Universidad Autónoma de Guadalajara

INCORPORADA A LA UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ESCUELA DE INGENIERIA EN COMPUTACION

BASES Y APLICACIONES DE LA INTELIGENCIA ARTIFICIAL Y LOS LENGUAJES DE LA QUINTA GENERACION.

TESIS PROFESIONAL

QUE PARA OBTENER EL TITULO DE

INGENIERO EN COMPUTACION

P R E S E N T A

LUIS FERNANDO MACIAS LIMON

GUADALAJARA, JAL., 1989

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

CONTENIDO

I. INTRODUCCION.

INTRODUCCION

Recientes desarrollos en nuevas tecnologías, especialmente en sistemas inteligentes basados en conocimientos (knowledge based) con capacidad de interactuar con lenguaje natural, comunicación oral y visual, además del uso y creación automática de conocimientos, por un lado presentan un reto al ser humano como el creador de la inteligencia, mientras que por el otro lado representan un gran beneficio para el desarrollo actual.

Investigaciones actuales en el area de robótica, CAD, reconocimiento de voz, visión, sistemas expertos y sistemas inteligentes, contribuyen en gran medida y confirman la noción de la creación de información por medio de computadoras.

Este estudio presenta las bases y aplicaciones de la Inteligencia Artificial, enfocada a los lenguajes de la quinta generación LISP y PROLOG. Estos lenguajes no son las únicas opciones disponibles, sino que existen varias notaciones que servirían de base para ilustrar las técnicas y aplicaciones cubiertas en este estudio.

El haber escogido LISP y PROLOG fué un tanto práctico: estos lenguajes son adecuados, populares, y tienen promesas de permanecer así por mucho tiempo. LISP puede servir como un medio genérico para expresar las ideas de lenguajes funcionales modernos y

lenguajes basados en lógica, y ya que LISP y PROLOG surgieron de modelos descriptivos, ambos soportan el diseño de programas denotacionales más diréctamente que los lenguajes de programación convencionales, como Pascal o C, que tuvieron como patrón un modelo operacional de funcionamiento.

Los lenguajes de programación más avanzados facilitarán el descubrimiento de las características más apropiadas para una programación efectiva. Entonces LISP podrá absorber estas características y continuar su gran aceptación, o bien, PROLOG podrá tener tal influencia en las ideas y productos estipulados por los esfuerzos en el proyecto de la quinta generación realizado por Japon, que los programadores cambien en dirección a PROLOG.

Este estudio esta enfocado en mostrar las características y diversas ramas de la Inteligencia Artificial, explicadas ampliamente en el capítulo II. Se trató de recopilar información relacionada con temas diversos, con el fin de mostrar brevemente el enfoque actual de las computadoras 'inteligentes', esto puede causar que algunos términos resulten un poco fantásticos, o bien, se crea que han sido exagerados, pero esto está completamente fuera del objetivo de este estudio.

Asimismo, se encontrará el lector con dificultad de entender términos, los cuales fueron traducidos directamente del inglés, quizás con no mucha acertación, pero generalmente aparece la

terminología inglesa como referencia. A la vez, se han incluído ejemplos en los cuales el idioma utilizado es el inglés, pero ya que el área de especialización es dentro de la ingeniería en computación, es de esperarse un nivel de conocimiento básico del idioma inglés por parte del lector.

Los capítulos en los que se muestran LISP y PROLOG como herramientas de inteligencia artificial, tienen la finalidad de dar al lector un conocimiento básico de los mismos. Se espera que al haber leído estos capítulos, el lector tenga la capacidad de entender programas escritos en estos lenguajes y las bases para crear programas simples en estos lenguajes. Se han incluído ejemplos en ambos, pues los ejemplos son la mejor forma de mostrar el funcionamiento de lenguajes de programación. Estos ejemplos podrán ser probados en el sistema elegido por el lector, con algunas modificaciones exigidas por su sistema.

El capítulo III habla de LISP y sus grandes cualidades para tratar listas, siendo éstas mostradas de forma gradual, pues esta podrá ser la primera experiencia del lector en lenguajes funcionales. Por el contrario, el capítulo referente a PROLOG hace un estudio más veloz, sin mostrar las características o comportamiento de sus funciones, esto, suponiendo que el lector comprendió claramente los tópicos mostrados en el capítulo referente a LISP, aunque ambos capítulos son completamente independientes.

El capítulo V es una breve comparación entre las cualidades y diferencias de LISP y PROLOG. Quizás los puntos tratados no sean los más característicos, simplemente son los que el escritor consideró importantes para ser mencionados.

Al llegar a la parte final de este estudio, se espera del lector tener un conocimiento básico en las áreas de Inteligencia Artificial, haber ampliado su percepción con respecto a los sistemas expertos y haber motivado su interés para seguir adelante adquiriendo conocimientos relacionados con esta interesante disciplina. A la vez, se espera que el lector tenga cierta preferencia por algunos de los lenguajes analizados. Ya sea LISP o PROLOG, cualquiera de los dos están encaminados a ser los lenguajes más importantes en los próximos años.

No se espera que el lector tenga conocimientos de ningún lenguaje de programación, pues esta es una disciplina diferente a la de los lenguajes convencionales, pero se espera que tenga conocimientos básicos para poder entender la terminología y formar su criterio respecto a alguno de los lenguajes mostrados.

Cualquiera que sea el interés y preferencia del lector, se espera que goze este estudio y encuentre útil la información mostrada.

CAPITULO II. HERRAMIENTAS PARA MANEJAR INTELIGENCIA ARTIFICIAL

A) INTERACTUANDO CON COMPUTADORAS 'INTELIGENTES'

La Memoria es la base del modelo humano de procesamiento de información, la cual no sólo puede ser recolectada y manipulada, sino que también puede ser almacenada por períodos cortos o largos. La información podrá ser accesada meses o aún años de haber sido adquirida inicialmente o accesada día con día. De hecho, cualquier tarea que se relacione con conocimientos presupone recolectar o accesar información.

La similaridad de la memoria humana con la memoria de computadoras esta relacionada con conceptos tales como sistemas activos o pasivos, elementos de memoria, técnicas de codificación, maneras de recuperar información y capacidad de almacenamiento. Partiendo de estas bases, existen analogías entre memoria humana y memoria de máquinas.

La memoria podrá ser vista como un conjunto de elementos independientes controlados por una unidad de procesamiento central que recupera información cuando es necesaria y a la vez almacena información en celdas de memoria especificas. Este proceso es similar a la memoria asociativa de los humanos, donde elementos especificos de memoria estan unidos con otros dentro de una red.

La información almacenada puede ser vista como **conocimiento**, un sustancioso cuerpo de información que puede soportar **inteligencia**. El desarrollo de máquinas inteligentes puede ser visto como el desarrollo de métodos de representación de conocimientos. Esta idea esta íntimamente unida con el desarrollo de **sistemas expertos** y otros sistemas de Inteligencia Artificial. Las **reglas**, representando una fracción de conocimiento son el concepto central de la ingeniería de conocimientos (knowledge engineering), existen varias formas de usar las reglas en la execución de un programa.

El desarrollo de modelos computacionales de memoria, sistemas basados en reglas o conocimientos y otros conceptos derivados de investigaciones en Inteligencia Artificial ayudan a conocer más a fondo la forma en que los humanos entendemos y procesamos la información, así como el construir mecanismos equivalentes en computadoras.

1. Lenguaje y entendimiento

El tema de entendimiento de computadoras implica una seria de preguntas, relacionadas principalmente con la semántica. ¿Cómo sabemos si una computadora realmente nos entiende?

Los seres humanos relacionamos entendimiento en términos de imágenes mentales, experiencias y comportamientos. Suponemos que si alguien puede contestar correctamente a preguntas, realmente nos entiende, aunque esto no es del todo cierto. Una persona o máquina puede dominar las reglas formales para traducir un conjunto de símbolos chinos en otro conjunto semántico, aunque no entienda lo que los símbolos denotan. La habilidad para manipular símbolos correctamente no implica entendimiento de los mismos.

Algunos expertos tomarían comportamientos como una guía más confiable. Se podrán tomar conclusiones al observar el comportamiento de personas ante varios estímulos de si existe un entendimiento real o ficticio. Existen cerebros de computadoras, equipados con cuerpos de robots que pueden manejar símbolos correctamente y comportarse en varias formas, simulando el entendimiento humano.

Los seres humanos usamos dos pasos al procesar una oración. Primero traducimos el lenguaje natural en una serie de instrucciones mentales que permiten que una pregunta sea contestada o una orden llevada a cabo, y decidimos si contestar a la pregunta o llevar a cabo la acción. El segundo paso consiste en buscar en la memoria y posiblemente construir un plan basado en los datos.

La importancia del lenguaje como un método de adquirir conocimientos se deriva de su estrecha relación con la información

disponible (conocimientos en un sistema experto). El lenguaje permite comunicación (conversación) y ayuda a formar criterios para diferenciar entendimiento en sistemas computacionales o humanos.

Por esta razón se han desarrollado programas con capacidad de diálogo con los seres humanos. Unos de los primeros y más citados es el programa **ELIZA**, escrito por Joseph Weizenbaum[1]. En un programa, llamado **DOCTOR, ELIZA** adopta la postura de un psiquiatra. A continuación se muestra un fragmento de una conversación, estando en letras mayúsculas las palabras de la computadora.

