción que esta ejecutando el actor. • *wear:* ropa que usa. • *carry:* lo que trae en las manos. • *X:* coordenada en x. • *Y:* coordenada en y. • *Z:* coordenada en z.
Room	• *#id_room:* identificador de habitación. • *rname:* nombre de la habitación. • *location:* localización. • *Function:* función o propósito. • *Adj_rooms:* cuartos adyacentes.
Inanimate1	• *#id_obj:* identificador del objeto • *id_room:* identificador del cuarto donde se encuentra el objeto. • *obj_name:* nombre del objeto • *function:* propósito del objeto • *owner:* nombre del propietario del objeto • *X:* coordenada en x. • *Y:* coordenada en y. • *Z:* coordenada en z.
Inanimate2	• *idobj3:* identificador del contenedor del objeto tipo 3. • *objname:* nombre del objeto. • *function:* propósito del objeto. • *owner:* nombre del propietario del objeto • *X:* coordenada en x. • *Y:* coordenada en y. • *Z:* coordenada en z.
Inanimate3	• *#idobj3:* identificador del contenedor de objetos tipo 3. • *Id_room:* identificador del cuarto donde se encuentra el contenedor. • *oname:* nombre del contenedor • *X:* coordenada en x. • *Y:* coordenada en y. • *Z:* coordenada en z.

Tabla 7-1 entidades de la base de datos *house*

- La entidad *actors* (actores), se refiere al robot y a las personas que se encuentran en alguna de las habitaciones de la casa. Las tres entidades: *inanimate1*, *inanimate2* y *inanimate3* se refieren a los objetos distribuidos en la casa. Todos los objetos contenidos dentro de la casa se dividen en tres grupos de objetos como se explica a continuación:
- Objetos inanimados de tipo 1 o *inanimate1*. Este tipo de objetos son aquellos cuya característica principal es que no contienen dentro de sí a otro objeto más pequeño ni tampoco están contenidos dentro de otro objeto más grande.
- Objetos inanimados de tipo 2 o *inanimate2*. Son aquellos objetos que están contenidos dentro de otro objeto más grande. Un ejemplo de este tipo de objetos es el martillo que está contenido en una caja de herramientas.
- Objetos inanimados de tipo 3 o *inanimate3*. La principal característica de estos objetos es que son contenedores de otros objetos más pequeños.

En la figura 7-1 se muestra el diagrama Entidad-Relación de la base de datos *house*, en donde las entidades descritas en la tabla 7-1 se representan por medio de un rectángulo, mientras que las relaciones entre dichas entidades se representan con un rombo. En dicho diagrama pueden verse las relaciones que existen entre cada una de las entidades.

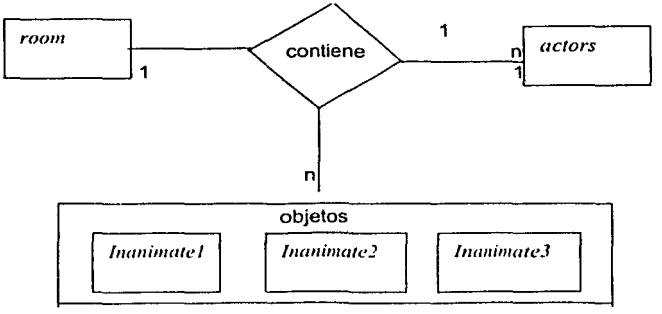

Figura 7-3. Diagrama Entidad-Relación de la base de datos *house*.

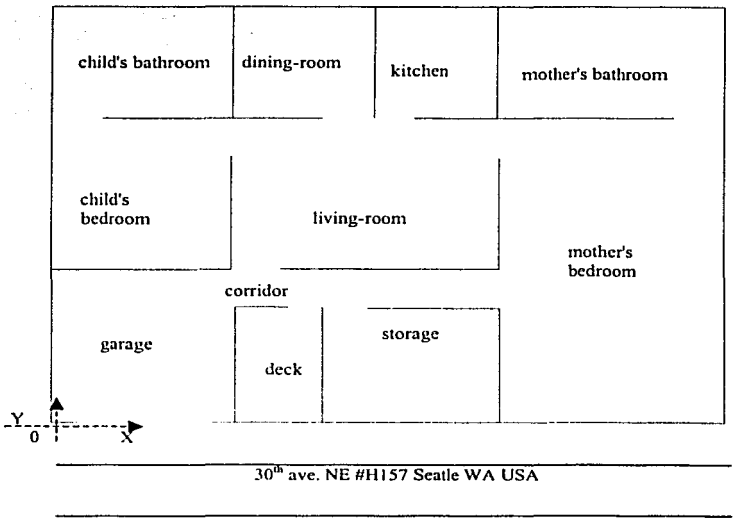

Figura 7-2. Vista superior de la casa.

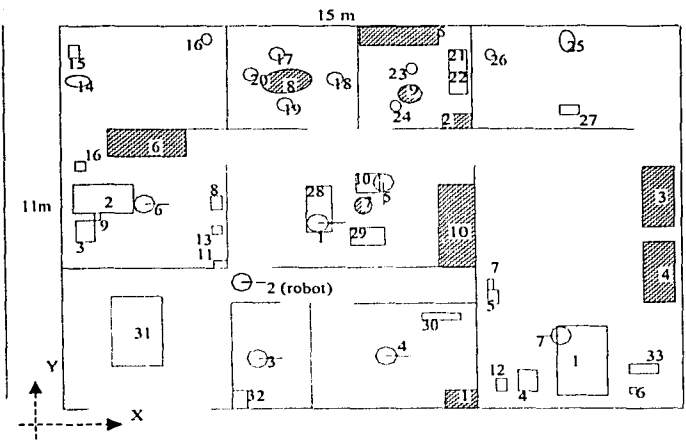

Figura 7-3 Distribución del contenido de la casa.

Las tablas constituyentes de la base de datos *house* se hicieron con base en la distribución de la casa, cuya vista superior se muestra en la figura 7-2, así como también, en la distribución de los objetos inanimados y los actores dentro de dicha casa. Para explicar esto véase la figura 7-3 en donde los objetos de tipo uno están representados por pequeños rectángulos sin sombrear y un número. Dicho número es el identificador del objeto que le es asignado en la tabla *inanimate1*. Los objetos de tipo 3 están representados por rectángulos sombreados y un número, el cual es el identificador del objeto que le es asignado en la tabla *inanimate2*. Y finalmente los actores, que sólo son siete, están representados por círculos con una pequeña línea que sobresale del círculo y un número, este número es el identificador del actor asignado en la tabla *actors*.

En la siguiente sección se presentan las cinco tablas que constituyen la base de datos *house*, así como también el diccionario de datos correspondiente a dicha base de datos.

7.2.1 Diccionario de datos y tablas de la base de datos *house*.

DICCIONARIO DE DATOS

PK= llave primaria
Objetos tipo 1= Son todos aquellos objetos que tienen la característica de no contener dentro de sí mismos a otros objetos, ni tampoco ser contenidos en otro objeto.
Objetos tipo 2= Son los objetos que están contenidos dentro de otros objetos.
Objetos tipo 3= Son contenedores de otros objetos.

NOTA:

El objetivo de la base de datos *house* es representar una casa. Debido a que la casa que se intenta representar en este trabajo físicamente se encuentra en E.U. y a que los datos relacionados con la descripción de la misma nos fueron entregados en inglés, los datos de las tablas correspondientes a la base de datos *house* están escritos en inglés. Por esta misma razón también el sistema experto *robot.clp* fue programado en inglés.

nombre	descripción	tipo de dato	*nulo*	*posibles* valores
Actino	Acción que puede estar realizando el actor en un momento determinado.	varchar (20)	no	studying, standing, playing, eating.
act_name	Nombre del actor.	varchar (12)	no	[1]mother, [2]robot, [3]agent1, [4]agent2, [5]1, [6]child, [7]father.
adj_rooms	Nombre de los cuartos adyacentes a un cuarto determinados.	varchar (80)	si	living room, dining room, mothers-bathroom, mothers-bedroom, childs-bathroom, childs-bedroom, garage, kitchen, deck, corridor, storage.
Ax	Valor de la coordenada X de un actor.	float	no	[0, 15]
Ay	Valor de la coordenada Y de un actor.	float	no	[0, 11]
Az	Valor de la coordenada Z de un actor.	float	no	[0,4]
Carry	Es el artículo que lleva el actor en un momento dado.	varchar(30)	no	book, pencil, notebook, pen, ball, sandwich.
function1	Propósito o utilidad de un objeto de tipo 1.	Varchar (20)	si	to sleep, to store, entertainment, to read, to rest, to programming, toilette, to wash, to sit, to cook, to transport, to inform.

function2	Propósito o utilidad de un objeto de tipo 2.	Varchar (20)	si	to fix, to eat, to dry, to wear, to cut, to cook, to drink, to play, to decorate.
(PK) id_act	Llave primaria o identificador de un actor determinado.	int	no	[1]=mother [2]=robot [3]=agent1 [4]=agent2 [5] = 1 [6]= child [7]=father
(PK) id_obj1	Llave primaria o identificador de un objeto de tipo 1 determinado.	int	no	[1]=mothers-bed [2]=childs-bed [3]=childs-drawer [4]=mothers-drawer [5]=mothers-tv [6]=mothers-radio [7]=mothers-videotape [8]=childs-tv [9]=childs-book [10]=sofa1 [11]=computer [12]=fathers-book [13]=childs-radio [14]=wcs-child [15]=toilets-child [16]=waterings-child [17]=chair1 [18]=chair2 [19]=chair3. [20]=chair4 [21]=stove [22]=dishwasher [23]=chair5 [24]=chair6 [25]=wcs-mother [26]=waterings-mother [27]=toilets-mother [28]=sofa2 [29]=sofa3 [30]=bicycle [31]=car [32]=chair7 [33]=newspaper
(PK)id_obj3	Llave primaria o identificador de un objeto de tipo 3 determinado.	int	no	[1]=toolbox [2]= fridge [3]=closet1 [4]=closet3 [5]=dishes-keeper [6]=closet4 [7]=table2 [8]=table1 [9]=table3 [10]=bookcase

(PK) id_room	Llave primaria o identificador de una habitación determinada.	int	no	[1]=home [2]=living-room [3]=dining-room [4]= mothers-batroom [5]=mothers-bedroom [6]=child-batroom [7]=child-bedroom [8]=garage [9]=kitchen [10]=deck [11]=corridor [12]=storage
location	Ubicación de una habitación determinada dentro de la casa. La localización de una habitación está dada tomando como referencia otra u otras habitaciones de la casa.	Varchar (70)	no	5 meters front of dining room and 1 meter front of the corridor. 4 meter to the left of the kitchen. 4 meter to the right of the kitchen. 4 meter to the right of the living room. 4 meter to the left of the dining room. 4 meter to the left of the living room. 4 meter to the left of the corridor. 4 meter to the right of the dining room. 2 meters to the right of the garage. 4 meters to the right of the garage. 2 meters to the right of the deck.

obj_name1	Nombre de un objeto de tipo 1	Varchar(20)	no	mothers-bed, childs-bed, childs-drawer, mothers-drawer, mothers-tv, mothers-radio, mothers-videotape, childs-tv, childs-book, Sofa1, computer, fathers-book, childs-radio, wcs-child, toilets-child, waterings-child, chair1, chair2, chair3, chair4, stove, dishwasher, chair5, chair6, wcs-mother, waterings-mother, toilets-mother, sofa2, sofa3, bicycle, car, chair7, newspaper
obj_name2	Nombre de un objeto de tipo 2	Varchar(20)	no	hammer, screwdriver, eggs, vegetables, meat, towel, red dress, pants, sweater, skirt, blue jeans, fathers-shoes, mothers-shoes, mothers-tennis, fathers-tennis, knife, spoons, spoons, plates, glasses, cups, towel, skates, blue jeans, sweater, childs-shoes, childs-shoes, flowers, water, lunch, coffee
obj_name3	Nombre de un objeto de tipo 3	Varchar(20)	no	Toolbox, fridge, closet1, closet3, dishes keeper, closet4, table2, table1, table3, bookcase.
owner1	Propietario de un objeto tipo 1 determinado.	Varchar(12)	si	mother, I, child, father.
owner2	Propietario de un objeto tipo 2 determinado.	Varchar(12)	si	mother, I, child, father.
rname	Nombre de la habitación	Varchar(25)	no	home, living-room, dining-room, mothers-batroom, mothers-bedroom, child-batroom, child-bedroom, garage, kitchen, deck, corridor, storage.

room_function	Propósito o función de una habitación determinada.	Varchar(30)	si	to stay, to eat, to take a shower, to rest, to study, to keep the car, to cook, to pass, to store
wear	Ropa que estan usando en un momento dado los actores.	Varchar(50)	no	red dress, red shoes, underwear. blue jeans, tennis, black shirt. purple short, t shirt, tennis. blue suit, black shoes. nothing.
x1	Valor de la coordenada X de un objeto tipo 1.	float	no	[0, 15]
x2	Valor de la coordenada X de un objeto tipo 2.	float	no	[0, 15]
x3	Valor de la coordenada X de un objeto tipo 3.	float	no	[0, 15]
y1	Valor de la coordenada Y de un objeto tipo 1	float	no	[0, 11]
y2	Valor de la coordenada Y de un objeto tipo 2.	float	no	[0, 11]
y3	Valor de la coordenada Y de un objeto tipo 3.	float	no	[0, 11]
z1	Valor de la coordenada Z de un objeto tipo 1.	float	no	[0,4]
z2	Valor de la coordenada Z de un objeto tipo 2.	float	no	[0,4]
z3	Valor de la coordenada Z de un objeto tipo 3.	float	no	[0,4]