Todos los hombres son iguales.
EN QUE SENTIDO?
Siempre nos molestan con una cosa u otra.
PUEDES PENSAR EN UN EJEMPLO ESPECIFICO?
Bien, my novio me hizo venir aquí.
TU NOVIO TE HIZO VENIR AQUI?
El dice que estoy presionada todo el tiempo.
ME DUELE OIR QUE ESTAS PRESIONADA.
Es cierto, yo soy infeliz.
CREES QUE EL VENIR AQUI TE AYUDARA A NO SER INFELIZ?
Necesito ayuda, eso es verdad.
QUE SIGNIFICARIA EL QUE YO TE AYUDARA? ...

---

[1] Simons, Geoff L, Are computers alive?, 1983

Es claro observar que existen varios puntos débiles en el sistema **ELIZA**, tal como que no es capaz de manipular oraciones compuestas, pero es realmente un gran paso hacia los programas conversacionales. [2] **ELIZA** no es aún un ejemplo de entendimiento de máquinas, pues necesitaría una base de datos más comprensiva, pero es obviamente un gran paso en esa dirección.

Asimismo, se han realizado programas que hacen que el psiquiatra platique con un modelo computarizado de un paciente. El sistema **PARRY** trata de un paciente con problemas de paranoia. Este tema fue escogido por la considerable información existente en relación con la paranoia, a la vez que puede ser utilizado para evaluar las diferencias entre las repuestas obtenidas por la computadora paranoica y un paciente equivalente.

Existen investigaciones para crear análisis por computadora de lenguajes naturales para traducción y otros propósitos. Estos sistemas pueden analizar el contenido de discursos, así como el significado de las palabras, con el fin de sugerir estructuras semánticas y corregir posibles errores.

Existen una exhaustiva investigación en el área de reconocimiento de voz. Actualmente existen sistemas reconocedores

---

[2] Actualmente **ELIZA** es disponible para microcomputadoras, en la versión creada por The Artificial Intelligence Research Group of Los Angeles.

creados para un sólo individuo, pero el enfoque está encaminado hacia sistemas capaces de reconocer cualquier individuo (speaker independent). Estas investigaciones son llevadas a cabo principalmente por compañías telefónicas y representan un reto para los alcances a corto plazo.

Es necesario recordar que los seres humanos poseemos abundante información acerca de diversos aspectos, y que esto hace que su entendimiento sea más profundo. Al mismo tiempo, es claro que el entendimiento por robots/computadoras, manifestado por diferentes formas de comportamiento, está mostrando grandes muestras de evolución.

2. Reconocimiento de patrones y percepción.

Actualmente los robots están adquiriendo el 'sentido' de la vista. Los ojos sólo ayudan a transmitir la información al cerebro, mientras que la base de la percepción es saber interpretar la información, o dicho de otra forma, reconocer patrones.

El reconocimiento de patrones en los seres humanos es una cualidad muy prodigiosa, mientras que en las computadoras aún es un tanto rudimentaria. Los robots estan desarrollando sus cualidades sensoriales y los cerebros para apoyarlas, pero aún hay un gran camino por recorrer antes de que tengan capacidad de ver, oler, oir o sentir como seres humanos u otros animales.

Los sentidos humanos detectan señales apropiadas, las cuales son convertidas a una forma entendible para las neuronas: Nuestros ojos pueden transmitir 4.3×10^6 bits de información al cerebro cada segundo. Varios investigadores han sugerido que existe un sistema de memoria periférica que actúa como un buffer sensorial para prevenir que el cerebro sea sobrecargado por tal torrente de información, lo cual ya está siendo aplicado en máquinas reconocedoras de patrones.

Los sistemas actuales son capaces de reconocer formas claras y bien definidas que ocupan posiciones pre-establecidas, tal como los reconocedores de letras (scanners), pero los problemas surgen cuando las formas son colocadas en posiciones no establecidas o tienen formas borrosas.

En tales circunstancias se ha encontrado conveniente identificar las formas en términos de 'cualidades'. Esto puede ser, ya sea por procesos secuenciales o paralelos. En un modelo de procesamiento en paralelo, llamado 'pandemonium', varias pruebas son aplicadas al mismo tiempo, cada cualidad es examinada por una decisión, la cual responde a estímulos específicos de alimentación.

Asimismo, se han desarrollado programas para reconocer figuras complejas como triángulos. Uno de los primeros programas en esta área, el programa SEE, identifica puntos de unión de líneas

(vértices), los cuales proveen información acerca de figuras geométricas complejas, el primer paso es identificar vértices que unen caras.

Es común que se usen patrones (plantillas) para reconocer caracteres alfanuméricos u otras figuras. Estos modelos se basan en el principio de que la figura alimentada coincida con la representación almacenada del patrón. Pero, se debe recalcar que estas técnicas sólo trabajarán si la forma es regular, bien definida y ocupan una posición pre-establecida.

B) SISTEMAS EXPERTOS BASADOS EN CONOCIMIENTOS.

Por medio de la Inteligencia Artificial, la computadora está pasando de ser una base de datos a una base de conocimientos. El sistema experto es un sistema capaz de pensar y refleccionar acerca de situaciones complejas. Dentro de la Inteligencia Artificial, que incluye voz, imagen, lenguaje natural, robótica y visión artificial entre otros, la investigación basada en los sistemas expertos es la más sobresaliente.

Como justificación a esto, Donald Michie, de la Universidad de Edinburgh afirma: "La clave acerca de los sistemas expertos es que no sólo proveen respuestas precisas, sino que también

justifican sus respuestas de una forma que tiene sentido al ser humano".[1]

El desarrollo de los sistemas expertos inició en la Universidad de Stanford, con el projecto DENDRAL, al cual le siguió MYCIN. La efectividad de un sistema experto es aceptable si el sistema captura no sólo conocimiento obtenido de textos, sino que también por la manera en que la mente del experto humano trabaja. Esto incluye: Intuición, reglas formales e informales y el conocimiento basado en experiencias. Todo esto es necesario para la base de conocimientos del sistema.

Por lo tanto, la efectividad del sistema es determinada por la calidad de la base de conocimientos, tal como utilidad, expandibilidad y confiabilidad entre otros. Que el sistema sea simple y que tenga mecanismos de auto-corrección y auto-aprendizaje para corregir sus propias deficiencias es deseado también.

Los sistemas expertos difieren de las computadoras convencionales en dos maneras: en lugar de ser distribuidos en un programa-aplicación y un sistema operativo, el software de los sistemas expertos consiste de una **base de conocimientos** y de un **procesador de deducciones.**

---

[1] Michi, Donald. Machine Intelligence and related topics. 1982.

La base de conocimientos contiene razonamientos y reglas especificas al problema a resolver, mientras que el procesador de deducciones evalúa las reglas. Los sistemas expertos mas avanzados generalmente contienen, además de las reglas relacionadas con el problema, una categoría de reglas mas a respetar, llamadas **super-reglas**, las cuales se encuentran separadas mutuamente. Las super-reglas determinan la estrategia por la cual el procesador de deducciones invocara a las reglas relacionadas con el problema, y al tenerlas separadas hace que el sistema tenga mas flexibilidad.

Los sistemas expertos actuales dependen en gran parte de dos métodos para encontrar una solución. La primera técnica, llamada **encadenamiento progresivo**, inicia con un estado conocido y aplica las reglas de conocimientos para encontrar una manera de encontrar la solución deseada. Por ejemplo, para encontrar la ruta a seguir para ir del punto A al el punto B, las reglas especificarían que hacer al encontrar intersecciones, mientras que las super-reglas especificarían donde comenzar y cómo evaluar las diferentes alternativas al encontrar intersecciones.

La segunda técnica, llamada **encadenamiento regresivo**, empezaría en el punto B y regresaría al punto A. La super-regla le diría al sistema que empezara desde la meta a conseguir. En este ejemplo es obvio que una combinación de ambas técnicas haría mas estrecho el espacio de busqueda para el sistema.

Para encontrar una ruta entre dos ciudades, este razonamiento basado en reglas es adecuado, pero en realidad existen muchas aplicaciones que requieren que el sistema tenga en cuenta principios básicos, como aplicaciones de ingeniería, que requieren tener consideracion de las leyes de la física o mecánica básica.

Este análisis nos hace calificar a los sistemas que únicamente pueden ejecutar procedimientos algorítmicos fuera de la categoría de los sistemas expertos. Los sistemas basados en deducciones pueden manejar incertidumbre por naturaleza, pero es recomendable usar algún método algorítmico cuando únicamente existe una solución bien definida, pues esto require menor cantidad de recursos.

1. Resolviendo problemas.

Los programas inteligentes tienen impacto en la manejo sistemático y la forma de codificación del conocimiento, además de la manera y sentido que usamos para resolver problemas. Técnicas mejoradas para formalizar el procesamiento de conocimiento son muy importantes para varias diciplinas, tal como la medicina, la economía y diversas ciencias. Para resolver problemas, las personas no solo poseen conocimientos, sino que también los manejan y utilizan en diferentes maneras:

- Clarificando la esencia del problema
- Analizando la relación entre los factores
- Desarrollando alternativas
- Creando opciones
- Decidiendo en su aceptación
- Desarrollando un plan
- Sugiriendo procedimientos para ser usados.

Ya sea inteligencia humana o de máquina, el que resuelva el problema debe ser altamente capaz de usar el conocimiento.

Una tarea puede calificar para el desarrollo de un sistema experto si hay al menos un ser humano experto en esa tarea. Pero este conocimiento no es suficiente, sino que el experto debe ser capaz de articular este conocimiento especial y explicar los métodos usados para resolver esta tarea, además de que esta tarea tenga límites bien definidos para su aplicación.

El procesamiento de conocimientos es a menudo enfocado a los requerimientos impuestos por la época en que vivimos, y las herramientas a nuestra disposición. La tecnología actual está enfocada a diagnósticos o clasificación de problemas en los que la solución depende principalmente de la posesión de una gran cantidad de conocimientos, como problemas de ingeniería o medicina.

Los sistemas expertos son únicos por el hecho de que obtienen conclusiones desde un almacén de conocimientos. Es por eso que se les llama 'basados en conocimientos' (knowledge-based). En su lógica utilizan reglas usadas por los expertos humanos cuando toman decisiones en su campo de especialización.

El conocimiento es codificado en una forma simbólica. Las conclusiones son obtenidas a través de deducciones lógicas en lugar de cálculos. Por consiguiente, los sistemas expertos ofrecen al humano una forma de lenguaje natural al hacer sus consultas. Esta cualidad es llamada **transparencia,** opuesta al enfoque de la 'caja negra' de un programa algorítmico.

Otra característica de los sistemas expertos es que no siguen un camino específico al tomar una decisión, sino que pueden escoger entre diferentes caminos al buscar una respuesta. Consideran el 'peso' de los factores, examinan suposiciones y toman decisiones apropiadas a los problemas a resolver.

Para simular razonamiento humano, una máquina inteligente debe ser fácilmente expandible y flexible, en otras palabras, debe incrementar su conocimiento por medio de nuevos conocimientos, esto es, ser 'capaz de aprender'.

2. Representando conocimientos por medio de reglas.

Una forma común de representar conocimientos por medio de reglas es usando reglas **IF-THEN** (también llamadas reglas de producción o de acción). Estas reglas especifican que si surge cierta clase de situación, cierta clase de acción puede ser tomada. En general estas reglas son usadas para capturar la clase de respuesta 'semi-lógica' a patrones familiares que caracterizan gran parte del razonamiento cotidiano del ser humano.

En general, reglas como éstas representan conocimientos acerca de algún área. La mayoría de los sistemas actuales basados en reglas (rule-based) contienen cientos de reglas, las cuales usualmente se obtienen durante semanas o meses por medio de información proporcionada por expertos. Por ejemplo, el sistema MYCIN contiene 450 reglas. El sistema R1 contiene alrededor de 800 y el sistema PROSPECTOR alrededor de 1600.

IF: (1) The current context is assigning devices to Unibus modules, and
(2) There is an unassigned dual-port disk drive, and
(3) The type of controller it requirers is known, and
(4) There are two such controllers, neither of which has any

devices assigned to it, and

(5) The number of devices that these controllers can support is known

THEN: (1) Assign the disk drive to each of the controllers, and

(2) Note that the two controllers have been associated and that each supports one device.

Reglas IF-THEN usadas por el sistema R1 para configurar el sistemas VAX de la compañia DEC.[2]

En cualquier sistema, las reglas estan relacionadas unas con otras para formar redes entre ellas, y una vez unidas, estas redes pueden representar un amplio cuerpo de información. La figura 1.1 presenta cómo las reglas son combinadas para formar una red.

---

[2] Andriole, Stephen J. Applications in Artificial Intelligence. 1985

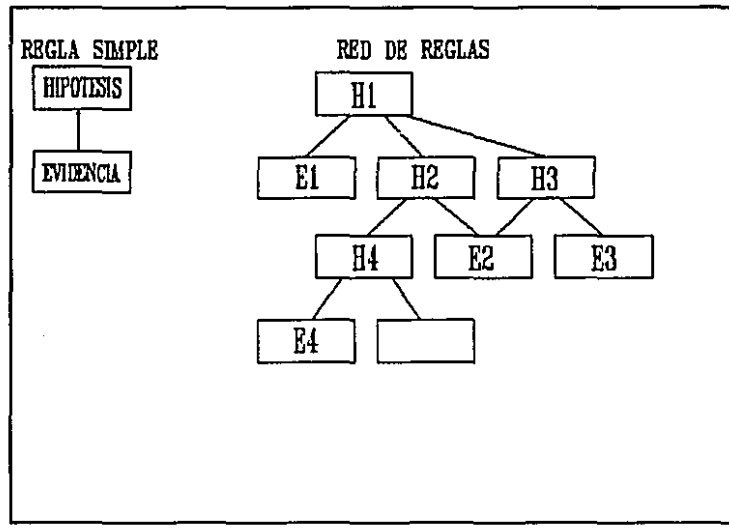

Fig 1.1. Combinación de reglas para crear redes de conocimiento.

Un experto generalmente tiene muchas reglas empiricas de acuerdo a las cuales la evidencia apoya a la conclusión o hipótesis, pero no con el cien por ciento de certidumbre. En estos casos, se utilizan valores numéricos con cada regla para indicar el grado al cual la hipótesis o conclusión apoya a la evidencia.

Las reglas en un sistema experto no son implementadas como subrutinas o como parte del programa fuente sino que las reglas

para una tarea especifica son escritas en un lenguaje especializado, el cual es alimentado al programa para producir una representación interna que hace al sistema experto un verdadero experto en esa tarea.

El programa por sí mismo es un intérprete y un mecanismo de razonamiento generalizado.

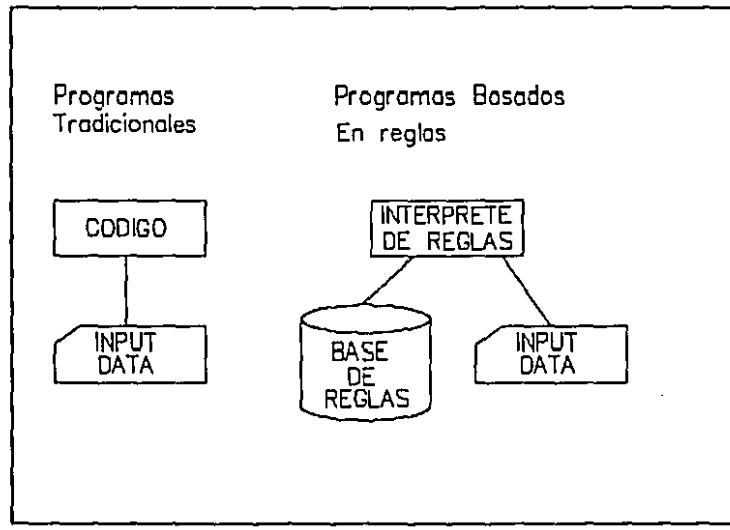

Fig 1.2. Diferencia entre programas tradicionales y programas basados en reglas.

La figura 1.2 nos hace ver una gran diferencia entre los sistemas convencionales y los sistemas basados en reglas: Un

programa basado en reglas es dividido en un programa de razonamiento general, llamado **intérprete de reglas** y un archivo de reglas obtenidas de un experto, llamado **base de reglas o base de conocimientos.** El intérprete de reglas carga la base de conocimientos y la usa para tener una consulta interactiva con el usuario.

Se debe hacer notar que no todos los sistemas expertos son basados en reglas. El uso de sistemas basados en reglas es más atractivo cuando la mayor parte del conocimiento proviene de asociaciones experimentales. Cuando información casual es disponible, otras formas de resolver el problema deberan ser usadas. Un ejemplo son las redes que unen nodos para representar relaciones casuales.

3. Características de un sistema experto.

Ahora es más claro notar la diferencia entre un sistema convencional y un sistema experto basado en reglas. Como se mencionó anteriormente, una gran diferencia es la separación del conocimiento (las reglas que forman una basa de conocimiento) del mecanismo de razonamiento general (interprete de reglas). Esta separación, en conjunto con la división de conocimiento general en reglas, ofrece varias ventajas:

- Desarrollo incremental de la base de conocimientos a través del tiempo al permitir a los expertos refinar viejas reglas, así como el añadir nuevas.
- El mismo sistema general puede ser usado para una variedad de aplicaciones al poder pasar de un set de reglas a otro con facilidad.
- El mismo conocimiento puede ser usado en diferentes maneras al cambiar el intérprete de reglas.
- El programa puede dar explicaciones simples y claras de su comportamiento al describir las reglas que estan siendo aplicadas. Esto a su vez es una manera muy efectiva para descubrir reglas erroneas.
- La posibilidad de desarrollar sistemas que pueden checar la consistencia de sus propias reglas, asi como modificar y aprender nuevas.

Otra caracteristica de la mayoría de los sistemas expertos es que tratan de 'mimificar' hasta cierto punto la forma en que los humanos tomamos las decisiones. El razonamiento de 'encuentra un patrón-dibuja una conclusión' es usado en muchos sistemas expertos asi como es usado por nosotros en los problemas de la vida cotidiana. Lo que nos distingue de los sistemas expertos es lo poco usual y lo valioso de su conjunto de reglas para encontrar una solución. Tales reglas a menudo son no conclusivas, pero son muy sugestivas para encontrar la solución.

4. Ejemplos de sistemas expertos usados en diagnósticos.

Para entender las cualidades y limitaciones de los sistemas expertos basados en conocimientos, es de gran ayuda examinar sistemas específicos creados para resolver problemas particulares.