Tablas constituyentes de la Base de Datos *House*

Tabla *room*

ROOM				
#id_room id int not null	**rname** varchar (25)	**location** varchar (70)	**room_function** varchar (30)	**adj_rooms** varchar (80)
1	home	30th ave NE #11157 Seatle WA USA	to live	
2	living room	5 meters front of dining room and 1 meter front of the corridor.	to stay	dining room, corridor, mothers-bedroom, childs-bedroom, kitchen.
3	dining room	4 meter to the left of the kitchen	to eat	kitchen, living-room, childs-bathroom
4	mothers-bathroom	4 meter to the right of the kitchen	to take a shower	kitchen, mothers-bedroom
5	mothers-bedroom	4 meter to the right of the living room	to rest, to study.	mothers-bathroom, living-room, storage, corridor
6	childs-bathroom	4 meter to the left of the dining room	to take a shower	childs-bedroom, dining room
7	childs-bedroom	4 meter to the left of the living room	to rest	childs-bathroom, living room, garage
8	garage	4 meter to the left of the corridor	to keep the car	corridor, childs-bedroom, deck.
9	kitchen	4 meter to the right of the dining room	to cook	dining-room, living-room, mothers-bathroom.
10	deck	2 meters to the right of the garage.	to stay	garage, corridor, storage.
11	corridor	4 meters to the right of the garage	to pass	garage, deck, storage, living room, mothers-bedroom.
12	storage	2 meters to the right of the deck	to store	deck, corridor, mothers-bedroom

Tabla *Actors*

ACTORS								
#id_act id int not null	id_room int not null	act_name varchar(12)	Actino varchar(20)	xa float	ya float	za float	wear varchar (50)	carry varchar (30)
1	2	mother	Studying	6.526	5.103	0	red dress, red shoes, underwear	book, pencil
2	11	robot	Standing	4.324	3.606	0		
3	10	agent1	Standing	5.131	1.425	0		
4	12	agent2	Standing	7.854	1.487	0		
5	2	I	Studying	7.831	6.465	0	blue jeans, tennis, black shirt	pen, notebook
6	7	child	Playing	2.176	5.870	0	purple short, t shirt, tennis	ball
7	5	father	Eating	12.06	2.067	0	blue suit, black shoes	sandwich

Tabla *Inanimate1*

INANIMATE1							
#id_obj1 id int not null	id_room int not null	obj_name1 varchar (20)	function1 varchar (20)	owner1 varchar (12)	x1 float	y1 float	z1 float
1	5	mothers-bed	to sleep	mother	12	2.33	0
2	7	childs-bed	to sleep	child	1.65	5.607	0
3	7	childs-drawer	to store	child	0.734	4.782	0
4	5	mothers-drawer	to store	mother	11.028	1.086	0
5	5	mothers-tv	entertainment	mother	10.554	2.980	0
6	5	mothers-radio	entertainment	mother	13.930	0.597	0
7	5	mothers-videotape	entertainment	mother	10.432	3.682	0
8	7	childs-tv	entertainment	child	3.575	5.699	0
9	7	childs-book	to read	child	0.887	5.393	0
10	2	sofa1	to rest	none	7.486	6.732	0
11	7	computer	to programming	none	3.636	4.232	0
12	7	fathers-book	to read	father	10.478	0.841	0
13	7	childs-radio	entertainment	child	3.605	4.996	0
14	6	wcs-child	nothing	child	0.359	9.250	0
15	6	toilets-child	toilette	child	0.389	10.441	0
16	6	waterings-child	to wash	child	3.506	10.592	0
17	3	chair1	to sit	none	5.261	10.169	0
18	3	chair2	to sit	none	6.635	9.456	0
19	3	chair3	to sit	none	5.442	8.525	0
20	3	chair4	to sit	none	4.607	9.577	0
21	9	stove	to cook	none	9.424	9.982	0
22	9	dishwasher	to wash	none	9.424	9.013	0
23	9	chair5	to sit	none	8.385	9.746	0
24	9	chair6	to sit	none	8.142	8.683	0
25	4	wcs-mother	Nothing	mother	12.341	10.532	0
26	4	waterings-mother	to wash	mother	10.406	9.987	0
27	4	toilets-mother	toilette	mother	12.614	8.688	0
28	2	sofa2	to rest	none	5.942	6.070	0
29	2	sofa3	to rest	none	7.857	4.870	0
30	12	bicycle	entertainment	child	8.669	2.518	0
31	8	car	to transport	father	2.339	1.224	0
32	10	chair7	to sit	none	4.443	0.510	0
33	5	newspaper	to inform	father	14.444	1.266	0

Tabla *Inanimate2*

INANIMATE2						
id_obj3 int not null	obj_name2 varchar (20)	function2 varchar (20)	owner2 varchar (12)	x2 float	y2 float	z2 float
1	hammer	to fix	father	9.313	0.437	0.21
1	screwdriver	to fix	father	9.938	0.188	0.21
2	eggs	to eat	none	9.532	8.443	0.65
2	vegetables	to eat	none	9.638	8.280	0.35
2	meat	to eat	none	9.500	8.25	0.875
3	towel	to dry	mother	14.341	6.625	1.25
3	red dress	to wear	mother	14.494	6.313	1.5
3	pants	to wear	father	14.494	5.563	1.5
3	sweater	to wear	father	14.250	6.688	1.5
3	skirt	to wear	mother	14.524	5.714	1.25
3	blue jeans	to wear	mother	14.799	6.625	0.75
4	fathers-shoes	to wear	father	14.494	4.438	0
4	mothers-shoes	to wear	mother	14.616	4.309	0
4	mothers-tennis	to wear	mother	14.372	4.450	0
4	fathers-tennis	to wear	father	14.310	3.942	0
5	knife	to cut	none	7.592	10.563	0.750
5	spoons	to eat	none	7.744	10.631	0.750
5	pan	to cook	none	8.325	10.750	0.520
5	plates	to eat	none	8.905	10.688	0.950
5	glasses	to drink	none	8.355	10.563	0.950
5	cups	to drink	none	8.416	10.813	0.950
6	towel	to dry	child	2.797	7.500	0.750
6	skates	to play	child	2.438	7.563	0
6	tennis	to wear	child	1.697	7.500	0
6	jeans	to wear	child	1.545	7.424	1.25
6	sweater	to wear	child	1.606	7.516	1.25
6	childs-shoes	to wear	child	1.422	7.313	0
6	childs-pants	to wear	child	1.270	7.563	1.25
7	flowers	to decorate	none	7.257	5.925	0.5
9	water	to drink	none	8.468	9.063	0.850
9	lunch	to eat	none	7.257	5.675	0.850
9	coffee	to drink	none	8.563	8.875	0.850

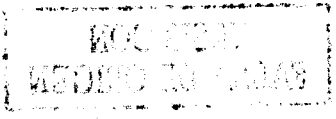

Tabla *Inanimate3*

INANIMATE3					
#id_obj3 id int not null	id_room int not null	obj_name3 varchar(20)	x3 float	y3 float	z3 float
1	12	toolbox	9.258	0.500	0
2	9	fridge	9.760	8.280	0
3	5	closet1	14.127	6.875	0
4	5	closet3	14.127	4.675	0
5	9	dishes-keeper	7.313	10.415	0
6	7	closet4	1.850	7.750	0
7	2	table2	7.393	5.653	0
8	3	table1	5.747	9.375	0
9	9	table3	8.500	8.912	0
10	2	bookcase	9.210	6.375	0

7.2.2 Ejemplos de Consultas a la Base de Datos *house.*

A fin de poder brindar una explicación del funcionamiento de la base de datos, enseguida se presentan los cinco diferentes tipos de consulta que el sistema experto hace a la base de datos. Para ello se muestran un ejemplo para cada tipo de consulta.

a) CONSULTA DE ACTORES.

El sistema experto *robot.clp* únicamente necesita que se le proporcione el nombre del actor sobre el cual se desea saber información. Por ejemplo, un tipo de pregunta que se le puede hacer al sistema experto *robot* es:

*"¿En dónde está el **niño**, qué está haciendo, cómo está vestido y qué tiene en las manos?"*

Para poder responder a la pregunta anterior el sistema experto tiene que enviar un comando SQL al servidor de base de datos Sybase. A continuación se muestra el

código del comando que es enviado desde CLIPS hasta el servidor de base de datos para poder dar respuesta a la pregunta de ejemplo que está siendo discutido. En donde la palabra en negritas, **'child'**, es el parámetro que indica el nombre del actor que se desea encontrar.

```
select r.rname, r.location, a.act_name, a.action, a.xa, a.ya, a.za, a.wear, a.carry
from room r, actors a
where a.id_room = r.id_room
and a.act_name = 'child'
```

Después de que el comando anterior es ejecutado en el servidor Sybase, el sistema experto *robot* es capaz de desplegar lo siguiente en pantalla, en donde las palabras en cursivas son los datos obtenidos de la base de datos.

The ***child*** is in the *childs-bedroom*, which location is:
4 meter to the left of the living room
The coordinates of the *child* are:
X: *2.176*
Y: *5.870*
Z: *0*

The *child* is *playing*, wearing *purple short, t-shirt, tennis*
and he/she has a *ball* in his hands.

b) CONSULTA DE OBJETOS TIPO 1 (*inanimate1*).

Igual que en el inciso anterior, el sistema experto sólo necesita saber el nombre del objeto de tipo 1 (no es contenedor de otros objetos ni está contenido en otro objeto) que se desea encontrar, en el caso de el ejemplo que se muestra enseguida el objeto es la bicicleta. Una pregunta que puede ser contestada por el sistema experto *robot* es la siguiente:

"¿Cuál es la localización de la ***bicicleta****, a quién le pertenece y para qué sirve?*

El código SQL que es enviado al servidor Sybase es el siguiente, en donde la palabra en negritas indica el nombre del objeto tipo 1 que se desea encontrar y es la

palabra que el sistema recibe como parámetro para poder ejecutar la búsqueda en la base de datos.

```
select r.rname,  r.location,  r.adj_rooms, u.obj_name1, u.function1, u.owner1, u.x1, u.y1, u.z1
from room r, inanimate1  u
where r.id_room  =  u.id_room
and u.obj_name1 ='bicycle'
```

La consulta a la base de datos hecha por medio del código anterior da como resultado los datos que se muestran en letras cursivas en el texto que se muestra enseguida, el cual es el texto desplegado por el sistema experto en pantalla :

The *bicycle* is in the *storage*, which location is:
2 meters to the right of the deck
and its adjacent room(s) is/are: *deck, corridor, mothers-bedroom*

The *bicycle* location is:
X: *8.669*
Y: *2.518*
Z: *0*
its owner: *child*
and its function is: *entertainment*

c) CONSULTA DE OBJETOS TIPO 2 (*inanimate2*).

Un objeto inanimado de tipo 2 es aquel que normalmente está contenido o guardado dentro de otro objeto, tal es el caso de los huevos, por ejemplo. Así que una pregunta que puede ser contestada por el sistema experto es la siguiente:

"¿En dónde están los **huevos**, *cuál es su localización en coordenadas x, y, z tanto del objeto que los contiene como de los huevos, para qué sirven y quién es su propietario?*

Para poder contestar la pregunta anterior, el sistema experto envía el siguiente código SQL al servidor, en donde la palabra en negritas, **'eggs'** representa el nombre del

objeto tipo 1 (huevos) que se desea localizar y es el parámetro que recibe el sistema experto para poder saber qué es lo que va a buscar.

```
select t.obj_name3, a.obj_name2, a.owner2, a.function2, a.x2, a.y2, a.z2,
r.rname, r.location, r.adj_rooms, t.x3, t.y3, t.z3
from inanimate3 t, inanimate2 a, room r
where r.id_room= t.id_room
and a.id_obj2 = t.id_obj3
and a.obj_name2 ='eggs'
```

El resultado que da la consulta anterior permite que el sistema experto despliegue es el siguiente texto en pantalla, en donde las palabras en cursivas son los datos enviados por la base de datos después de efectuar la consulta:

The *eggs* is/are in the *fridge*, which is in the *kitchen*.
The kitchen location is:
4 meter to the right of the dining room
And its adjacent rooms is/are:
dining-room, living-room, mothers-bathroom.

The coordinates of the *fridge* are:
x: *9.760*
y: *8.280*
z:*0*
The coordinates of the *eggs* are:
X: *9.532*
Y: *8.443*
Z: *0*
its function: *eating*
its owner: *none*

d) CONSULTA DE OBJETOS TIPO 3 (*inanimate3*).

Un objeto tipo 3 es el contiene dentro de sí a objetos de tipo 2. Por ejemplo un closet es un objeto de tipo 3 y un pantalón es un objeto de tipo 2. Un tipo de pregunta que puede ser contestada por el sistema experto *robot*, es la siguiente:

¿En dónde está el ***closet4*** *y cuáles son los objetos (de tipo 2) que están contenidos dentro del closet4?*

El código SQL enviado desde CLIPS hasta el servidor Sybase es el siguiente, en donde la palabra escrita en negritas es el parámetro que recibe el sistema experto antes de enviar el código SQL al servidor Sybase:

```
select  r.rname, t.obj_name3, r.location, t.x3, t.y3, t.z3
from inanimate3 t,  room r
where r.id_room = t.id_room
and t.obj_name3 ="closet4"

select  d.obj_name2
from inanimate3 t, inanimate2 d
where t.id_obj3=d.id_obj3
and t.obj_name3 = "closet4"
```

El resultado de la consulta anterior permite que el sistema experto haga el siguiente despliegue en pantalla, en donde las palabras escritas en cursivas son los datos arrojados por la base de datos después de ejecutar el código SQL mostrado anteriormente.

The ***closet4*** is in the *childs-bedroom*, which location is:
4 meters to the left of the living room.
The coordinates of the *closet4* are:
X: *1.85*
Y: *7.75*
Z: *0.0*

The next objects were found in the 'closet4':
childs-pants
childs-shoes
sweater
jeans
tennis
skates
towel

e) CONSULTA DE HABITACIONES.