El problema general de diagnosis es clasificar un objeto, evento o situación en la base de cierta información acerca de sus características. Las categorías pueden o no ser mutuamente exclusivas, y los datos ser obtenidos secuencialmente o en paralelo. En un sentido formal, los problemas de diagnosis pueden ser vistos como problemas en teorías de decisión estadística, cuya solución usualmente requiere la estimación de la función de probabilidades multi-variables provenientes de grandes cantidades de datos. La forma de análisis de los sistemas basados en conocimientos a tales problemas substituye efectivamente el conocimiento y juicio de seres humanos expertos para este tipo de funciones.

Varios sistemas expertos impresionantes han sido desarrollados con propósitos de diagnosis médico. El programa **INTERNIST**, desarrolado en la Universidad de Pittsburgh usa información de 4000 manifestaciones posibles para diagnosticar problemas de medicina internista que puede involucrar multiples casos de 500 diferentes

tipos de enfermedades. El programa contiene una larga taxonomía de tipos de enfermedades, asi como reglas que unen manifestaciones de estos tipos y un ingenioso procedimiento de control para reducir las clases de enfermedades que son típicas de estas manifestaciones.

En muchas pruebas, **INTERNIST** ha demostrado la capacidad de dar diagnósticos acertados a múltiples casos de enfermedades descritos en publicaciones médicas como casos de particular dificultad.

Otro sistema de diagnóstico conocido es el sistema **MYCIN**, desarrollado en la Universidad de Stanford. **MYCIN** ofrece diagnósticos a infecciones provenientes de bacterias y recomienda un antibiótico como terapia. **MYCIN** esta organizado en base al uso sistemático de una larga colección de reglas que unen los datos del paciente con hipótesis de infección. Fórmulas basadas en una teoría parecida a la probabilidad de certidumbre son usadas para acertar la naturaleza tan variada del conocimiento médico.

De varias maneras, el programa hace uso de la modularidad que las reglas proveen para expresar este conocimiento médico. Desde el punto de vista del desarrollador del sistema, la modularidad permite desarrollo del sistema a largo plazo al tener la facilidad de expansión y refinamiento de la base que contiene las reglas (base de reglas). Este programa obtiene información del usuario por el método de **encadenamiento regresivo** hacia las reglas. Esto

permite al programa tener explicaciones simples pero útiles de su razonamiento al dejar conocer las reglas que esta usando.

C) Inteligencia Artificial y programación lógica

Crear un sistema experto supone crear un sistema que tenga experiencia parecida a la humana y muestre un comportamiento inteligente en algún área de especialización. Tales sistemas a menudo ofrecen un conocimiento comprensivo acerca de una aplicación específica y tal conocimiento debe ser codificado y refinado en una fase de desarrollo prolongada.

Los lenguajes de programación actuales generalmente no proveen estas ventajas, pero varios expertos afirman que para construir un sistema experto necesitamos partir de modelos tradicionales de procesamiento de datos y buscar nuevas arquitecturas. El concepto de sistemas basados en reglas (rule-based) ha sido un gran paso en esta dirección, estos ofrecen la posibilidad de codificar conocimientos en una forma clara que puede ser ejecutada con facilidad.

La **programación lógica** puede ser vista como un tipo de programación basada en reglas, en ella, la semántica de las reglas es clara y la modularidad de las reglas garantizada, es por eso que las reglas son escritas en una forma lógica.

1. Del procesamiento de datos a sistemas basados en reglas

De acuerdo a un punto de vista tradicional, un programa de computadora consiste en un set de instrucciones, expresadas en un lenguaje de programación especial. La máquina usa estas instrucciones en un conjunto de datos para obtener un resultado esperado. Supongamos un sistema imaginario que sea 'consejero de impuestos' para nuestro análisis. Cada vez que el programa es ejecutado, presenta datos especificando las circunstancias financieras de una persona en particular, y presenta sugerencias acerca de cómo esa persona puede ahorrar en sus pagos de impuestos. Entre otras cosas, el programa debería indicar si sería mejor tener los impuestos separados en una pareja, o si la persona tiene mejores opciones de pagos, etc.

Construir tal programa sería una tarea muy sencilla y obviamente representaría gran utilidad. Sin embargo, que pasaría si cambiaran las leyes relacionadas con impuestos? El problema es que cualquier cambio pequeño podría afectar muchas secciones del programa. Generalmente sería muy difícil seguir las partes del programa que dependen directa o indirectamente de los cambios de dichas leyes. Por supuesto que esto se reduce sustancialmente al escribir un programa claro y bien estructurado, pero sería muy difícil pronosticar los cambios que serán requeridos en el futuro.

El problema más claro con este supuesto programa es que al ser escrito en lenguajes convencionales, lo que el programa sabe es mezclado con **cómo** el programa realizará estas tareas, así, sería muy difícil para un no-programador (o aún para programadores) inspeccionar el código fuente y determinar si la versión del programa realmente toma en cuenta los últimos cambios en las leyes de impuestos o los conocimientos que el mismo posee. Ya que el programa se ejecuta en una forma **procedural**, los conocimientos del programa no pueden ser fácilmente consultados. En un sistema **basado en reglas**, lo que anteriormente se veía como un programa es visto con dos componentes: La **base de conocimientos** y el **procesador de deducciones**, explicados anteriormente.

La figura 1.3 presenta un ejemplo del popular juego del 'gato', escrito en sintáxis del inglés.[3]

```
IF
     there is an 'X' in square A
     AND there is an 'X' in square B
     AND square C is empty
     AND square A-B-C form a line
THEN
     put a 'O' in square C
```

[3] Gill, Karamjit S. Artificial Intelligence for society, 1986

Figura 1.3. Código para resolver el juego de 'gato'

Esta regla expresa una pieza de la estrategia/conocimiento para resolver el 'gato'. Generalmente se esperará que el sistema basado en reglas jugará el juego bien, teniendo cierto número de reglas especificando posibles acciones a tomar en diferentes situaciones. Dependerá del procesador de deducciones determinar cual regla utilizar en diferentes juegos, no importando cómo trabaja, siempre y cuando se obtengan respuestas satisfactorias.

La ventajas más notorias de los sistemas basados en reglas sobre programas convencionales es su transparencia y modularidad. La transparencia se debe a que el conocimiento puede ser examinado directamente, lo cual es muy importante para mantener y depurar programas. La modularidad se refleja del hecho de que el conocimiento es dividido en reglas pequeñas e independientes, lo cual hace muy sencillo cambiar un sistema basado en reglas al añadir o alterar una simple regla. La modularidad también permite que los sistemas puedan aprender por sí mismos hasta cierto grado.

2. De los sistemas basados en reglas a la programación lógica

Aunque es posible construir sistemas basados en reglas que suponen ciertas ventajas como la modularidad y transparencia, en la realidad, los sistemas diseñados no aprovechan esas ventajas. Esto se debe en parte a que hay muchas características en los sistemas basados en reglas que comprometen la independencia de las reglas. El problema principal radica en que las reglas iniciales dependen de acciones y cambios del estado actual. Debido a que el uso de una regla depende de cierto estado, en general un conjunto de reglas que son aplicables en cierto momento, dependen de reglas que han sido escogidas previamente.

Si el creador de las reglas no es cuidadoso, cuando el sistema corre una regla, puede causar un cambio, el cual evita que otra regla pueda ser usada aunque fuera de gran utilidad. En tales situaciones, el orden en el cual el procesador de deducciones decide correr las reglas es muy importante. Una acción común a tomar por el programador es añadir acciones y condiciones a las reglas para asegurar que las reglas corran en cierto orden. Esto, sin embargo, es opuesto a las bases de la programación basada en reglas y hace regresar a las técnicas convencionales de programación, las cuales proveen cierta secuencia de instrucciones.

Si un sistema basado en reglas mantiene una base de datos consistente en los 'hechos' que conoce y las acciones tomadas por

sus reglas son restringidas a crear nuevos hechos, esa clase de problemas no surgirá. Con estas modificaciones, una regla aplicable siempre podrá ser aplicada, y no puede ser negada su ejecución por otra regla.

Tal sistema es esencialmente un **sistema puro de deducciones**, ya que el trabajo del procesador de deducciones es obtener conclusiones de los hechos proporcionados inicialmente, usando reglas como reglas de deducciones. Esta es una de las ideas básicas que soporta la programación lógica.

Un sistema de programación lógica puede ser considerado como un tipo de sistema basado en reglas, donde la base de datos y las reglas son principios lógicos. En tales sistemas, el procesador de deducciones es un probador de teoremas. La ventaja de usar lógica como el lenguaje para nuestras reglas y bases de conocimientos es que en lógica la noción de encadenamiento progresivo es claro y preciso.

Para tener un sistema de programación lógica, es necesario dar al probador de teoremas sugerencias o **control de información** además de las reglas y la base de conocimientos. Estas sugerencias pueden incluir el sugerir que ciertas reglas sean tratadas antes que otras, que ciertas posibilidades no se exploren, etc. De aquí surge la bien conocida **regla de Kowalski:**

Algoritmo = Lógica + control.

Aunque el imponer sugerencias en un sistema basado en reglas pueda resultar en diferentes (o contradictorias) respuestas, la idea de un sistema programado en lógica es que principios lógicos correctos garantizarán que las respuestas sean correctas. El único efecto que el componente de control puede tener es el cambiar el orden en que las soluciones sean encontradas, cambiar el grado de dificultad al encontrar la solución o posiblemente afectar si una solución será o no encontrada.