Finalmente, una habitación de la casa puede contener tanto actores como objetos de tipo 1, 2 y 3. En este tipo de consulta es posible preguntarle al sistema experto acerca de la ubicación y contenido de una habitación específica. Un ejemplo de este tipo de pregunta se muestra a continuación:

*¿Cuál es la ubicación del **almacen**, cuáles son las habitaciones adyacentes a el, qué actores se encuentran en el y cuáles son los objetos de tipo 1, 2 y 3 que se encuentran en el?*

El código del comando SQL que permite contestar la pregunta anterior se muestra enseguida, en donde la palabra en negritas es el nombre de la habitación y es la palabra que el sistema experto recibe como parámetro para poder enviar el código SQL al servidor Sybase:

```
/* código SQL para la ubicación del cuarto*/
select r.rname, r.location, r.adj_rooms, r.room_function
from  room r
where r.rname = 'storage'

/* código SQL para encontrar la lista de objetos tipo 1*/
select u.obj_name1
from inanimate1 u, room r
where u.id_room = r.id_room
and r.rname = 'storage'

/* código SQL para encontrar la lista de objetos tipo 3*/
select t.obj_name3
from inanimate3 t, room r
where t.id_room = r.id_room
and r.rname = 'storage'

/* código SQL para encontrar la lista de objetos tipo 2*/
select d.obj_name2
from inanimate2 d, inanimate3 t, room r
where d.id_obj3 = t.id_obj3
and t.id_room = r.id_room
and r.rname = 'storage'

/* código SQL para encontrar la lista de actores*/
select  a.act_name
from  room r, actors a
where r.id_room = a.id_room
and r.rname = 'storage'
```

El código SQL anterior es enviado desde CLIPS hasta el servidor Sybase y permite que el sistema experto *robot* proporcione el siguiente despliegue en pantalla, en donde las palabras escritas en cursivas son los datos enviados por el servidor Sybase después de ejecutar el código SQL anterior:

The ***storage***'s location is:
2 meters to the right of the deck.
Its function is: *to store.*
And its adjacent rooms are:
deck, corridor, and mothers-bedroom

The next actor/s was/were found in the *'storage'*:
agent2

The next objects were found in the *'storage'*:
bicycle (type 1)
toolbox (type 3)
screwdriver (type 2)
hammer (type 2)

7.3 Servidor SQL Sybase.

Esta sección está dedicada a dar a conocer algunos aspectos del servidor SQL Sybase, ya que fue dicho servidor el que se utilizó para realizar tanto la interfaz CLIPS-Sybase, como la base de datos *house*.

Sybase es una base de datos tan potente como Oracle, ya que ambas manejas el lenguaje estándar para las bases de datos SQL, ambas soportan los protocolos de acceso remoto TCP/IP, ambas pueden manejar procedimientos almacenados, manejo de la misma velocidad en la búsqueda de los datos, aún cuando esto también depende en gran parte de cómo hace las consultas el programador.

A pesar de que tanto Sybase como Oracle son dos bases de datos muy potentes, se optó por utilizar Sybase por varias razones:

- Una de las razones por las cuales se optó por usar un servidor SQL Sybase para Linux en lugar de algún otro es porque es posible obtenerlo de manera gratuita en la página WEB de Sybase. Aunque ese no fue el factor decisivo para la elección del servidor SQL, si es una de las ventajas que presenta el servidor SQL Sybase para Linux.
- Sólo Sybase permite manipular sus librerías para establecer una interfaz propia de comunicación.
- La información acerca del manejo y administración de Sybase es mas difundida.

La principal razón por la que se usó Sybase para realizar la interfaz CLIPS-SYBASE, es porque permite manipular sus librerías para establece una interfaz propia de comunicación, lo cual fue el factor decisivo al elegir el servidor SQL. Las ventajas que se tiene al usar la base datos Sybase es que los desarrolladores de la Base de Datos, permiten hacer uso de las librerías que tiene la base de datos en la parte de la conectividad, de tal manera que es posible tener una forma propia de conectarse, esto quiere decir, que los programas de conexión son programados por la persona que desea tener una interfaz de comunicación y lo puede desarrollar en cualquier lenguaje de programación. Para el caso de la interfaz desarrollada en la presente tesis la programación de la misma se llevó a cabo utilizando Lenguaje "C", esto debido a que, como se menciona en el capítulo cinco, CLIPS está escrito en C así como también las funciones incluidas en *DBlibrary* de Sybase. Debido a que dichas funciones fueron muy importantes para la realización de la interfaz en la sección siguiente se habla de ellas.

7.3.1 Requerimientos Para la Instalación del Servidor Sybase

La versión del servidor Sybase que se obtuvo de la página WEB de Sybase es: *Sybase Adaptative Server Enterprise 11.0.3.3*. Para poder instalar dicho servidor fue necesario tener una máquina con 32 Mb libres en memoria y 140 Mb de espacio en disco. Para poder tener una ejecución razonable, se requiere un procesador con 64 Mb de memoria. La versión del servidor Sybase para Linux viene en tres archivos con extensión RPM. Una vez que se obtuvieron dichos archivos se procedió a instalarlos de la siguiente manera:

```
#rpm –ihv sybase-asc-11.0.3.3-6.i386.rpm
#rpm –ihv sybase-ocsd-10.0.4-6.i386.rpm
#rpm –ihv sybase-doc-11.0.3.3-6.i386.rpm
```

Finalmente, se configuró el servidor SQL Sybase basándose en los documentos instalados en el directorio: /opt/sybase/doc. Para levantar el servisor Sybase primero es necesario *logearse* con el nombre de usuario: *sybase*, el cual se crea cuando se configura el servidor, y desde el directorio: opt/sybase/install se ejecuta el siguiente comando:

```
./starserver –f ./RUNSYBASE
```

7.3.2 *DBlibrary*

DBlibrary es un conjunto de funciones escritas en C que permiten que exista comunicación entre un programa de aplicación y un servidor Sybase. Los programas escritos usando las funciones de *DBlibrary* pueden ejecutarse en máquinas *cliente*, es decir en PC's conectadas a un servidor de base de datos LINUX en una red LAN, así como también pueden ser ejecutados en el mismo servidor de bases de datos.

DBlibtrary utiliza una estructura llamada LOGINREC para poder acceder a servidores específicos. Los programas de aplicación usan una o más estructuras, llamados DBPROCESSES, para comunicarse con el servidor Sybase. Una estructura BDPROCESS es un manejador único para la base de datos y representa un conjunto de

variables ambientales que controlan la conexión a la base de datos. Cada estructura DBPROCESS opera de manera independiente y es responsable de ejecutar las siguientes actividades:

- Envío de comandos al servidor.
- Regresar los resultados enviados por el servidor.
- Envío de mensajes provistos por el servidor.

Cualquier información almacenada en los buffers de la estructura DBPROCESS puede ser accedida por medio del uso de las funciones definidas en *Dblibrary.* Los resultados devueltos por el servidor Sybase son enviados a un buffer interno de la estructura BDPROCESS como un conjunto de uno o más renglones. Dichos resultados almacenados en el buffer son leídos por el programa de *aplicación* un registro a la vez por medio de la función *dbnextrow* incluida en *DBlibrary.*

7.3.3 Funciones básicas de *DBlibrary.*

Existen alrededor de 250 funciones disponibles en *Dblibrary.* Algunas de dichas funciones son menos usadas que otras, esto depende de los requerimientos y diseño de una aplicación en específico. A continuación se describen las funciones más comúnmente utilizadas.

- **dberrhandle() y dbmsghandle().** *dberrhandle()* instala una rutina de manejo de errores, la cual se llama automáticamente si la aplicación encuentra un error. *ddmsghandle()* instala una rutina para el manejo de mensajes, la cual es llamada para proporcionar mensajes informativos enviados por el servidor SQL cuando un error ha sido encontrado.

- **dbinit ().** Inicializar *DBlibrary.* Debe ser la primera rutina de *DBlibrary* a la que debe llamar el programa de aplicación.

- **dblogin ().** Coloca una estructura (definida por LOGINREC), la cual se usa para conectar todos los DBPROCESS a Sybase. El término "coloca" se refiere a la asignación de memoria mediante un apuntador a un espacio de memoria suficientemente grande en donde los datos serán almacenados. La función *dblogin* regresa un apuntador a la estructura LOGINREC o un NULL cuando la asignación de memoria falla. La función *dblogin* provee una estructura de tipo LOGINREC, la cual es usada para establecer una conexión con el servidor Sybase. Dentro de LOGINREC existen dos componentes: la estructura DBSETLPWD() que asigna el password que se usará para conectarse al servidor y DBSETLAPP() que coloca el nombre de la aplicación, el cual aparecerá en la tabla *sysprocess* del servidor SQL. Existen algunas rutinas útiles para asignar otros aspectos de LOGINREC, sin embargo, la mayoría de las veces esas rutinas son opcionales.

- **dbopen().** Abre una conexión entre la aplicación y el servidor Sybase. Esta función utiliza la estructura LOGINREC provista por db*login()* para logearse en el servidor y y regresa una estructura DBPROCESS, la cual sirve como el conducto de intercambio de información entre la aplicación y el servidor. Después de llamar esta rutina la aplicación se conecta con el servidor SQL y entonces es capaz de enviar comando de tipo *Transact-SQL* hacia el servidor y procesar los resultados obtenidos tras la ejecución de dichos comandos.

- **dbcmd().** Esta rutina se encarga de llenar el buffer de comandos con comandos de tipo *Transact-SQL*, los cuales, una vez colocados en dicho buffer, pueden ser enviados al servidor SQL. Cada vez que se llama a *dbcmd()*, simplemente se agrega el texto proporcionado al final del texto que se encuentre en ese momento en el buffer. Esta función permite incluir comandos múltiples en el buffer.

- **dbsqlexec().** Esta función ejecuta los comandos que se encuentran en el buffer, es decir se encarga de enviar el contenido del buffer hacia el servidor SQL, el cual los lee y ejecuta.

- **dbresults().** Esta rutina obtiene los resultados del comando *Transact-SQL* que acaba de ser ejecutado. El programa necesita llamar a *dbresults()* una vez por cada comando que se encuentra en el buffer.

- **dbbind().** Se encarga de asignar las columnas resultantes a las variables del programa.

- **dbnextrow().** Se encarga de leer un renglón y colocar los resultados en las variables de programa especificadas por medio de una llamada previa a *dbbind()*. Cada vez que se llama a *dbnextrow* se lee otro renglón, hasta que el último renglón es leído y hasta que se regrese NO_MORE_ROWS. El procesamiento de los resultados toma lugar dentro del ciclo de *dbnextrow()*, ya que cada llamada a dicha función sobre-escribe los valores previos de las variables del programa.

- **dbclose().** Cierra la conexión para una estructura especifica de DBPROCESS y desaloja la memoria ocupada por dicha estructura.

- **dbexit().** Cierra la conexión con el servidor SQL Sybase y desaloja a la estructura DBPROCESS, así como también elimina cualquier estructura inicializada por *dbinit()*. Esta rutina debe ser la última en ser llamada por el programa.

CAPÍTULO 8

IMPLEMENTACIÓN DE LA INTERFAZ CLIPS-SYBASE.

(Sybfun.c)

8.1 Introducción.

El presente capítulo tiene la finalidad de brindar una descripción de la interfaz CLIPS-Sybase, la cual es la parte medular de la tesis y consiste en establecer una conexión entre un servidor SQL Sybase en donde se hizo una base de datos, la cual se describe en el capítulo siete y el sistema experto desarrollado en CLIPS *Robot.clp* descrito en los capítulos previos al capítulo dedicado a la descripción de la base de datos. Tanto el sistema experto desarrollado en CLIPS como la base de datos hecha en el servidor SQL Sybase se implementaron bajo un ambiente de sistema operativo Linux, el cual soporta los protocolos de comunicación entre máquinas de manera remota. Se explicará el desarrollo de la interfaz CLIPS-Sybase.

8.2 Objetivo.

El objetivo de la interfaz descrita en el presente capítulo es permitir que un sistema experto hecho en CLIPS sea capaz de establecer una conexión con una base de datos hecha en un servidor de base de datos SQL Sybase. Esto con la finalidad de que el sistema experto hecho en CLIPS pueda extraer información de dicha base de datos.

Para poder mostrar el funcionamiento de la interfaz se desarrolló el sistema experto *Robot.clp*, el cual se describe en el capítulo seis. Así como también se hizo la base de datos *house*, la cual se describe en el capítulo ocho. Básicamente el sistema experto *Robot.clp* solicita información y envía, por medio de la interfaz CLIPS-Sybase, esa solicitud al servidor SQL Sybase, quien se encarga de realizar la búsqueda y de enviar los

resultados obtenidos de regreso al sistema experto por medio de la interfaz descrita en el presente capítulo.

8.3 Desarrollo de la Interfaz CLIPS-Sybase.

Para llevar a cabo el desarrollo de la interfaz CLIPS-Sybase se aprovechó la ventaja de que CLIPS está diseñado para una completa integración con otros lenguajes tales como Ada y C, esto quiere decir que CLIPS puede llamarse desde un lenguaje estructurado, ejecutar su función y regresar el control al programa que lo mando llamar.

Debido a la ventaja anteriormente mencionada el desarrollo de la interfaz se llevó a cabo mediante la realización de un programa de aplicación escrito en C llamado *sybfun.c*, cuyo código se anexa en el apéndice A, ya que el código de dicho programa es estructurado, fue posible definir dentro de él funciones que son externas a CLIPS, las cuales pueden llamarse desde CLIPS y una vez que completan su tarea el control se devuelve a CLIPS. Dichas funciones externas a CLIPS son las que básicamente proveen la interfaz entre el servidor SQL Sybase y CLIPS.[24]

El programa *sybfun.c* es un programa de aplicación que hace uso de algunas funciones de librería que son provistas al instalar el servidor SQL Sybase. Dichas funciones son fundamentalmente la base sobre la cual fue hecha la interfaz CLIPS-Sybase. Dichas funciones son denominadas *DB-Library* o funciones de librería del servidor SQL y están contenidas en el directorio donde fue instalado el servidor SQL, en el caso del servidor utilizado para la realización de la interfaz la instalación se llevó a cabo en el directorio raíz llamado *opt*, donde fue creado el directorio *sybase*, dentro del cual fueron instaladas las funciones de biblioteca repartidas en dos directorios: *include* y *lib*, es

[24] *Cfr. CLIPS User's Guide.* Versión 6.1. p.1

decir, las funciones de biblioteca del servidor quedaron instaladas en la siguiente dirección.