La discusión anterior sugiere que la programación lógica transformará la forma en que el software es creado, aunque la realidad es un tanto más compleja de lo presentado, pues no existe aún un sistema de programación que cumpla completamente los ideales de la programación lógica.

El lenguaje 'de programación lógica' más utilizado es el **PROLOG**, que aún tiene algunas deficiencias respecto a la forma en utilizar la programación lógica. Por ejemplo, **PROLOG** permite al programador crear 'efectos laterales' que destruyen la independencia de las reglas, a la vez permite al programador escribir oraciones lógicas inadecuadas, las cuales pueden ser compensadas por anotaciones de control apropiadas. **PROLOG** es un gran paso hacia la programación lógica, aunque la programación lógica pura tiene metas aún a largo plazo.

D. ALGUNOS LENGUAJES DE INTELIGENCIA ARTIFICIAL.

La creación de los lenguajes de programación nació a raiz de que las computadoras son capaces de entender ceros y unos únicamente, y esto dificulta en extremo su programación. Existen diversos lenguajes de programación, todos ellos con sus cualidades y defectos propios.

Ya que la Inteligencia Artificial involucra una gran variedad de tipos de información, la diferentes partes del programa deben ser capaces de interactuar para hacer su uso mas efectivo. El uso de **encadenamiento regresivo**, así como de busquedas son vitales para la Inteligencia Artificial, por lo tanto el lenguaje debe ser programado fácilmente para repetir una y otra vez la misma serie de pasos con diferentes datos cada vez.

Asi como otras areas de computación, la Inteligencia Artificial confia más en unos lenguajes que otros. Sin entrar en detalles, a continuación se presentaran algunos lenguajes y sus características básicas, para que el lector pueda formar su propio jucio con respecto a los mismos.

1. LISP.

LISP (LISt Processing) es uno de los primeros lenguajes de programación. Fue creado por John McCarthy por los años 50's, y es ahora uno de los lenguajes de Ingeligencia Artificial más populares en los Estados Unidos. LISP se caracteriza por el uso de listas como elementos básicos. Estas listas no tienen un límite de tamaño, por lo cual pueden ser modificadas y expandidas con facilidad. Además, las listas pueden contener símbolos, variables, palabras, cadenas de palabras, además de otras listas.

Los arreglos son limitados a números o cadenas de un número prefijado de caracteres. Una lista en LISP se representa como un grupo de elementos encerrados entre paréntesis: **(ESTA ES UNA LISTA)** es una lista que contiene cuatro elementos, los símbolos: ESTA, ES, UNA y LISTA.

(ESTA ES (UNA LISTA CON TRES ELEMENTOS)) es también una lista con tres elementos: ESTA, ES y la lista (UNA LISTA CON TRES ELEMENTOS) que contiene cinco elementos a su vez. Un programa en LISP no es nada más que una lista, y como una lista puede contener otras listas, un programa en LISP que pueda controlar otros programas resulta muy sencillo de escribir.

Un programa escrito en LISP puede resultar un tanto difícil de interpretar, pues existen listas dentro de listas, todas ellas

marcadas por paréntesis, pero una vez que se adquiere experiencia en su programación, todo parece tener sentido. Por ejemplo, la expresión D-2 * A + 5/B escrita en varios lenguajes, adquiere una forma diferente en LISP: (-D(+ (* 2 A)(/5 B))), donde las reglas de precedencia son completamente diferentes pero congruentes.

2. PROLOG

PROLOG (PROgramming in LOGic) fue creado alrededor de 1970 por Alan Colmerauer en Francia, y su uso es más popular en Europa. Los japoneses estan usando modificaciones de PROLOG en sus projectos de la "Quinta Generación" para construir máquinas inteligentes.

El propósito de PROLOG es permitir al programador especificar sus tareas en una forma lógica, en lugar de programar la máquina en una forma procedural. PROLOG implementa una estrategia sintáctica manejada por metas, basada en el principio de resolución. El conocimiento en PROLOG esta representado en una notación de predicados, de gran facilidad para interactuar con el sistema experto.

La programación en PROLOG difiere de la programación en LISP, así como de la mayoría de los lenguajes, en que el programador usa

PROLOG para declarar el problema, pero no puede especificar las formas o procedimientos que el sistema usara para alcanzar una solución.

En PROLOG, las declaraciones de control estan separadas por la lógica y son invocadas de una manera transparente al programador. Por ejemplo, en PROLOG se pueden escribir dominios y reglas de la siguiente manera: gusta (ricardo,jane). Este es una aserción acerca de Ricardo y Jane. Ricardo y Jane son escritos en letras minúsculas porque son los nombres de objetos específicos y no son variables. Las variables deben ser escritas con la primera letra en mayúscula: gusta (rocky,X):- gusta (X,tennis) es una regla que expresa que "A Rocky le gusta X si a X le gusta el tennis".

Una vez que tenemos varios factores y reglas, podríamos preguntar al interprete de PROLOG lo siguiente:

?- gusta (rocky,X),

donde la base de datos (de reglas) consiste en las siguientes reglas y aserciones:

```
gusta (rocky,susana).
gusta (rocky,tennis).
gusta (rocky,X):-gusta (X,tennis).
gusta (cesar,tennis).
```

PROLOG respondería de esta manera:

X = susana;

X = tennis;

X = rocky;

X = cesar;

no

Detalles de la forma en que se alcanzó esta solución seran proporcionados en el capítulo referente a PROLOG. Ahora es suficiente con tener una visión general del comportamiento de PROLOG.

3. LOGO

LOGO (del griego, "pensamiento") fue desarrollado en los laboratorios de Inteligencia Artificial del Instituto MIT por Seymour Papert como un lenguaje para ayudar a los niños a desarrollar sus facultades para resolver problemas. LOGO puede ser aprendido fácilmente a través de participación activa, que ayuda a desarrollar una manera lógica de pensar.

Una característica de LOGO diferente a LISP y PROLOG es su gran capacidad para producir imágenes gráficas, además de su capacidad para procesar listas, lo que lo hace un buen candidato para programación de Inteligencia Artificial.

Un programa en LOGO puede ser construído pieza por pieza al escribir pequeñas rutinas que pueden ser incorporadas en rutinas y programas más grandes.

Como ejemplo de las cualidades de LOGO, usemos el contruir una casa. Los comandos en LOGO dicen que dirección tomar y en que ángulo moverse. Cuando un número sigue a los comandos "forward" y "backward" se refiere al número de "pasos" a mover. Cuando un número sigue "right" o "left" indica cuantos grados girar. Una rutina en LOGO para dibujar un cuadrado sería:

```
TO SQUARE
REPEAT 4[FORWARD 50 RIGHT 90]
END
```

4. Smalltalk

El punto de vista de este lenguaje relativamente nuevo es muy diferente al de LISP y PROLOG. LISP usa funciones, PROLOG se concentra en lógica, pero SMALLTALK es "orientado a los objetos" (object-oriented). Todo en SMALLTALK es un objeto. Un objeto no necesita ser un objeto fisico forzosamente, sino que puede ser un conjunto de información.

SMALLTALK programa funciones mandando mensajes de un objeto a otro. Por ejemplo, en el problema '5 + 2', el número 5 mandaría un mensaje al número 2 de "sumate a mí". La naturaleza de este lenguaje lo hace ideal para escribir juegos de aventuras, en los cuales los objetos son mounstruos, castillos o armas.

Cada objeto en SMALLTALK pertenece a una clase especial de objetos, cada uno con sus propios atributos. Por ejemplo, dentro de los mounstruos puede haber gremlins o goblins, pero todos los mounstruos comparten ciertas características como que todos atacarán al aventurero, se esconderán en cavernas y serán extremadamente fuertes, por ejemplo. Las especificaciones del objeto incluyen cómo será manipulado el objeto cuando reciba un mensaje.

Una ventaja que comparte SMALLTALK con LISP y PROLOG es que los programas pueden ser escritos en pequeñas unidades, esto hace sencilla la contrucción de programas de considerable complejidad.

Antes de pasar al próximo capítulo se debe recordar que ningún lenguaje es "perfecto" para cualquier aplicación. Ya que la Inteligencia Artificial se vuelve más sofisticada día con día, diferentes lenguajes o dialectos serán desarrollados para cumplir con las necesidades reinantes. Es a juicio del lector el encontrar un lenguaje que cumpla sus necesidades de programación y respuesta.

CAPITULO III. LISP COMO LENGUAJE DE INTELIGENCIA ARTIFICIAL

LISP COMO LENGUAJE DE INTELIGENCIA ARTIFICIAL

LISP fue diseñado para aplicaciones de Inteligencia Artificial, por lo cual los programas que pueden procesar tareas de razonamiento deben ser capaces de realizar calculos no numéricos. Por ejemplo, un programa que puede procesar lenguaje natural, debe ser capaz de representar las palabras en una oración y sus significados. El tipo de representaciones usadas para tales fines se conocen como expresiones simbólicas.