/opt/sybase/include

/opt/sybase/lib

Para escribir el programa *sybfun.c* se siguieron los pasos que típicamente se siguen para realizar cualquier programa de aplicación que utiliza las funciones provistas por las librerías de Sybase, los cuales son mostrados a continuación:

1. Ingresar un nombre de usuario a un servidor SQL.
2. Colocar los comandos SQL-Transact dentro de un buffer y enviarlos al servidor SQL.
3. Procesar los resultados obtenidos del servidor SQL, si es que existe alguno, ejecutando un comando a la vez y un renglón del resultado a la vez. Los resultados pueden ser colocados en variables del programa, donde pueden ser manipulados por la aplicación.
4. Manejar mensajes de errores enviados tanto por las funciones de biblioteca del servidor, como por el propio servidor SQL.
5. Cerrar la conexión con el servidor SQL.[25]

8.3.1 Archivos de Cabecera

Una de las partes fundamentales de un programa de aplicación, como lo es el programa *sybfun.c*, que utiliza las funciones de biblioteca del servidor SQL es la declaración de los archivos de cabecera, en esta parte fueron incluidos al inicio de *sybfun.c* tanto el archivo *sybfront.h* como el *sybdb.h* programa, ambos son archivos de cabecera requeridos en todos los códigos fuente que contengan llamadas a rutinas contenidas dentro de las librerías de Sybase.

---

[25] *http://www.cs.sfu.ca/CourseCentral/Software/DB-LIBRARY.*

El archivo *sybfront.h* debe aparecer primero porque define constantes simbólicas, tales como valores de retorno de funciones y los valores de salida STDEXIT y ERREXIT. Además *sybfront.h* incluye typedefs para los tipos de datos que pueden ser usados en la declaración de las variables del programa.

El segundo archivo, *sybdb.h,* contiene definiciones adicionales y typedefs, la mayoría de los cuales son usados únicamente por las rutinas de las funciones de biblioteca y no deben ser utilizadas directamente por el programa. El typedef más importante contenido en *sybdb.h* es la estructura DBPROCESS, la cual debería únicamente ser accedida a través de las funciones de biblioteca, en lugar de ser accedida directamente por el programa. Las rutinas de las funciones de biblioteca se comunican con el servidor SQL a través de la estructura DBPROCESS. La mayoría de dichas rutinas requieren un DBPROCESS como primer parámetro.

El tercer archivo de cabecera, *syberror.h* contiene valores de errores severos y se incluyó en *sybfun.c* dado que el programa hace referencia a estos valores.

En adición a los archivos de cabecera *sybfront.h, sybdb.h* y *syberror.h*, también fue incluido el archivo *clips.h*, el cual es un archivo de cabecera propio de CLIPS que permite realizar la integración de CLIPS con una función externa o aplicación, como es el caso de la integración de programa *sybfun.c* con CLIPS. El archivo *Clips.h* permite que sean incluidos los prototipos de las funciones utilizadas en el programa *sybfun.c*.

Además de los archivos de cabecera, un programa de aplicación de *DB-Library* utiliza *rutinas de DB-Library*. Algunas de las rutinas y macros disponibles en *DB-Library* son listadas en la sección 7.3.3 En donde puede observar que la mayoría de ellas empiezan con el prefijo *db*. Las rutinas se mandan llamar en letras minúsculas, mientras que las macros se escriben con letras mayúsculas. Existen otras rutinas que no

empiezan con el prefijo *db*, pero éstas no son explicadas porque pertenecen a otras librerías especiales del servidor y no fueron utilizadas para la realización de la interfaz.

8.3.2 Funciones Constituyentes del Programa *sybfun.c*

Las funciones que constituyen el programa que logra hacer la interfaz, *sybopen(), sybadd(), sybcmd()* y *sybquery()*, se hicieron con base en las rutinas provistas por el servidor Sybase, es decir, con base en las funciones de librería del servidor SQL, tales como *dbopen, dbquery, dbcmd, dbclose, dbsqlexec(), dbnextrow* y *dbresults() y dblogin*

A continuación se hace una descripción detallada de cada una de las funciones que constituyen el programa *sybfun.c* y que son las que logran hacer la interfaz entre un programa hecho en CLIPS y el servidor SQL Sybase. Dichas funciones son *sybopen(), sybadd(), sybcmd()* y *sybquery()*.

- **sybopen()**

Esta función establece una conexión con el servidor de base de datos Sybase. Antes de que haga una conexión verifica si existe una conexión previa y en caso de existir la cierra para posteriormente abrir otra. Para cerrar la conexión se utilizó la rutina *dbclose()*, la cual cierra la conexión con el servidor Sybase y desaloja a la estructura DBPROCESS. También limpia cualquier estructura que haya sido iniciada por *dbinit()*.

Una vez que ya no hay ninguna conexión establecida, entonces *sybopen()* establece una nueva conexión para lo cual hace uso de las rutinas de las funciones de biblioteca *dblogin* y *dbopen*.

La rutina *dblogin()* provee una estructura llamada LOGINREC, la cual es utilizada por las funciones de biblioteca para ingresar un nombre de usuario al servidor Sybase.

Las macros que le siguen colocan ciertos componentes de LOGINREC. En este caso se utilizó la macro DTSETLPWD(), el cual asigna el password o contraseña que será usado por las funciones de biblioteca para ingresar al servidor y DBSETLUSER(), el cual asigna el login o nombre de usuario para ingresar al servidor Sybase.

Una vez que se cuenta tanto con la contraseña como con el nombre de usuario para ingresar al servidor, entonces es posible usar la rutina de *dbopen()*, la cual establece una conexión entre una aplicación, en este caso la función *sybfun.c* quien será integrada con CLIPS para lograr hacer la interfaz CLIPS-Sybase, y el servidor Sybase. *dbopen()* utiliza la estructura LOGINREC provista por *dblogin()* para ingresar al servidor y regresa una estructura DBPROCESS, la cual apunta al buffer de comandos y sirve como conducto de comunicación entre la aplicación y el servidor.

Después de que la rutina *dbopen()* es llamada y la función *sybfun.c* es integrada con CLIPS, se logra establecer la interfaz CLIPS-Sybase, permitiendo de este modo que sean enviados comandos Transact-SQL desde un programa hecho en CLIPS hasta el servidor Sybase. La comunicación entre CLIPS y el servidor SQL se lleva a cabo a través de comandos Transact-SQL. Dichos comandos son introducidos en el buffer de comandos al cual está apuntando DBPROCESS.

La sintaxis de la función que permite establecer una conexión desde un programa o sistema experto hecho en CLIPS y el servidor Sybase es la siguiente.

Sintaxis:

(dbopen usuario contraseña BD_nombre)

en donde *usuario* representa el nombre de usuario o login de una cuenta en el servidor Sybase, *contraseña* debe ser la clave o password que permite ingresar al

servidor y *BD_nombre* representa el nombre de la base de datos que será utilizada para el sistema experto hecho en CLIPS.

A continuación se muestra un ejemplo en donde se establece una conexión entre CLIPS y la base de datos *house*, la cual se define en el capítulo 8.

```
CLIPS> (dbopen tesis tesis123 house)
TRUE
```

- **sybclose()**

La función *dbclose()* es la encargada de cerrar cualquier conexión antes de establecer una.

Sintaxis:

```
(dbclose)
```

- **sybadd()**

La función *sybadd* agrega los comandos Transact-SQL al buffer de comandos, pero no los ejecuta todavía.

Esta función es útil cuando desde CLIPS se desea construir el código de una consulta en código Transact-SQL, ya que permite que dicho código sea distribuido en múltiples líneas, mejorando de esta forma la legibilidad del código fuente del programa. La sintaxis de esta función es mostrada a continuación.

Sintaxis:

```
(dbadd "segmento de código SQL)
```

- ***sybcmd()***

La función *sybcmd()*, permite que se ejecute un comando Transact-SQL, regresando el número de renglones encontrados o un –1 en caso de que un error ocurra.

Para lograr su cometido, la función *sybcmd()*, hace uso de cuatro de las rutinas de las funciones de biblioteca del servidor SQL, dichas rutinas son *dbcmd()*, *dbsqlexec()*, *dbnextrow* y *dbresults()*.

La rutina *dbcmd()* hace posible que múltiples comandos Transact-SQL sean mandados desde CLIPS hacia el buffer de comandos. Cuando el buffer de comandos se encuentra lleno, entonces la rutina *dbsqlexec()* envía el grupo de comandos hacia el servidor SQL, el cual los recibe y ejecuta.

Una vez que el grupo de comandos especificado por dbsqlexec() ha sido ejecutado en el servidor SQL, la aplicación de CLIPS debe procesar cualquier resultado que sea devuelto por el servidor.

El resultado es devuelto en renglones por medio de un procedimiento almacenado, el cual es una colección de comandos de Transact-SQL del tipo SELECT y de ejecución que permiten tener control de flujo del lenguaje.[26]

Existen dos tipos de renglones de resultado: renglones regulares, los cuales son las columnas generadas por la ejecución de comandos de tipo SELECT y los renglones computados, los cuales son las columnas generadas a partir del cómputo de una cláusula que contiene comandos de tipo SELECT.

La sintaxis de la función *dbcmd()* es la siguiente.

Sintaxis:

[26] *Transac-SQL User's Guide*, p.13-1

(dbcmd "comando SQL")

A continuación se muestra mediante un ejemplo la forma en como se puede ejecutar esta función desde la línea de comandos de CLIPS, dando por hecho que ya se estableció mediante *dbopen()* la conexión entre CLIPS y la base de datos *house,* la cual es descrita en el capítulo ocho. El comando múltiple que se muestra en el ejemplo consulta dos tablas de la base de datos *house* para poder obtener información acerca de la persona identificada como "*child*". Además se especifica que dicha información sea colocada en una tabla temporal de la base de datos llamada #temp, esto se hace con la finalidad de que posteriormente dicha tabla pueda ser consultada con un comando SQL simple mediante la función *dbquery()*.

```
CLIPS> (dbcmd " select r.rname, r.location, a.act_name, a.action, a.xa, a.ya, a.za, a.wear,
                a.carry into #temp
                from room r, actors a
                where a.id_room = r.id_room
                and a.act_name = 'child' ")

1
```

El ejemplo anterior muestra como se puede ejecutar un comando SQL desde la línea de comandos de CLIPS mediante la función *dbcmd*. En dicho ejemplo se puede observar que el código o comando SQL que será ejecutado en el servidor Sybase es lo que está entre comillas y después del nombre de la función, es decir, después de la palabra *dbcmd*. Al ejecutarse la función del ejemplo en CLIPS, es decir, después de teclear ENTER, aparecerá un número, el cual indica el número de registros encontrados después de ejecutar el comando SQL indicado entre comillas. En el caso del ejemplo aparecerá un 1, el cual indica que después de ejecutarse el comando SQL en el servidor únicamente se encontró un registro.

Es importante señalar que la función *dbcmd()* recibe todo lo que está entre comillas, es decir, el código de un comando SQL dentro del cual se incluye el parámetro

que, en este caso, determina la persona de la cual se desea saber información. Por lo tanto para construir un sistema experto que reciba una variable como parámetro, en este caso el parámetro sería *"child"*, es necesaria la construcción de una función en CLIPS que pueda recibir un parámetro y que concatene todas las partes, incluyendo los espacios y las comillas, que conforman el código del comando SQL para que posteriormente se envíe como argumento a la función *dbcmd()*.

A continuación se muestra la función construida en CLIPS que recibe un parámetro y concatena el código del comando SQL.

```
(deffunction sql_codigo
                                        (?parametro)
(str-cat select" "r.rname," "r.location," "a.action," "a.xa," "a.ya," "a.za," "a.wear," "a.carry,"
"a.act_name" "into" "#temp" "from" "room" "r," "actors" "a" "where" "a.id_room" "=" "r.id_room"
"and" "a.act_name" "=" "?parametro))
```

La función anterior puede ser mandada a llamar simplemente escribiendo su nombre y entre paréntesis el parámetro que se desee enviar como se muestra enseguida.

```
(bind x (sql_codigo child))
(dbcmd x)
```

En donde el resultado de la función *sql_codigo* es guardado en la variable "x", la cual contendrá el código del comando SQL y el valor del parámetro que se desee. La variable "x" será recibida después por la función *dbcmd()*, la cual ejecutará el comando SQL, en este caso dicho comando indica que el resultado de la consulta es almacenado en una tabla temporal de la base de datos llamada #temp, es decir los datos resultantes aún no son recibidos dentro de CLIPS.

Hasta el momento, aunque ya se explicó cómo ejecutar un comando múltiple (consulta a más de una tabla de la base de datos) mediante la función *dbcmd()*, únicamente se ha dicho que los datos resultantes de ese comando son colocados en una

tabla temporal en la base de datos, pero finalmente lo que interesa es tener dichos datos resultantes de la consulta a la base de datos *house* disponibles para que el sistema experto *Robot.clp* sea capaz de proporcionarlos cuando le sea solicitado hacerlo. Para lograr que el sistema experto pueda ver no sólo el número de registros encontrados después de una consulta a la base de datos sino que también pueda ver los datos resultantes para poder procesarlos se hizo otra función a la cual se le llamó *dbquery()*, la cual es explicada a continuación.

- **sybquery()**

La función *sybquery()*, ejecuta un solo comando Transact-SQL, regresa el número de renglones encontrados ó –1 en caso de la ocurrencia de un error. Esta función es una de las más importantes dentro de la interfaz, ya que no sólo ejecuta un comando SQL, sino que además prepara los datos resultantes de esa ejecución para que puedan ser procesados por CLIPS dentro de un sistema experto.

La función *sybquery()*, utiliza las mismas rutinas de librería que son utilizadas por *sybcmd()*, es decir, ejecuta básicamente la misma acción que *sybcmd()*, la diferencia entre ambas funciones es que una ejecuta un grupo de comandos SQL, mientras que la otra sólo ejecuta un comando y además *sybquery()* permite que los datos resultantes sean procesados por CLIPS.

El resultado de la ejecución de un comando es devuelto en renglones regulares. Dentro de cada conjunto de resultados de un comando, los renglones del resultado son procesados uno a la vez.

La función *sybquery()* utiliza a la rutina *dbresults()* para obtener los resultados del comando SQL actual y los prepara para que puedan ser procesados por la aplicación. *dbresults()*, es mandada a llamar una vez por cada comando que se encuentre en el buffer.

Además la función *sybquery()* utiliza también a la rutina *dbnextrow()*, la cual lee un renglón y coloca los resultados en las variables especificadas por el programa, en este caso se utilizó la variable 'x', la cual es una variable local declarada dentro de la función *sybquery()*. Cada vez que *dbnextrow()* es llamada se lee otro renglón del resultado, esto se repite hasta que el último renglón es leído y entonces se regresa un NO_MORE_ROWS. El procesamiento de los datos toma lugar dentro de la rutina *dbnextrow()*.

Todos los renglones de un resultado son insertados por la función sybquery() dentro de CLIPS como *facts* o *hechos*, permitiendo de esta manera que los datos resultantes de la ejecución de un comando SQL sean procesados por del Sistema Experto hecho en CLIPS *Robot.clp*. La sintaxis de *sybquery()* es la siguiente:

Sintaxis:

(dbquery "*comando SQL*" nombre_deftemplate)

donde *comando SQL* es el comando que será ejecutado por el servidor SQL y *nombre_deftemplate* es el nombre de una estructura de CLIPS llamada deftemplate, la cual es similar a una tabla de una base de datos. En dicha estructura se define el nombre de cada campo y el tipo de dato que contendrá. Tanto el nombre de los campos como el tipo de dato que contendrá cada uno debe ser exactamente igual a los campos de las tablas de la base de datos que se deseen utilizar.

Retomando el ejemplo anterior, en donde se ejecutaron múltiples comandos con solo una llamada a la función *dbcmd()* y se insertó el resultado en una tabla temporal de la base de datos llamada #temp. Ahora es posible utilizar la función *dbquery()*, ya que esta función no puede ejecutar comandos múltiples es necesario que primero se use la función

dbcmd() y después con *dbquery()* se puede ejecutar un comando que simplemente traiga los datos de la tabla #temp e inserte dichos datos como hechos en un deftemplate. Para el ejemplo que está en discusión, la estructura deftemplate sería definida de la siguiente forma. Nótese que los nombres de los campos definidos en dicha estructura son exactamente iguales a los de la tabla #temp. A continuación se muestra la estructura deftemplate *inf_act* después la estructura de la tabla temporal #temp, la cual se creó mediante un *into #temp* cuando se ejecuto el comando múltiple SQL que se usó para explicar la función *dbcmd()*. La tabla *#temp* contiene los datos resultantes de la consulta a la base de datos. Ello con la finalidad de hacer notar que los campos de la tabla son iguales a los campos definidos en el deftemplate.

```
(dbcmd " select r.rname, r.location, a.act_name, a.action, a.xa, a.ya, a.za, a.wear,
         a.carry into #temp
         from room r, actors a
         where a.id_room = r.id_room
         and a.act_name = 'child' ")
```

```
CLIPS> (deftemplate inf_act
            (field  act_name
            (type STRING) )
            (field  location
             (type STRING) )
            (field  rname
             (type STRING) )
            (field  action
             (type STRING) )
            (field  wear
             (type STRING) )
            (field  carry
             (type STRING) )
            (field  xa
            (type FLOAT) )
            (field  ya
            (type FLOAT) )
            (field  za
            (type FLOAT) ))
```

TABLA #temp
rname
location
act_name
carry
action
wear
x
y
z

Una vez definida la estructura deftemplate, en este caso llamada *inf_act*, se puede ejecutar la función *dbquery()* para introducir dentro de dicha estructura deftemplate *inf_act* los datos resultantes como hechos.

```
CLIPS>(dbquery "select * from #temp" inf_act )
```

8.3.3 Procesamiento de los Resultados.

En la sección anterior se explicó que para que CLIPS pudiera procesar los resultados primero se ejecutaba la función *dbcmd()* y después la función *dbquery()*. La primera de dichas funciones se encarga de mandar los datos resultantes de una consulta a una tabla temporal y finalmente *dbquery()*, hace una consulta a esa tabla temporal y envía los datos que encuentra en ella a una estructura deftemplate como se explico en la sección anterior.

A continuación se describe la forma en que se logra que los datos sean ingresados como hechos dentro de una estructura deftemplate.

Para cada renglón del resultado obtenido del servidor SQL, se construye una cadena, la cual es insertada en CLIPS en forma de un *hecho*. Esta cadena es construida en un campo dentro de un *deftemplate* en CLIPS, el cual corresponde a la columna del dato que se obtuvo del servidor y por consiguiente este campo debe llamarse exactamente igual tanto en el *deftemplate* como en la tabla de la base de datos.

Para llevar a cabo la construcción de la cadena que será ingresada dentro de CLIPS, se utilizó un buffer llamado *fact_tmp*, el cual es usado para ensamblar dicha cadena *fact_size* como *fact_len* son dos variables estáticas de tipo entero cuyos valores son mantenidos como base por el programa. La variable *fact_size* indica el tamaño actual del buffer *fact_tmp* y *fact_len* es la longitud de la cadena que se encuentra en dicho

buffer. De tal modo que (fact_size + fact_len) apunta a la terminación nula de la cadena. Se le asigna al buffer *fact_tmp* un tamaño de memoria lo suficientemente grande para una aplicación como un programa de CLIPS, dicha asignación de memoria se lleva a cabo como se muestra a continuación.

```
if ((fact_tmp = (char *)malloc((fact_size = 32768))) == NULL)
```

Si la cadena por ingresar está entre comillas ya sean simples o dobles son ignoradas, esto se logra con mediante la siguiente instrucción.

```
add_to_fact_tmp("\n  ("); /* )
```

Una vez que se ha construido la cadena dentro del buffer *fact_tmp*, entonces se inserta dicha cadena como un hecho mediante el uso de la función de CLIPS llamada *AssertString()* como se muestra a continuación.

```
AssertString(fact_tmp)
```

donde se asume que una estructura *deftemplate* ha sido creada para recibir a la cadena que fue construida en el buffer *fact_tmp* antes de que sea ejecutada la función *dbquery()*.

8.4 Integración de CLIPS con el Programa de Aplicación de las Funciones *DBlibrary sybfun.c.*

Una de las caracteristicas más importantes de CLIPS es su capacidad para integrarse con funciones externas o programas de aplicación. En esta sección se describirá como se llevó a cabo la integración de CLIPS con el programa de aplicación *sybfun.c*.

Fue necesario realizar algunas modificaciones dentro del código principal de CLIPS, es decir, dentro del archivo *main.c*, el código del cual se anexa al final de los capítulos. Lo primero que se hizo fue agregar los siguientes archivos de cabecera.

```
#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>
```

8.4.1 Declaración de las Funciones Externas.

Todas las funciones descritas en el punto 8.3.2 son funciones externas a CLIPS y como tales fueron definidas en CLIPS para que dichas funciones puedan ser accedidas por los programas escritos en CLIPS.

Para que dichas funciones externas fueran definidas en CLIPS fue necesario modificar la función *UserFunctions*. Esta función es inicializada en el archivo de CLIPS *main.c* y debe ser modificada ahí mismo.

Dentro de *UserFunctions*, debe efectuarse una llamada a la rutina *DefineFunction* para cada función externa que está siendo integrada con CLIPS.

A continuación de muestra el fragmento de código que contiene las modificaciones que fueron hechas a la función *UserFunctions* para que las funciones *sybopen, sybclose, sybcmd, sybquery* y *sybadd* fueran definidas en CLIPS.

```
VOID UserFunctions()
 {
      /* declaración de las funciones externas que efectúan la interfaz CLIPS-Sybase*/
    extern int sybopen();
    extern int sybclose();
    extern int sybcmd();
    extern int sybquery();
    extern int sybadd();

    DefineFunction("dbopen", 'b',PTIF sybopen, "sybopen");
    DefineFunction("dbclose", 'b',PTIF sybclose, "sybclose");
    DefineFunction("dbadd", 'b',PTIF sybadd, "sybadd");
    DefineFunction("dbcmd", 'i',PTIF sybcmd, "sybcmd");
    DefineFunction("dbquery", 'i',PTIF sybquery, "sybquery");
 }
```

El primer argumento de *DefineFunction* es el nombre de la función de CLIPS, es decir, es una cadena de representación que será utilizado para llamar a la función desde un programa hecho en CLIPS. El segundo argumento es el tipo del valor que será regresado a CLIPS, en este caso se utilizaron dos tipos: "b" indica que se trata de un valor booleano, mientras que "i" indica un valor entero. El tercer argumento indica es un

apuntado a la función y el cuarto argumento es el nombre con el que la función fue declarada en el programa de aplicación.[27]

Además de las modificaciones a la función *UserFunctions* fue necesario declarar las funciones que son utilizadas para el manejo de mensajes de errores del servidor SQL *err_handler* y *msg_handler*, así como también fue necesaria la declaración de la rutina *dbinit()*, la cual sirve para inicializar a las funciones de biblioteca del servidor SQL Sybase. A continuación se muestra un fragmento del archivo main.c que muestra la declaración de dichas funciones.

```
/* manejadores de errores y mensajes*/
int err_handler();
int msg_handler();


/*inicializa a las funciones de biblioteca del servidor SQL*/
dbinit();
```

8.4.2. Recompilación del Código Fuente de CLIPS.

Después de que se realizaron todos los cambios necesarios dentro del archivo *main.c* para lograr la integración de CLIPS con el programa *sybfun.c* se colocó a este último archivo en el mismo directorio del código fuente de CLIPS para que pudiera ser compilado y ligado con CLIPS y lograrse de esta forma la interfaz CLIPS-Sybase.

Para llevar a cabo la compilación se utilizó el archivo *makefile.cc*, ya que se utilizó el compilador C estándar de Linux. Dicho archivo contiene el nombre de todos los archivos que conforman el código fuente de CLIPS, así como también contiene las directivas de los archivos que son necesarios para la compilación del código completo. Por ello fue necesario agregar al inicio de *makefile.cc* las direcciones de las funciones de biblioteca del servidor SQL Sybase y dentro del conjunto de nombres de todos los

[27] *Cfr.* Giarratano, Joseph, *CLIPS Reference Manual, vol. II Advanced Programming Guide*, p 15-16

archivos que forman el código fuente de CLIPS se agregó también el nombre del programa externo *sybfun.c*, esto se hizo con la finalidad de que dicho programa externo fuera compilado y ligado junto con el resto de los programas que componen CLIPS. A continuación se muestran unas líneas de instrucciones, las cuales fueron agregadas al archivo original de *makefile.cc*, para llevar a cabo la correcta compilación y ligado de el nuevo código de CLIPS que permite el uso de las funciones que conforman la interfaz.

```
LIBS   =    -I/opt/sybase/include -lsybdb -L/opt/sybase/lib -lm
CFLAGS1 = -O1 -I/opt/sybase/include -lsybdb -L/opt/sybase/lib
CFLAGS  = -O2 -I/opt/sybase/include -lsybdb -L/opt/sybase/lib
```

Finalmente se ejecutó el siguiente comando desde el prompt de sistema operativo Linux, para realizar la compilación del código de CLIPS junto con el código del programa *sybfun.c*.

```
Makefile –f makefile.cc clips
```

La ejecución del comando anterior dará como resultado un archivo ejecutable de CLIPS llamado *clips*, el cual puede ser ejecutado de la siguiente forma.

```
./clips
        (CLIPS V6.0 05/12/93)
CLIPS>
```

Desde el prompt de CLIPS es posible ejecutar las nuevas funciones *dbopen, dbcmd, dbadd, dbquery* y *dbclose*, cuya sintaxis se muestra en el punto 8.3.2.

Las funciones mencionadas anteriormente son las que conforman la interfaz CLIPS-Sybase y pueden no sólo ser ejecutadas desde el prompt de CLIPS de manera aislada sino que es posible utilizarlas para la construcción de otros sistemas expertos que requiera efectuar consultas a una base de datos definida en el servidor SQL Sybase.

CONCLUSIONES.

En el transcurso del presente trabajo se presentó una manera posible de darle solución, mediante la interfaz CLIPS-Sybase, al problema de comunicación entre un sistema experto y un sistema de bases de datos, así como también se implementó tanto un sistema experto como una base de datos para poder probar el funcionamiento de la Interfaz CLIPS-Sybase.

CLIPS es una herramienta útil para el desarrollo de Sistemas Expertos, la cual ofrece tres diferentes paradigmas o metodologías para solucionar un determinado problema. Dichas metodologías son: Reglas de Producción, Programación Estructurada y Programación Orientada a Objetos (POO).

El sistema experto *robot* se desarrolló utilizando tanto reglas de producción como programación estructurada y por supuesto, utilizando también las nuevas funciones provistas por la Interfaz *CLIPS-Sybase*, las cuales son: dbopen(), dbclose(), dbquery() y dbcmd.

La principal aportación de esta tesis consiste en el desarrollo de la Interfaz *CLIPS-Sybase*, la cual permite que un sistema experto codificado usando CLIPS intercambie información con un servidor de Bases de Datos Sybase; es decir, la Interfaz permite que la parte del sistema experto que lleva a cabo el procesamiento de las reglas intercambie información con un sistema estándar de Base de Datos Sybase, permitiendo así, que el sistema experto pueda aprovechar gran parte de la eficiencia de las técnicas de manejo de memoria y procesamiento de consulta de las Bases de Datos.