Los dos tipos básicos de expresiones simbólicas en LISP son los atoms y las listas. Todas las expresiones válidas en LISP consisten en estas dos expresiones. Un atom es un elemento simple, por ejemplo:

cielo
25
secundario
este_es_un_atom_largo

Es fácil de ver que un atom es una cadena de caracteres, números o símbolos de puntuación. Los atoms ofrecen la flexibilidad de representar información no-numérica, por ejemplo, si deseamos representar piezas de ajedrez, sólo definimos atoms como rey, caballo o reina. Es necesario notar que los números son atoms, pero siempre representarán su valor numérico, en contraste con rey, que puede tomar cualquier significado en un programa.

LISP (LISt Procesing) basa su gran capacidad en el uso de las listas. Los siguientes ejemplos presentan listas:

```
(2 4 6 8)
(luis)
(jamón (mayonesa salami))
(+ 5 43)
()
```

Una lista es una secuencia de elementos encerrados entre paréntesis. Cada elemento de la lista puede ser un atom o una lista. Así, **(mayonesa salami)** es una lista que a su vez es elemento de la lista que contiene dos elementos. Las listas pueden tener un único elemento, como en **(luis)** o ser una lista vacía, conocida como nil o **()**.

El uso de las listas es realmente importante en procesamientos simbólicos porque son un mecanismo básico para agrupar símbolos. Las listas pueden ser de cualquier tamaño, con capacidad de encadenamiento sucesivo. Las listas pueden representar una infinidad de símbolos con diferentes características, ya sea un sistema para diagnósticos o un sistema de alimentación.

Es importante tener en cuenta la correspondencia de paréntesis en el uso de listas, esto es, los paréntesis deberán siempre estar bien balanceados. El siguiente ejemplo es una lista bien balanceada:

(A (B (C)) D)

mientras que el siguiente no lo es:

(A (B (C) D)

A continuación se muestran diferentes listas que pueden parecer difícil de analizar:

((clavel violeta))	un componente, una lista.
()	no componentes, una lista vacía o nil.
((A) (123))	dos componentes, ambos son listas.
(() ())	dos componentes, ambos listas vacías.
(((())))	un componente, nil.

LISP siempre representa datos en forma de listas y atoms, tanto para datos de entrada como para obtener resultados. Existen funciones para extraer componentes de las listas, construir nuevas listas o hacer varias operaciones con las mismas. Por lo tanto un programa en LISP, describirá la relación entre la lista de entrada y la lista resultante.

A. FUNCIONES Y VARIABLES EN LISP.

La forma de crear programas en LISP es por medio de definir funciones. Las operaciones fundamentales de LISP son por sí mismas funciones predefinidas que pueden ser usadas para componer funciones nuevas que cumplan las necesidades del programador. Las funciones son aplicadas a datos de entrada (argumentos) y entregan resultados que pueden ser a su vez alimentados a otras funciones como argumentos.

Por ejemplo, una de las funciones predefinadas en LISP es la función **reverse**. Su argumento es una lista y su resultado es una nueva lista, la cual tiene los mismos argumentos, pero presentados en orden inverso. Los siguientes ejemplos muestran su uso:

ARGUMENTO	RESULTADO
(A B)	**(B A)**
((diente reo) (CC) 234)	**(234 (CC) (diente reo))**
(Van Halen)	**(Halen Van)**

Los componentes del resultado de **reverse** son presentados exactamente a la inversa de su argumento inicial, pero la estructura interna de cada componente permanece igual, como puede observarse en la lista del segundo ejemplo.

La notación correcta usada en LISP para denotar el uso de una función a su argumento es el formar una lista en la cual el primer componente es el nombre de la función y el segundo componente es el argumento. Para aplicar **reverse** al argumento **(Van Halen)** debemos de escribir:

(reverse '(Van Halen))

El apóstrofe que precede al argumento de **reverse** es usado correctamente. Ambos, datos y funciones, toman la forma de listas, por lo tanto debe haber una forma de especificar cual es cada cual. El apóstrofe en el ejemplo anterior marca la lista o atom como

datos, esto es, dice al intérprete que no evalúe este argumento como una función, sino como una simple lista.

Cuando usamos un intérprete en LISP, el intérprete nos dice que esta listo para aceptar una llamada a función al imprimir un prompt en la pantalla. En los ejemplos presentados en este capítulo usaremos => como prompt, pero es necesario tomar en cuenta que diferentes sistemas de LISP pueden usar prompts diferentes. Las palabras maracadas en negritas presentan información que puede ser manejada por el intérprete en uso.

reverse es sólo una de las varias funciones internas de LISP. Otra función común es la función **equal**, la cual compara dos listas o atoms. El resultado de la función **equal** es el atom "T" si los dos argumentos son iguales, de otra manera es el atom "NIL" (una lista vacía).

Ejemplos:

```
=> (equal '(piano flauta) '(violín))
NIL
=> (equal '(trama) '(trama))
T
=> (equal '(naranja 8) '((naranja 8)) )
NIL
```

Ahora podemos usar las funciones aprendidas en conjunto para encontrar las palabras palindrómicas. Una frase palindrómica es aquella que permanece igual cuando el orden de sus letras es invertido. Podemos usar la función **equal** para comparar dos listas, y la función **reverse** para invertir el orden de los componentes.

Ejemplos:

```
=> (equal '(B O B) (reverse '(B O B)))
T
=> (equal '(N A O M I) (reverse '(N A O M I)))
NIL
```

Estas expresiones describen la forma de verificar si una frase es palindrómica o no lo es, pero son un tanto agotadoras de escribir, considerando que debemos escribir la misma frase dos veces, una para el primer argumento en **equal** y la otra para la función **reverse**.

Es de esperarse que LISP ofrezca una forma más conveniente de manejar funciones, esto es por medio de la operación **defun**, que nos permite definir funciones a nuestra conveniencia. Esta operación nos deja asignar un nombre a una expresión. La operación de **equal** es de tipo diferente al de **reverse** o **equal** porque deja un record permanente para ser ejecutado. **defun** asigna un nombre a una función, la cual podrá ser utilizada durante toda la sesión de LISP.

La función **defun** es usada para definir funciones. Estos son sus componentes:

1. Nombre de la función	**palindrometest**
2. Argumentos de la lista	**frase**
3. Especificación del resultado	**(equal frase ...)**

El nombre de la función debe ser un atom que empieza con una letra. Los argumentos de la lista deben ser una lista de atoms, todos empezando con una letra. La especificación del resultado designa un resultado para la función y puede invocar otras funciones al usar cualquier combinación de nombres en la lista de argumentos como argumentos para estas funciones, o podrá especificar un valor como datos.

La siguiente definición de **palindrometest** especifica que esta función será una función de un argumento (reconocido como **frase** en la especificación del resultado) y que su valor será el resultado de comparar el argumento (**frase**) al inverso del mismo.

```
=> (defun palindrometest (frase)
     (equal frase (reverse frase)))
palindrometest
```

Como se puede observar, esta función combina **equal** y **reverse** de la misma forma en que se utilizó anteriormente, pero la diferencia es que el argumento tiene el nombre de **frase**, el cual evita el tener que especificar el mismo argumento dos veces. A continuación se presentan unos ejemplos del uso de nuestra nueva función:

Ejemplos:

```
=> (palindrometest '(B O B))
T
=> (palindrometest '(a b c b a))
T
=> (palindrometest '(H a l e n))
NIL
```

Existe una gran infinidad de funciones que pueden ser creadas con **equal** y **reverse** combinadas, las cuales pueden resultar muy interesantes. Asimismo, la mayoría de los sistemas LISP tienen docenas de funciones internas de fácil uso. En este análisis serán omitidas algunas funciones, de las cuales varias son redundantes. A través de los años, LISP ha ofrecido formas alternativas para operaciones equivalentes, y esto trataremos con el fin de hacer este estudio tan conciso como sea posible sin omitir los conceptos centrales. Estas omisiones no afectarán la finalidad de este estudio, el comprender la programación en LISP y tener la habilidad para leer programas creados en este lenguaje.

La función **append** es usada en la construcción de listas también. Esta función toma dos o más argumentos, cada cual deberá ser una lista. La función construye una nueva lista conteniendo los elementos de cada uno de los argumentos. Los ejemplos siguientes serán suficiente para entender el uso de esta función:

```
=> (append '(a b) '(c))
(a b c)
=> (append '(calcio granito) '(cuarzo) '(minerales))
(calcio granito cuarzo minerales)
=> (append '(a) '(b) '(c) '(d (e f)))
(a b c d (e f))
```

1. Cómo extraer componentes de una lista.

La función **reverse** como todas las funciones, entrega una nueva representación de la información inicial. Si el argumento es la

inicial y apellido de una persona, tendremos:

```
=> (reverse '(H Gomez))
(Gomez H)
```

Ahora supongamos que tenemos una lista de nombres ordenados de la misma forma, y nos gustaría tenerlos en el orden de apellido-inicial. En otras palabras, queremos aplicar la función **reverse** a todos los componentes de las lista individualmente. Pero no queremos aplicar la función **reverse** a la lista de nombres como una sola lista, porque eso no reformateará ninguno de los nombres individuales:

```
=> (reverse '( (H Gomez) (R Lopez) (C Castro) )
((C Castro) (R Lopez) (H Gomez))                --> Respuesta erronea!
```

Existen varias formas de lograr nuestro objetivo, todas ellas requiriendo el uso de una nueva función. La función a utilizar es **mapcar**. El nombre no nos dice nada. Desafortunadamente, esta es una característica de muchas funciones en LISP. Se debe recordar que LISP es uno de los lenguajes más antiguos y al ser diseñado los diseñadores se preocupaban más en que sus funciones fueran operacionales que en lo descriptivo que sus nombres parecieran.[4]

mapcar aplica una función a cada uno de los componentes de la lista, individualmente, y entrega como resultado una lista de los resultados de cada una de estas funciones. La función **mapcar** tiene

---

[4]En particular, **car** viene de "contents of address register", que es un componente que los diseñadores de LISP usaron como puntero a un elemento de la lista.

dos argumentos: una función a ser aplicada y una lista de componentes a ser afectada.