Además al utilizar la Interfaz *CLIPS-Sybase*, la Base de Datos pasa a formar parte del sistema experto. Dado que el programa escrito en CLIPS junto con la Base de Datos conforman un solo Sistema Experto, ya que la *Base de Conocimientos* del Sistema Experto, la cual está constituida por *reglas* y *hechos*, se reparte entre el código de CLIPS y la Base de Datos de la siguiente manera: la parte de las reglas permanece en el código del programa escrito en CLIPS, mientras que los *hechos* son almacenados en las tablas de la Base de Datos en forma de datos. De esta manera, CLIPS y el servidor Sybase se unen, como se muestra en la figura 1, para formar una herramienta más poderosa útil para el futuro desarrollo de Sistemas Expertos que pueden ser aplicados en distintas áreas.

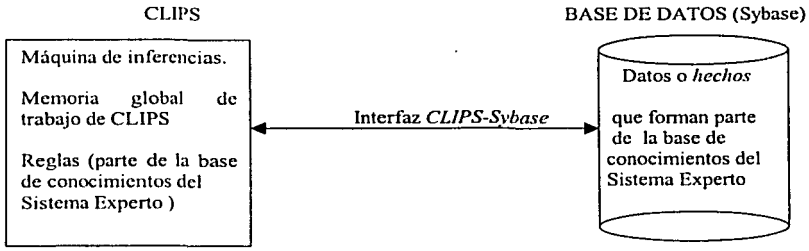

Figura 1. CLIPS y un servidor de BD, unidos para formar una herramienta más poderosa para el desarrollo de Sistemas Expertos.

Otra ventaja que brinda la Interfaz es que el código del sistema experto escrito en CLIPS no es tan extenso y complicado como es el código del mismo sistema experto hecho utilizando otro paradigma de programación como la Programación Orientada a Objetos (POO).

La manera de hacer el sistema experto *robot* presentada en este trabajo no es la única opción para hacerlo, ya que otra forma de realizar el mismo sistema experto es por medio de la POO. Sólo que al utilizar dicha metodología se tiene la desventaja de que en caso de que la *Base de Conocimientos* fuera muy grande todos los datos contenidos en ella tendrían que ser integrados en el propio código fuente del sistema experto, lo cual significaría que dicho código fueran realmente extenso, complicado de entender y difícil para poder darle mantenimiento.

Cuando se hizo la comparación en cuanto a tiempo de ejecución entre el sistema experto *robot* programado con reglas de producción, programación estructurada y usando la Interfaz *CLIPS-Sybase* y el mismo sistema experto pero usando POO, no se observó una diferencia considerable en el tiempo de ejecución. Esto es debido a que la Base de Conocimientos del sistema experto *robot*, es decir los *hechos* almacenados en forma de datos en la Base de Datos no representan una cantidad considerable de información y por lo tanto bajo estas circunstancias no es posible percatarse del alcance de los beneficios, en cuanto a la rapidez de acceso a la información almacenada en la base de datos, que trae consigo el uso de un manejador de Bases de Datos unido a el sistema experto.

El uso de la Interfaz *CLIPS-Sybase* permite que se aprovechen, por una lado los beneficios de la Base de Datos y por otro lado, los beneficios de los Sistema Expertos. Además de que debido a que el desarrollo de dicha interfaz se llevó a cabo bajo un ambiente de Sistema Operativo Linux el costo de su implementación es muy bajo.

Al utilizar la interfaz CLIPS-Sybase es posible aprovechar las caracteristicas de las reglas de los Sistemas Expertos, las cuales permiten que el sistema experto pueda

emular el razonamiento de un experto humano en algún área del conocimiento utilizando, además de las reglas de inferencia, los hechos básicos que se almacenan en la base de datos hecha en Sybase.

Se puede concluir que al tener unidos por medio de la Interfaz *CLIPS-Sybase* a un sistema experto y una base de datos, es posible aprovechar las ventajas tanto del sistema experto como de la base de datos, es decir es posible aprovechar el manejo de memoria de la base de datos lo cual proporciona rapidez en el tiempo de respuesta y además se puede aprovechar que los Sistemas Expertos, debido a que usan reglas y aplican la inferencia lógica con ellas, son capaces de responder a consultas que no pueden ser expresadas por medio de un lenguaje de consulta estándar (SQL), es decir, gracias a las reglas y a la inferencia lógica un sistema experto tiene la capacidad de deducir un nuevo hecho partiendo de hechos viejos y de las reglas contenidas en su base de conocimientos.

Finalmente, se puede concluir que la Interfaz *CLIPS-Sybase* desarrollada en la presente tesis, puede servir para el desarrollo de futuros Sistemas Expertos, los cuales pueden ser aplicados en una gran variedad de áreas o campos tanto de la Inteligencia Artificial como de otros, como la medicina por citar sólo un ejemplo.

BIBLIOGRAFÍA

[1] Rolston, David W. *Principles of Artificial Intelligence and Expert Systems Development.* Edit. McGraw-Hill. 1988.

[2] Feingenbaum, Edward A. *Knowledge Engineering in the 1980's*, Dept. of. Computer Science, Stanford University, Stanford C.A, 1982.

[3] Giarratano, Joseph. ed. al. *Expert Systems: principles and programming*

[4] Turban, Efraim, *Expert Systems and Applied Artificial Intelligence*. Edit. McMillan. 1992

[5] Siemens, Nixdorf, Dieter Nebendhal, *Sistemas Expertos,* parte 2 experiencia de la práctica.

[6] Newell, Allen and Simon, Herbert A. *Human Problem Solving.* Prentice Hall. 1972.

[7] J. A. Robinson, *A Machine- Oriented Logic Based on the Resolution Principle*, J. ACM.

[8] Korth, Henry ed. al. *Fundamentos de bases de datos.* University of Texas, Austin. Primera edición. 1987.

[9] Codd, E. F., *A Relational Model Large Shared Bank.* Comm ACM. 1970.

[10] Delobel, C, M. Adiba .*Bases de donne'es et systèmes relationnels* DUNOD. 1982

MANUALES

- Giarratano, Joseph C. *CLIPS User's Guide*
- .Giarratano, Joseph, CLIPS Reference Manual, vol. I Basic Programming Guide.
- Giarratano, Joseph, *CLIPS Reference Manual, vol. II Advanced Programming G*
- Transac-SQL User's Guide

DIRECCIONES DE INTERNET

- http://www.cs.sfu.ca/CourseCentral/Software/DB-LIBRARY. Se refiere a la descripción de las funciones de biblioteca del servidor de base de datos Sybase, las cuales fueron usadas para la realización de la interfaz CLIPS-Sybase.

- http://www.sybase.com. Se refiere a la información de los productos Sybase.

- http://www.cinefantastico.com/nexus7/ia/robots.htm. Página de Inteligencia Artificial, cuyo contenido se centra específicamente en el área de la robótica.

APÉNDICE A
CÓDIGO FUENTE DEL PROGRAMA
SYBFUN.C

```
#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>
#include "clips.h"

/* La función dbdatlen() regresa el numero de bytes de los datos contenidos      */
/* en una columna, sin embargo los datos son convertidos a cadenas. Enteros,     */
/* fechas, horas, etc. necesitaran más espacio cuando sean convertidos a         */
/* cadenas, así se declara un tamaño máximo, se debe declarar por lo menos       */
/*el tamaño máximo esperado                                                      */
#define MIN_SIZE 25


VOID *AddSymbol();

static DBPROCESS *dbproc = NULL;


/* fact_tmp apunta al inicio del buffer usado para ensamblar la cadena           */
/* e inserta el renglón que arrojo la consulta a la Base de Datos como un hecho(fact) */
static char *fact_tmp;

/* fact_size es el tamaño actual del buffer fact_tmp.                            */
/* fact_len es la longitud de la cadena en el buffer fact_tmp                    */
/*  estos valores son mantenidos  por el programa como base.                     */
/* Notese que (fact_tmp + fact_len) apunta a la terminación nula de              */
/* la cadena que esta siendo construida.                                         */
static int fact_size, fact_len;

int sybopen() {
/* establece una conexión al servidor de base de datos sybase. si una conexión   */
/* esta abierta, entonces la cierra. La sintaxis es :                            */
/* (dbopen usuario passwd dbnombre), donde usuario es el nombre del usuario,     */
/* passwd es el password, y dbnombre es el nombre de la base de datos que        */
/* se va a usar. dbopen regresa un TRUE si tiene éxito en la conexión.           */
/* o FALSE si falla                                                              */
  LOGINREC *login;

  if (ArgCountCheck("dbclose", EXACTLY, 3) == -1)
    return;
  if (dbproc != NULL)
    dbclose(dbproc);
  login = dblogin();
  DBSETLUSER(login, RtnLexeme(1));
  DBSETLPWD(login, RtnLexeme(2));
  dbproc = dbopen(login, NULL);
  dbloginfree(login);
  if (dbproc == NULL)
    return 0;
  if (dbuse(dbproc, RtnLexeme(3)) == FAIL)
```

```
  return 0;
 return dbproc != NULL;
}

int sybclose() {
/* cierra una conexión con el servidor de datos sybase, si alguna esta   */
/* abierta, siempre regresa TRUE en cualquier caso                       */
 dbclose(dbproc);
 dbproc = NULL;
 return 1;
}

sybadd() {
/* agrega comandos al buffer de comandos pero no los ejecuta todavía                 */
/* sintaxis: (dbadd "comando1" "comando2" "comando3")                                */
/* regresa TRUE cuando tiene éxito, y FALSE cuando falla                             */
/* EL USUARIO DEBE SEPARAR LOS COMANDOS CON ESPACIOS EN BLANCO                       */
 int i,j;

 if ((dbproc == NULL) || (DBDEAD(dbproc)))
  return -1;
 j = RtnArgCount();
 for (i=1; i <= j; i++)
  if (dbcmd(dbproc, RtnLexeme(i))==FAIL)
   return;
 return 1;
}

sybcmd() {
/* ejecuta un comando de SQL, regresa el nmero de renglones encontrados,          */
/* ó -1   en caso de error. La sintaxis es (dbcmd "query"), donde query es        */
/* el comando a ser ejecutado.                                                    */
 int i = 0;
 RETCODE x;

 if (ArgCountCheck("dbcmd", EXACTLY, 1) == -1)
  return -1;
 if ((dbproc == NULL) || (DBDEAD(dbproc)))
  return -1;
 dbcmd(dbproc, RtnLexeme(1));
 if (dbsqlexec(dbproc) == FAIL)
  return -1;
 while((x=dbresults(dbproc)) != NO_MORE_RESULTS) {
  if (x == FAIL)
   return -1;
  while((x=dbnextrow(dbproc)) != NO_MORE_ROWS) {
   if (x == FAIL)
        return -1;
   i++;
  }
 }
 return i;
}


sybquery() {
/* ejecuta un comando de SQL, regresa el número de renglones encontrados,        */
```

```
/* ó -1    en caso de error. Todos los renglones encontrados son insertados          */
/* como hechos(facts). La sintaxis es (dbquery "query" template_name)                 */
/* donde query es el comando a ser ejecutado y template_name es el nombre             */
/* de el deftemplate que recibirá el dato como hecho, esta función asume que un deftemplate ha sido */
/* creado para este propósito.                                                        */
  int i = 0, j,l;
  RETCODE x;
  int ctype;
  char *temp;

  /* comprueba el numero de argumentos*/
  if (ArgCountCheck("dbquery", EXACTLY, 2) == -1)
    return -1;
  if ((dbproc == NULL) || (DBDEAD(dbproc)))
    return -1;

  /* envia solo un comando SQL */
  dbcmd(dbproc, RtnLexeme(1));
  if (dbsqlexec(dbproc) == FAIL)
    return -1;


  /* obtiene cualquiera de los renglones encontrados y los ingresa como    */
  /* hechos. Comienza con un buffer lo suficientemente grande para la      */
  /*  mayoría de las aplicaciones                                          */
  if ((fact_tmp = (char *)malloc((fact_size = 32768))) == NULL) {
    fprintf(stderr, "unable to allocate memory for fact_tmp in db_query\n");
    dbexit();
    exit(1);
  }

  /* es posible tener resultados múltiples si son ejecutados comandos      */
  /* múltiples                                                             */
  while((x=dbresults(dbproc)) != NO_MORE_RESULTS) {
    if (x == FAIL)
      return -1;
    while((x=dbnextrow(dbproc)) != NO_MORE_ROWS) {
      if (x == FAIL)
          return -1;
  /* Para cada renglón se construye una cadena, la cual se hace que        */
  /* CLIPS la ingrese como un hecho. Esta cadena es construida en un       */
  /* campo, el cual corresponde a la columna del dato que se obtuvo de     */
  /* sybase, y por tanto este campo debe llamarse igual que el campo       */
  /* de dicha columna en sybase                                            */

      fact_len = 0;


/* Requerido para CLIPS 6.0 --mgk */
      add_to_fact_tmp("(");
/* mgk */

      add_to_fact_tmp(RtnLexeme(2));
      j = 0;
      while(j++ < dbnumcols(dbproc)) {
          if ((l=dbdatlen(dbproc, j)) == NULL)
            continue;  /* skip null entries.. */
```

```
        add_to_fact_tmp("\n  (");  /* ) comment for vi */
        add_to_fact_tmp(dbcolname(dbproc, j));
        ctype = dbcoltype(dbproc, j);
        /* si la cadena esta entre comillas, entonces se ignorar tanto    */
        /* las comillas dobles como las sencillas                         */
        if ((ctype == SYBCHAR) || (ctype == SYBTEXT) ||
                (ctype == SYBDATETIME) || (ctype == SYBDATETIME4)) {
          add_to_fact_tmp(" \"");
      /*Es necesario tener un buffer temporal para contar las "          */
      /* y/o las '.                                                      */
          if ((temp=(char *)malloc((l> MIN_SIZE ? l + 1 : MIN_SIZE))) == NULL) {
            fprintf(stderr, "unable to allocate temp string buffer in my_dbquery\n");
            dbexit();
            exit();
          }
          dbconvert(dbproc, ctype, dbdata(dbproc, j), dbdatlen(dbproc, j),
                          SYBCHAR, temp, -2);
          add_qs_to_fact_tmp(temp);
          free(temp);
          add_to_fact_tmp("\"");
        } else {
          add_to_fact_tmp(" ");
          fact_tmp_alloc( MIN_SIZE );
          dbconvert(dbproc, ctype, dbdata(dbproc, j), dbdatlen(dbproc, j),
                          SYBCHAR, (fact_tmp + fact_len), -2);

/* actualiza el fact_len despues dbconvert() convierte directamente a fact_tmp*/
          while(*(fact_tmp+fact_len)) fact_len++;
        }
        add_to_fact_tmp(")");
    }

/* Requerido por CLIPS 6.0 -- mgk */
    add_to_fact_tmp(")");
/* mgk */

    AssertString(fact_tmp);       /* Se asume que existe un deftemplate para */
                                  /* recibir el dato como un hecho           */
    i++;
   }
  }
  free(fact_tmp);
  return i;
}

add_to_fact_tmp(s)
/* anexa una cadena al buffer fact_temp, alojando mas espacio si es    */
/* necesario. No son ignoradas las comillas "'s o \'s                  */
char *s;
{
  int i;
  char *s1;

  i = strlen(s);
  fact_tmp_alloc(i);
  for (s1 = fact_tmp + fact_len ; *s1 = *s ; s1++, s++);
  fact_len = fact_len + i;
```

```
}

add_qs_to_fact_tmp(s)
/* anexa una cadena al buffer fact_temp alojando mas espacio si es      */
/* necesario.  Si ignora las comillas "'s y \'s con a \                 */
char *s;
{
 int i;
 char *s1;

 i = strlen(s);
 for (s1 = s; *s1; s1++)
  if ((*s1 == "") || (*s1 == '\\'))
   i++;
 fact_tmp_alloc(i);
 for (s1 = fact_tmp + fact_len ; *s1 = *s ; s1++, s++)
  if ((*s == "") || (*s == '\\')) {
   *s1++ = '\\';
   *s1 = *s;
  }
 fact_len = fact_len + i;
}

fact_tmp_alloc(i) {

/* verifica si se necesita mas memoria para el fact_tmp                 */
/* aloja mas memoria de la necesaria para evitar el realojamiento después */

 if ((i + fact_len) >= fact_size) {
  if ((fact_tmp = (char *)realloc(fact_tmp, (fact_size = 2*(fact_size+i)))) == NULL) {
   fprintf(stderr, "could not allocate memory for fact_tmp in dbquery\n");
   dbexit();
   exit(1);
  }
 }
}



/* manejador de mensajes de errores para las funciones de biblioteca del servidor SQL Sybase */

int err_handler(dbproc, severity, dberr, oserr, dberrstr, oserrstr)
DBPROCESS      *dbproc;
int            severity;
int            dberr;
int            oserr;
char           *dberrstr;
char           *oserrstr;
{
if ((dberr != SYBEPWD) && ((dbproc == NULL) || (DBDEAD(dbproc))))
  return(INT_EXIT);
else
  {
  printf("DB-Library error:\n\t%s\n", dberrstr);
  if (oserr != DBNOERR) printf("Operating-system error:\n\t%s\n", oserrstr);
  return(INT_CANCEL);
  }
```

```
}
int msg_handler(dbproc, msgno, msgstate, severity, msgtext,
        srvname, procname, line)
DBPROCESS     *dbproc;
DBINT         msgno;
int           msgstate;
int           severity;
char          *msgtext;
char          *srvname;
char          *procname;
DBUSMALLINT   line;
{

if (severity == 0) return(0);

printf ("Msg %ld, Level %d, State %d\n", msgno, severity, msgstate);

if (strlen(srvname) > 0)  printf ("Server '%s', ", srvname);
if (strlen(procname) > 0) printf ("Procedure '%s', ", procname);
if (line > 0)             printf ("Line %d", line);

printf("\n\t%s\n", msgtext);

return(0);
}
```

APÉNDICE B

CÓDIGO FUENTE DEL SISTEMA EXPERTO
ROBOT.CLP

```
/*************************************************************/
/*      UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO              */
/*                                                           */
/*          FACULTAD DE INGENIERIA                           */
/*        División de Ingeniería Eléctrica                   */
/*                                                           */
/*                                                           */
/*         INTERFAZ CLIPS-SYBASE                             */
/*        Sistema Experto Robot.clp                          */
/*                                                           */
/*  Realizado por:                                           */
/*                  Toledo Paniagua Gpe. Mireya.             */
/*                  Martínez Meza Edgar.                     */
/*                                                           */
/*************************************************************/
;;;=========================================================
;;;   SISTEMA EXPERTO ROBOT.CLP
;;;
;;;   Este sistema experto hace uso de la interfaz CLIPS-SYBASE para poder proporcionar información
;;;    concerniente al contenido de una casa, la cual esta definida en la base de datos HOUSE en el servidor
;;;   Sybase.
;;;
;;;   Al ejecutarse, el sistema despliega en pantalla el menú que permite elegir una de cinco posibles opciones
;;;   para hacer la consulta a robot.clp.
;;;
;;;   El sistema permite mostrar la manera de utilizar las funciones provistas por la Interfaz CLIPS-SYBASE.
;;;   La sintaxis de dichas funciones es:
;;;
;;;   (dbopen login passwd nombre_BD)
;;;   (dbcmd "comando SQL")
;;;   (dbquery "comando SQL" nombre_deftemplate)
;;;
;;;   NOTACION:
;;;   actors: son las personas o robot(s) dentro de la casa,
;;;   inanimate1: objetos inanimados tipo1 (sin contenido, ni contenidos)
;;;   inanimate2: objetos inanimados tipo2 (contenidos en otro objeto)
;;;   inanimate3: objetos inanimados tipo3 (contenedores de otros objetos)
;;;   room: habitaciones de la casa.
;;;
;;;    ;;;     Para ejecutar puede escribir los siguientes comandos  desde CLIPS-SYBASE:
;;;   (load robot.clp)
;;;   (reset)
;;;   (run)
;;;   o simplemente al ejecutar CLIPS-SYBASE hagalo de la
;;;   siguiente forma:
;;;   ./clips -f loadrobot.clp
;;;=========================================================

;;;***************
;;;* DEFFUNCTIONS *
```

```
;;;**********************

(deffunction yes-or-no-p (?question)
 (bind ?x bogus)
 (while (and (neq ?x yes) (neq ?x y) (neq ?x no) (neq ?x n))
   (format t "%s(Yes or No) " ?question)
   (bind ?x (lowcase (sym-cat (read)))))
 (if (or (eq ?x yes) (eq ?x y)) then TRUE else FALSE))

/* Código SQL para la búsqueda de un actor*/
(deffunction sqlcod1
(?var1)
(str-cat select" "r.rname," "r.location," "a.action," "a.xa," "a.ya," "a.za," "a.wear," "a.carry," "a.act_name"
"into" "#tmp" "from" "room" "r," "actors" "a" "where" "a.id_room" "=" "r.id_room" "and" "a.act_name" "="
"?var1))


/* Código SQL para la búsqueda de un objeto tipo3 (contenedores)*/
(deffunction sqlcod2a
(?var2a)
(str-cat select" "r.rname," "t.obj_name3," "r.location," "t.x3," "t.y3," "t.z3"
"into" "#tmp" "from" "inanimate3" "t," "room" "r" "where" "r.id_room" "=" "t.id_room" "and" "t.obj_name3"
"=" "?var2a))


(deffunction sqlcod2b
(?var2b)
(str-cat select" "d.obj_name2" "into" "#tmp" "from" "inanimate3" "t," "inanimate2" "d" "where" "t.id_obj3"
"=" "d.id_obj3" "and" "t.obj_name3" "=" "?var2b))


/* Código SQL para la búsqueda de un objeto tipo2 (contenidos dentro de otro objeto)*/
(deffunction sqlcod3
(?var3)
(str-cat select" "d.obj_name2," "d.x2,"  "d.y2," " d.z2," "d.function2," "d.owner2," "t.obj_name3," "t.x3,"
"t.y3," "t.z3," "r.rname," "r.location" "into" "#tmp" "from" "inanimate2" "d," "inanimate3" "t," "room" "r"
"where" "r.id_room=t.id_room" "and" "d.id_obj3" "=" "t.id_obj3" "and" "d.obj_name2" "=" "?var3))


/* Código SQL para la búsqueda de un objeto tipo1 (ni contenedores, ni contenidos en otro)*/
(deffunction sqlcod4
(?var4)
(str-cat select " " r.rname, " " r.location," " u.obj_name1," " u.function1," "
u.owner1," " u.x1," " u.y1," " u.z1" "into" "#tmp" "from" " room" " r," " inanimate1" " u" "where" "
r.id_ room" " =" " u.id_room" "and" "u.obj_name1" "=" " ?var4))

/* Código SQL para la búsqueda de una habitacion*/
(deffunction sqlcod5a
(?var5a)
(str-cat select" "r.rname," "r.location," "r.adj_rooms," "r.room_function" "into" "#tmp" "from" "room" "r"
"where" "r.rname" "=" "?var5a))
```

```
/*lista de nombres de actores*/
(deffunction sqlcod5act
(?var5act)
(str-cat select" "a.act_name" "into" "#tmp" "from" "room" "r," "actors" "a" "where" "r.id_room" "="
"a.id_room" "and" "r.rname" "=" "?var5act))


/* lista de objetos tipo1  dentro de la habitación*/
(deffunction sqlcod5b
(?var5b)
(str-cat select" "u.obj_name1" "into" "#tmp" "from" "inanimate1" "u," "room" "r" "where" "u.id_room" "="
"r.id_room" "and" "r.rname" "=" "?var5b))

/* lista de objetos tipo3  dentro de la habitación*/
(deffunction sqlcod5c
(?var5c)
(str-cat select" "t.obj_name3" "into" "#tmp" "from" "inanimate3" "t," "room" "r" "where" "t.id_room" "="
"r.id_room" "and" "r.rname" "=" "?var5c))


/* lista de objetos tipo2  dentro de la habitación*/
(deffunction sqlcod5d
(?var5d)
(str-cat select" "d.obj_name2" "into" "#tmp" "from" "inanimate2" "d," "inanimate3" "t," "room" "r" "where"
"d.id_obj3" "=" "t.id_obj3" "and" "t.id_room" "=" "r.id_room" "and" "r.rname" "=" "?var5d))


;;;**************
;;;* TEMPLATES *
;;;**************

(deftemplate inf_act
 (field  act_name
    (type STRING) )
(field  location
    (type STRING) )
  (field  rname
    (type STRING) )
  (field  action
    (type STRING) )
  (field  wear
    (type STRING) )
  (field  carry
    (type STRING) )
  (field  xa
    (type FLOAT) )
  (field  ya
    (type FLOAT) )
  (field  za
    (type FLOAT) ))

 (deftemplate list_obj2a
  (field  obj_name3
    (type STRING) )
  (field  x3
    (type FLOAT) )
  (field  y3
```

```
    (type FLOAT) )
   (field z3
    (type FLOAT) )
   (field rname
    (type STRING) )
   (field location
    (type STRING) ))

(deftemplate list_obj2b
  (field obj_name2
    (type STRING) ))

(deftemplate loc_obj2
(field obj_name2
    (type STRING) )
(field function2
    (type STRING) )
(field owner2
    (type STRING) )
(field x2
    (type FLOAT) )
(field y2
    (type FLOAT) )
(field z2
    (type FLOAT) )
(field obj_name3
    (type STRING) )
(field x3
    (type FLOAT) )
(field y3
    (type FLOAT) )
(field z3
    (type FLOAT) )
(field rname
    (type STRING) )
(field location
    (type STRING) ))

(deftemplate obj1inf
   (field rname
    (type STRING) )
   (field location
    (type STRING) )
   (field obj_name1
    (type STRING) )
   (field function1
    (type STRING) )
   (field owner1
    (type STRING) )
   (field x1
    (type FLOAT) )
   (field y1
    (type FLOAT) )
   (field z1
    (type FLOAT) ))

(deftemplate roomsa
```

```
  (field  mame
    (type STRING) )
  (field  location
    (type STRING) )
(field  adj_rooms
    (type STRING) )
(field  room_function
    (type STRING) ) )

(deftemplate roomsact
  (field  act_name
    (type STRING) ) )

(deftemplate roomsb
  (field  obj_name1
    (type STRING) ) )

(deftemplate roomsc
  (field  obj_name3
    (type STRING) ) )

(deftemplate roomsd
  (field  obj_name2
    (type STRING) ) )

;;;*****************
;;;* INITIAL STATE *
;;;*****************

(deffacts information
  (valid-answer actor a)
  (valid-answer inanim3 b)
  (valid-answer inanim2 c)
  (valid-answer inanim1 d)
  (valid-answer room e))

;;;***************
;;;* STARTUP RULE *
;;;***************

(defrule startup
  =>
  (printout t "Lets  make a query to SYBASE!!" crlf crlf)
  (printout t "choose  the letter of the " crlf)
  (printout t "option you desire." crlf crlf)
  (printout t "For example if you want" crlf)
  (printout t "to know information about" crlf)
  (printout t "an actor just choose the letter a. And so on." crlf crlf)
  (assert (get-human-option)))
```

```
;;;*********************
;;;* HUMAN CHOICE RULES *
;;;*********************

(defrule get-human-move
  (get-human-option)
  =>
(printout t "(a) Information about an actor " crlf "(b) Inanimate objects with content" crlf "(c) Inanimate
objects inside another object " crlf "(d) Inanimate objects without content and are not inside other object)" crlf
"(e) Rooms" crlf )
  (assert (human-choice (read))))

(defrule good-human-option
  ?f1 <- (human-choice ?choice)
  (valid-answer ?answer $? =(lowcase ?choice) $?)
 ?f2 <- (get-human-option)
  =>
  (retract ?f1 ?f2)
 (assert (human-choice ?answer)))

(defrule bad-human-option
  ?f1 <- (human-choice ?choice)
  (not (valid-answer ?answer $? =(lowcase ?choice) $?))
  ?f2 <- (get-human-option)
  =>
  (retract ?f1 ?f2)
  (assert (get-human-option)))


  ;;;*************************
  ;;;* QUERY1 RULES a)ACTORS*
  ;;;*************************
  (defrule ini_actor
  (human-choice actor)
  =>
  (dbopen tesis tesis123 house)
  (printout t  "Robot tell me where the next actor is: "crlf
  "Please write the actor's name  you want to be found by Robot." crlf
  "Remember you have to write it between simple quotes ' ' " crlf)
  (assert (parametro (read)))
  (assert (next_par1)))
  (defrule pasa_param1
  (next_par1)
  (parametro ?param)
  =>
  (bind ?x (sqlcod1 ?param))
  (assert (com ?x))
  (assert (next_q1)))


  (defrule query1
  (next_q1)
  (com ?x)
  (parametro ?param)
  =>
  (dbcmd ?x)
  (if (eq (dbquery "select * from #tmp" inf_act) 0)
```

```
then (printout  t "The actor who you are looking for is not inside the house,
or probably you wrote incorrectly its/her/his name." crlf crlf)
(assert (determine-actor-again))
else (assert (disp_act))))

(defrule display_actor_query_result
(disp_act)
(parametro ?param)