Ejemplos:

```
=> (mapcar #'reverse '( (H Gomez) (R Lopez) (C Castro))
((Gomez H) (Lopez R) (Castro C))
=> (mapcar #'reverse '( (A B C) (D E) (F G))
((C B A) (E D) (G F))
```

El símbolo # recuerda a LISP que **reverse** es el nombre de una función. La mayoria de los sistemas LISP no permiten el uso de un nombre de función como un argumento a otra función a menos que el nombre de la función sea precedido por un #.

Hasta ahora solo se han utilizado funciones que pueden cambiar el orden de una lista o comparar las mismas, pero en el manejo de listas es fácil observar que se necesitan funciones con capacidad de extraer componentes localizados en cualquier lugar de la lista. La función **car** extrae el primer componente de una lista. Sólo tiene un argumento, el cual debe ser una lista, no un atom, y su resultado es el primer componente de esa lista. La función **cadr** (otro nombre poco sugestivo) extrae el segundo componente de la lista.

Ejemplos:

```
=> (car '(piano violin guitarra))
(piano)
=> (car '(1 4) 3 5 7)
(1 4)
=> (car '(1 2 3 4))
1
=> (cadr '((hola piolin) (1 2) (ayer lunes))
(1 2)
```

Como se mencionó anteriormente, existe una variedad de funciones relacionadas que llevan nombres semejantes. La función **cad...r** extrae el elemento n+1 de la lista, donde n es una o más **d**'s en la función **cad...r**. Así, **caddr** estrae el tercer elemento de la lista, **cadddr** extrae el cuarto y así sucesivamente.

(car '(a b c d))	resulta en	**a**
(cadr '(a b c d))	resulta en	**b**
(caddr '(a b c d))	resulta en	**c**
(cadddr '(a b c d))	resulta en	**d**

La función **cdr** también acepta una lista como argumento, pero a la inversa sólo regresará la cola de la lista, y borrará el primer elemento de la misma. Por ejemplo, en la lista (a b c) la función **cdr** regresará la lista (b c). La función **cons** inserta un elemento al frente de una lista. Esta función tiene dos argumentos, el primero podrá ser una atom o una lista, mientras que el segundo deberá ser una lista. Esta función regresa una lista en la cual el primer argumento ha sido insertado al principio del segundo argumento.

Ejemplos:

```
=> (cons 'x '(1 2 3))
(x 1 2 3)
=> (cons '(a b) '(f g))
((a b) f g)
```

Si se comprenden claramente las funciones mencionadas, el siguiente ejemplo será fácil de entender:

```
=> (cons (car '(gato perro raton))(cdr'(gato perro raton)))
(gato perro raton)
```

El procedimiento a seguir es primeramente evaluar los argumentos. El primer argumento, (car '(gato perro raton)) regresará **gato**. El segundo argumento, (cdr '(gato perro raton)) regresará la lista **(perro raton)**, y al aplicar **cons**, insertamos **gato** al inicio de la lista **(perro raton)**, obteniendo como resultado la lista presentada.

La función **list** permite crear nuevas listas. Esta función puede tomar cualquier número de argumentos, los cuales pueden ser atoms o listas, y regresa una nueva lista al cubrir los paréntesis alrededor de los argumentos. Es importante notar la diferencia entre **list** y **cons**:

```
=> (list 'a '(b c))
(a (b c))
=> (cons 'a '(b c))
(a b c)
```

list crea una nueva lista, con **a** y **(b c)** como elementos, mientras que **cons** 'abre' la lista **(b c)** y le inserta **a** al principio de la misma.

2. Manejo de variables

Las variables juegan un rol muy importante en cualquier lenguaje de programación, en LISP, las variables son símbolos que

representan otra función. Al igual que las listas, los atoms pueden ser interpretados en LISP de dos maneras. Dentro de una lista, un atom será como tal cuando es precedido por un apóstrofe. A su vez, un atom sin apóstrofe será tratado como un variable. Así, los números son atoms, pero no pueden servir para nombrar variables. Siempre representan su valor numérico, por lo cual no deben ser precedidos por un apóstrofe.

Ejemplos:

```
=> (+ 3 5)
8
=> (- 5 2)
3
```

Podemos asignar un valor a un atom con el operador **setq**. Por ejemplo, podemos asignar el valor de 25 al atom **y** de la manera siguiente:

```
=> (setq y 25)
25
```

setq tiene dos argumentos, evalúa primeramente el segundo argumento, le asigna el resultado al primer argumento y regresa el valor del segundo argumento, asi pues, el primer argumento deberá ser un atom no-numérico. Ahora se podrá usar el atom **y** en expresiones para representar el valor de 25.

Ejemplos:

```
=> (cons 'y '(a b))
(y a b)
=> (cons y '(a b))
(25 a b)
```

En el primer ejemplo, **y** es precedida por un apóstrofe, por lo

cual LISP la trata literalmente, sin importar que se le haya asignado un valor anteriormente. En el segundo caso, y es evaluada, pues no es precedida por un apóstrofe, y es considerada por LISP como una variable con valor 25, insertada a la lista **(a b)**.

Debemos considerar que en LISP un valor puede ser cualquier expresión, no únicamente un valor numérico. Es cierto que una variable puede tener sólo un valor a la vez, pero una gran ventaja que ofrece LISP es que este valor puede ser una lista:

Ejemplos:

```
=> (setq musica '(flauta violin arpa))
(flauta violin arpa)
=> musica
(flauta violin arpa)
=> 'musica
musica
=> (car musica)
flauta
=> (cdr musica)
(violin arpa)
=> (list '(piano solfeo) musica)
((piano solfeo)(flauta violin arpa))
```

Los ejemplos anteriores dejan ver lo útil que las variables pueden ser, especialmente si una lista será usada varias veces, evita el escribirla repetidas veces. Así, un programa escrito para entender textos en lenguaje natural deberá tener facilidad de accesar variables cuyo valor es una lista de oraciones individuales, cada una de la cual es una lista de frases, y a su vez cada frase es una lista de palabras (atoms).

Hasta aquí las variables consideradas han sido variables globales, pues retienen su valor hasta que su valor es cambiado con otra llamada de **setq** o terminada una sesión de LISP. Los parametros usados al definir funciones son considerados variables locales, pues solo conservan su valor a traves del uso de una función.

Ejemplos:

```
=> (defun doble(num)
     (* num 2))
doble
=> (doble 7)
14
=> num
Unbound variable:num
```

Cuando **doble** es llamado en estos ejemplos, al parametro **num** se le asignó el valor de 7, pero fuera de ser utilizado en una función, si se trata de recuperar el valor de **num**, se obtendrá un mensaje de error, pues la variable es considerada sólo como variable local de la función **doble**.

3. Comparando elementos y condiciones.

LISP ofrece una gran variedad de funciones con fines comparativos, los cuales pueden tener uno o varios argumentos, ya sean atoms o listas. La mejor manera de entender el uso de ciertas funciones es por medio de ejemplos ilustrativos. Los siguientes

ejemplos son fácil de entender y se explican por sí mismos. Es importante entender estas funciones con claridad, pues serán utilizadas posteriormente.

```
=> (atom 'perro)                    _ Es realmente un atom? (t o nil)
t
=> (atom '(perro))
nil

=> (listp 'perro)                   _ Es argumento una lista? (t o nil)
nil
=> (listp '(perro))
t

=> (numberp 10)                     _ Es argumento un número?
t
=> (numberp '(10))
nil

=> (zerop 5)                        _ Es argumento igual a cero?
nil
=> (zerop nil)
nil

=> (null nil)                       _ Es argumento igual a nil?
t
=> (null '(a b c))
nil
=> (null (car '(x y z)))
nil

=> (oddp 5)                         _ Es argumento número non?
t
=> (oddp 4)
nil
=> (oddp 19)
t

=> (< 5 6)                          _ Es 5 menor que 6?
t
=> (< 5 5)
nil
=> (< 6.5 5)
nil

=> (> 5 6)                          _ Es 5 mayor que 6?
nil
=> (> 5 5)
```

```
nil
=> (> 6.5 5)
t
```

La finalidad de esta sección es escribir funciones que ofrezcan capacidad de análisis, las funciones anteriores nos permiten realizar pruebas distintas y ahora ya poseemos las herramientas para realizar llamadas a funciones. Consideremos el ejemplo siguiente:

```
=> (defun testciudad(ciudad)
     (cond ((equal ciudad 'puebla) 'buena)
           ((equal ciudad 'chapala) 'mala)
           (t 'o.k.)))
```

La función **testciudad** regresará **buena** si la ciudad escogida es puebla, de otra forma regresará **mala** como respuesta. Un **cond** puede tener uno o más argumentos, cada uno de los cuales deberá ser una lista. Los argumentos de **cond** son llamados casos, y deberán tener al menos dos elementos a comparar. En este ejemplo existen tres casos:

```
[1]  ((equal ciudad 'puebla) 'buena)
[2]  ((equal ciudad 'chapala) 'mala)
[3]  (t 'o.k.)))
```