(inf_act (xa ?cx) (ya ?cy) (za ?cz) (location ?loc)(mame ?room)
 (carry ?obj) (wear ?cloth) (action ?actn) (act_name ?aname))
=>
(printout t crlf
"The " ?aname " is " ?actn " in the " ?room", which location is: "crlf
?loc"." crlf
"The "  ?aname"'s location is:" crlf
"X: "?cx crlf
"Y: "?cy crlf
"Z: "?cz crlf
"She/he/it is wearing: " ?cloth "." crlf
"And she/he has in her/his hands the next object(s): " crlf ?obj crlf)
(assert (determine-ask-again)))

(defrule ask-actor-again
  ?f1 <- (determine-actor-again)
  =>
  (retract *)
  (assert (human-choice actor))
  (if (not (yes-or-no-p "Do you want to try again? "))
     then
     (halt)))


;;;*************************
;;;* QUERY2 RULES b)INANIM3*
;;;*************************
(defrule ini_obj3
(human-choice inanim3)
=>
(dbopen tesis tesis123 house)
(printout t "Robot, tell me  where the next object type 3 is and which objects are inside it." crlf
"Please write the object type 3's name  you want to be found by Robot." crlf
"Remember you have to write it between simple quotes ' ' " crlf)
(assert (parametro (read)))
(assert (next_par2)))


(defrule pasa_param2
(next_par2)
(parametro ?param)
=>
(bind ?x (sqlcod2a ?param))
(bind ?y (sqlcod2b ?param))
(assert (com2 ?y))
(assert (com1 ?x))
(assert (next_a)))

(defrule query2a
```

```
(next_a)
(parametro ?param)
(com1 ?x)
=>
(dbcmd ?x)
(if (eq (dbquery "select * from #tmp" list_obj2a) 0)
then (printout  t "The object who you are looking for is not inside the house,
or probably you wrote incorrectly its name." crlf crlf)
(assert (determine-obj3-again))
else (dbcmd "drop table #tmp")
(assert (displ_inanim2a))))

(defrule display_objects_inside_another_query_result
(displ_inanim2a)
(list_obj2a (obj_name3 ?o3)(x3 ?tx)(y3 ?ty)(z3 ?tz) (mame ?m)(location ?loc))
=>
(printout t crlf "The  " ?o3  " is in the " ?m ", which location is: " crlf
?loc"." crlf
"The coordinates of the " ?o3 " are:" crlf
 "X: " ?tx crlf
"Y: " ?ty crlf
"Z: " ?tz crlf)
(assert (determine-ask-again))
(assert (next_b)))

(defrule query2b
(next_b)
(parametro ?param)
(com2 ?y)
=>
(dbcmd ?y)
(if (eq (dbquery "select * from #tmp" list_obj2b) 0)
then (printout  t "There was found nothing in the "  ?param crlf crlf)
else (dbcmd "drop table #tmp")
(assert (displ_inanim2b))
(printout t crlf "The next objects were found in the " ?param ":" crlf)))

(defrule display_content
(displ_inanim2b)
(list_obj2b (obj_name2 ?o2))
=>
(printout t ?o2 crlf)
(assert (determine-ask-again)))

(defrule ask-again-obj3
  ?f1 <- (determine-obj3-again)
  =>
  (retract *)
  (assert (human-choice inanim3))
  (if (not (yes-or-no-p "Do you want to try again? "))
    then
    (halt)))


;;;**************************
;;;* QUERY3 RULES c)INANIM2*
;;;**************************
```

```
(defrule ini_obj2
(human-choice inanim2)
=>
(dbopen tesis tesis123 house)
(printout t
"Robot, tell me where the next object type 2 is, which its function is and who its owner is." crlf
"Please write the object type 2's name  you want to be found by Robot." crlf
"Remember you have to write it between simple quotes ' ' " crlf)
(assert (parametro (read)))
(assert (next_par3)))

(defrule pasa_param3
(next_par3)
(parametro ?param)
=>
(bind ?x (sqlcod3 ?param))
(assert (com ?x))
(assert (next_q3)))

(defrule query3
(next_q3)
(com ?x)
=>
(dbcmd ?x)
(if (eq (dbquery "select * from #tmp" loc_obj2) 0)
then (printout  t "What you are looking for is not at the house ,
or probably you wrote incorrectly its name." crlf crlf)
(assert (determine-obj2-again))
else (assert (disp_obj2_inf))))

(defrule display_object2_information
(disp_obj2_inf)
(parametro ?param)
(loc_obj2 (obj_name2 ?obj2) (function2 ?fun2) (owner2 ?ow) (x2 ?xd) (y2 ?yd) (z2 ?zd) (obj_name3 ?obj3)
(x3 ?xt) (y3 ?yt) (z3 ?zt) (mame ?m) (location ?loc))
=>
(printout t crlf "The "?obj2  " is/are in the " ?obj3 ", which is in the " ?rn"." crlf
"The " ?m  " location is: " ?loc "." crlf crlf
"The coordinates of the " ?obj3 " are: " crlf
"X: " ?xt crlf "Y: "?yt crlf "Z: "?zt crlf crlf
"The coordinates of the " ?obj2 " are: " crlf
"X: " ?xd crlf "Y: " ?yd crlf "Z: " ?zd crlf
"its function is: " ?fun2 crlf
"and its owner is: "?ow crlf)
(assert (determine-ask-again)))

(defrule ask-again-obj2
  ?f1 <- (determine-obj2-again)
  =>
  (retract *)
  (assert (human-choice inanim2))
  (if (not (yes-or-no-p "Do you want to try again? "))
    then (halt)))
;;;**************************
;;;* QUERY4 RULES d)INANIM1 *
;;;**************************
(defrule ini_obj1
```

```
(human-choice inanim1)
=>
(dbopen tesis tesis123 house)
(printout t
"Robot, tell me where the next object type 1 is, which its function is and who its owner is." crlf
"Please write the object type 1's name  you want to be found by Robot." crlf
"Remember you have to write it between simple quotes ' '" crlf)
(assert (parametro (read)))
(assert (next_par4)))

(defrule pasa_param4
(next_par4)
(parametro ?param)
=>
(bind ?x (sqlcod4 ?param))
(assert (com ?x))
(assert (next_q4)))

(defrule query4
(next_q4)
(com ?x)
=>
(dbcmd ?x)
(if(eq (dbquery "select * from #tmp" obj1inf) 0)
then (printout  t "What you are looking for is not at the house,
or probably you wrote incorrectly its name."  crlf)
(assert (determine-obj1name-again))
else (assert (disp_obj1_inf))))

(defrule display_obj1_query_result
(disp_obj1_inf)
(parametro ?param)
(obj1inf (rname ?rn) (location ?loc) (obj_name1 ?on) (function1 ?fun) (owner1 ?ow) (x1 ?xu) (y1 ?yu) (z1
?zu))
=>
(printout t crlf crlf
"The " ?on " is in the " ?rn", which location is: " crlf
?loc crlf
"The coordinates of the " ?on " are:" crlf
"X: " ?xu crlf "Y: " ?yu crlf "Z: " ?zu crlf
"its owner: " ?ow crlf
"and its function is: " ?fun crlf crlf)
(assert (determine-ask-again)))


(defrule ask-obj1-again
  ?f1 <- (determine-obj1name-again)
  =>
  (retract *)
 (assert (human-choice inanim1))
  (if (not (yes-or-no-p "Do you want to try again? "))
    then  (halt)))
;;;**************************
;;;* QUERY5 RULES e)ROOMS *
;;;**************************

(defrule inicio
```

```
(human-choice room)
=>
(dbopen tesis tesis123 house)
Robot, tell me  where the next object type 3 is and which objects are inside it.Please write the object type 3's
name  you want to be found by Robot.
Remember you have to write it between simple quotes ' '

(printout t
"Robot, tell me  the location in the house of the next room." crlf
"Please write the room's name you want to be found by Robot." crlf
"Remember you have to write it between simple quotes ' '"crlf)
(assert (parametro (read)))
(assert (next_par5)))

(defrule pasa_param5
(next_par5)
(parametro ?param)
=>
(bind ?x (sqlcod5a ?param))
(bind ?act (sqlcod5act ?param))
(bind ?b (sqlcod5b ?param))
(bind ?c (sqlcod5c ?param))
(bind ?d (sqlcod5d ?param))
(assert (com ?x))
(assert (comact ?act))
(assert (comb ?b))
(assert (comc ?c))
(assert (comd ?d))
(assert (next_room_a)))

(defrule query5
(next_room_a)
(com ?x)
=>
(dbcmd ?x)
(if (eq (dbquery "select * from #tmp" roomsa) 0)
then (printout  t "What you are looking for is not at the house ,
or probably you wrote incorrectly its name." crlf crlf)
(assert (determine-rname-again))
else (dbcmd "drop table #tmp")
(assert (disp_room_inf))))

(defrule display_room_query_result
(disp_room_inf)
(parametro ?param)
(roomsa (rname ?rn) (location ?loc) (adj_rooms ?ar) (room_function ?func))
=>
(printout t crlf "The " ?rn "'s location is: " crlf
 ?loc". " crlf"Its function is: " ?func "."crlf "And its adjacent rooms are: "crlf  ?ar crlf )
(assert (determine-ask-again))
(assert (next_room_act)))

(defrule query5act
(next_room_act)
(parametro ?param)
(comact ?act)
=>
```

```
(dbcmd ?act)
(if (eq (dbquery "select * from #tmp" roomsact) 0)
then (assert (next_room_b))
(dbcmd "drop table #tmp")
else (dbcmd "drop table #tmp")
(assert (displ_listact))
(printout t crlf "The next actor/s was/were found in the " ?param ":" crlf))))

(defrule display_list_actors
(declare (salience 5))
(displ_listact)
(roomsact (act_name ?an))
=>
(printout t ?an crlf)
(assert (determine-ask-again))
(assert (next_room_b)))


(defrule query5b
(next_room_b)
(parametro ?param)
(comb ?b)
=>
(dbcmd ?b)
(if (eq (dbquery "select * from #tmp" roomsb) 0)
then (printout  t ""crlf crlf)
else (dbcmd "drop table #tmp")
(assert (displ_listb))
(printout t crlf "The next objects were found in the " ?param ":" crlf))))

(defrule display_list_objectsb
(declare (salience 10))
(displ_listb)
(roomsb (obj_name1 ?o1 ))
=>
(printout t ?o1 " (type 1) " crlf)
(assert (determine-ask-again))
(assert (next_room_c)))


(defrule query5c
(next_room_c)
(parametro ?param)
(come ?c)
=>
(dbcmd ?c)
(if (eq (dbquery "select * from #tmp" roomsc) 0)
then (printout  t " "crlf crlf)
else (dbcmd "drop table #tmp")
(assert (displ_listc))))

(defrule display_list_objectsc
(declare (salience 15))
(displ_listc)
(roomsc (obj_name3 ?o3))
=>
(printout t ?o3 " (type 3) " crlf)
```

```
(assert (determine-ask-again))
(assert (next_room_d)))

(defrule query5d
(next_room_d)
(parametro ?param)
(comd ?d)
=>
(dbcmd ?d)
(if (eq (dbquery "select * from #tmp" roomsd) 0)
then (assert (determine-ask-again))
else (dbcmd "drop table #tmp")
(assert (displ_listd))))

(defrule display_list_objectsd
(displ_listd)
(roomsd (obj_name2 ?o2))
=>
(printout t ?o2  " (type 2)" crlf)
(assert (determine-ask-again)))


(defrule ask-rname-again
  ?f1 <- (determine-rname-again)
  =>
  (retract *)
  (assert (human-choice room))
  (if (not (yes-or-no-p "Do you want to try again? "))
    then
    (halt)))


;;;*******************
;;;* ASK AGAIN RULE *
;;;*******************
(defrule ask-again-for-an-option
  ?f1 <- (determine-ask-again)
  =>
  (retract *)
  (assert (initial-fact))
  (assert (valid-answer actor a))
  (assert (valid-answer inanim3 b))
  (assert (valid-answer inanim2 c))
  (assert (valid-answer inanim1 d))
  (assert (valid-answer room e))
  (if (not (yes-or-no-p "Do you want to start again? "))
    then
    (halt)))
```