Lisp evalúa los casos de **cond** uno a la vez, del primero al último, y tan pronto como un caso sea verdadero, se realiza la acción especificada y la evaluación de **cond** termina (no se analizan

los casos posteriores), regresando el valor de la acción tomada por el caso. Consideremos esta llamada a **testcidudad:**

```
=> (testciudad 'chapala)
mala
```

En este ejemplo, al parámetro **ciudad** se la ha asignado el valor de **chapala**, y se procede a evaluar el primer caso de **cond**, **(equal ciudad 'puebla)** el cual regresa **nil**, por lo cual la acción para este caso no se lleva a cabo. Al evaluar el segundo caso, **(equal ciudad 'chapala)**, regresa **t**, por lo cual la acción correspondiente es llevada a cabo, y no se evalúan más casos posteriormente. La acción en este caso es el atom **mala**, y la función entrega esta respuesta. Ahora consideremos el siguiente ejemplo:

```
=> (testciudad 'toronto)
o.k.
```

En este caso, al parámetro ciudad se la ha asignado la ciudad de **toronto**. Al ser probado el primer caso, **cond** entrega como respuesta **nil**, al igual que al ser evaluado el segundo caso. Finalmente, se evalúa el tercer caso, pero aquí se encuentra algo extraño, el primer argumento en el tercer caso es **t**, pero **t** siempre se evalúa a sí mismo, y como resultado siempre se entrega **t**, que es un valor diferente de **nil**. Así, en este ejemplo, el tercer caso entregará **o.k.**, que fue la acción programada para **t**.

4. Funciones lógicas

Hasta ahora se han venido presentando varios predicados que LISP ofrece para hacer pruebas y manipulaciones con listas y atoms. Las funciones lógicas **or**, **not** y **and** pueden ser usadas en conjunto con predicados para realizar pruebas más poderosas.

La función lógica más simple es **not**, que acepta un sólo argumento, el cual puede ser cualquier expresión válida en LISP . Si el predicado es evaluado como **nil**, **not** entregará **t**, que es la negación de la respuesta. Así pues, **not** entregará **t** si su argumento no es verdadero.

Ejemplos:

```
=> (not nil)
t
=> (not t)
nil
=> (not (atom '(a b c))
t
```

La función lógica **or** acepta uno o más argumentos. Evalúa sus argumentos de izquierda a derecha y regresa el primer valor que encuentra diferente de **nil**. Si todos los argumentos evaluados son **nil**, la función **or** regresará como respuesta **nil**.

Ejemplos:

```
=> (or t nil)
t
=> (or nil nil)
nil
=> (or (zerop 5) (1 2 5) (* 2 3))
(1 2 5)
```

La función lógica **and** también toma uno o más argumentos y evalúa de izquierda a derecha. Si encuentra un argumento que sea **nil**, inmediatamente entregará **nil** como respuesta sin evaluar más argumentos. Si todos los argumentos entregaron como respuesta un valor diferente de **nil** (verdaderos), **and** entregará el valor del argumento finalmente analizado (el de la extrema derecha).

Ejemplos:

```
=> (and 5 nil)
nil
=> (and 'a 'b '(1 2 3))
(1 2 3)
=> (and (zerop 0) (atom 1) '(a b))
(a b)
```

Ejemplo: Escribamos una función para probrar listas con dos o menos elementos. Una lista satisface esta condición si es una lista vacía (nil). Si no es una lista vacía, pero es una lista con uno o dos componentes también satisface nuestra condición. En LISP podríamos describir esta función de la manera siguiente:

```
=> (defun hasta_dos (lista)
     (cond ((null lista) 't)                 No tiene componentes
           ((null (cdr lista) 't)            Tiene un componente
           ((null (cddr lista) 't)           Exactamente 2 componentes
           (t 'ok)))
hasta_dos
```

Esta forma de escribir esta función no es realmente la más adecuada. Una lista tiene dos o menos elementos si: es una lista vacía, o si tiene un componente o si tiene exactamente dos componentes. Resulta claro que **or** nos ofrece una ventaja para constuir esta función en una manera más clara:

```
=> (defun hasta_dos (lista)
     (or  (null lista)
          (null (cdr lista))
          (null (cddr lista))
     ))
hasta_dos
```

Ejemplo: Consideremos una función que pueda comparar dos números, y tenga la capidad de informarnos si esos dos números son iguales, ambos son pares, ambos son nones, si son negativos o simplemente no tienen nada en común. Si utilizamos las funciones que conocemos hasta el momento, es fácil comprender la siguiente función:

```
=> (defun compara(a b)
     (cond ((equal a b) 'iguales)
           ((and (oddp a) (oddp b)) 'ambos_nones)
           ((and (not (oddp a))(not oddp b))) 'ambos_pares)
           ((and (< a 0) (< b 0) 'ambos_negativos)
           (t 'no_similares)))
compara
```

La función acepta como argumentos **a** y **b**, y se procede a comparar si son iguales por medio de la función **equal**. Si no son iguales, **cond** procede a evaluar el siguiente caso, el cual utiliza la función **oddp** para probar si **a** y **b** son ambos nones. El tercer caso probará si ambos son pares al utilizar la función **not** en combinación con **oddp**. El cuarto caso utilizará la función menor-que para comparar **a** y **b**. En caso de que ningún caso entregue como respuesta **t**, simplemente **t** entregará la respuesta de **no_similares**. Los siguientes son ejemplos del uso de **compara**:

```
=> (compara 5 5)
iguales
=> (compara 3 5)
ambos_nones
=> (compara -2 -3)
```

```
ambos_negativos
=> (compara -5 -5)
iguales
```

5. Funciones iterativas.

En LISP existen varias funciones que tienen capacidad de realizar una secuencia de acciones controladas. Para clarificar el uso de las funciones iterativas, supongamos que necesitamos una función que leerá tres números, los sumará y entregará como resultado la suma de ellos mismos. Si se quiere imprimir (print) un prompt antes de cada input, dicha función sería algo similar a:

```
=> (defun suma_tres()
     (let ((sum 0))
          (print "Numero?")
          (setq sum (+ sum (read)))
          (print "Numero?")
          (setq sum (+ sum (read)))
          (print "Numero?")
          (setq sum (+ sum (read)))))
suma_tres
```

Esta función presenta tres llamadas a función idénticas que imprimen un prompt y tres funciones idénticas que leen los tres números y los suman a sí mismos al total. Es obvio que esta no es la mejor manera de escribir acciones que se repiten, especialmente si esta función consistiera en sumar diez números. LISP ofrece una forma especial, **loop**, para realizar iteraciones.

loop puede tomar cualquier número de argumentos, los cuales LISP evalúa repetidamente en secuencia hasta que es forzado a salir. El patrón de **loop** es el siguiente:

```
(loop <expresión1>
      <expresión2>
          .
          .
      <expresiónn>)
```

Cada expresión en el ciclo es evaluada en orden de izquierda a derecha, y de arriba hacia abajo en orden secuencial una y otra vez. LISP continuará repitiendo estas instrucciones en el ciclo hasta que encuentre una expresión que lo forze a salir del mismo.

Consideremos el argumento de entrada a la función como el controlador de la iteración, este nos dirá cuando salir del ciclo iterativo. El ciclo será ejecutado repetitivamente hasta que lea un valor específico, y al ser encontrado saldrá del mismo. Consideremos una forma iterativa de escribir la función **palindrometest** analizada anteriormente. Esta versión leerá un dato cada vez dentro del ciclo e imprimirá el palindrómico correspondiente:

```
=> (defun palindrometest()
     (let (input)
          (loop
               (print "Dame una lista, -atom = salir- :")
               (setq input (read))
               (cond ((not (listp input)) (return 'terminado)))
               (print (append input (reverse input))))))
palindrometest
```

En esta versión, cada vez que un elemento se asigna a **input**, una variable local, el valor es usado para determinar cuando salir del ciclo. La prueba para salir del ciclo es **listp**, si es **input** una lista o no lo es. Si **input** no es una lista, el ciclo terminará y LISP devolverá el control al usuario. Si **input** es una lista, se imprimirá el palindrómico correspondiente. Los siguientes ejemplos ilustran el uso de esta función.

```
=> (palindrometest)
Dame una lista, -atom = salir- : (uno dos tres)
(uno dos tres tres dos uno)
Dame una lista, -atom = salir- : (b o b)
(b o b b o b)
Dame una lista, -atom = salir- : un_atom
terminado
```

Ejemplo: Consideremos el siguiente ejemplo, que consiste en adivinar una palabra secreta programada por el usuario. Este programa contará el número de intentos utilizados para encontrar la palabra correcta.

```
=> (defun adivina(pal_secreta)
     (let (input) (count 1))
          (loop
               (print "Tu palabra?:")
               (setq input (read))
               (cond ((equal input pal_secreta) (return count)))
               (setq count (1 + count)))))
adivina
```

Cada vez que no pasamos por el ciclo, aumentamos el contador **count** (variable local), el cual tendrá como dato el número de intentos en encontrar la palabra clave. Al encontrar la palabra clave, el programa entregará el número de intentos como dato.

B. RECURSION EN LISP

Una función recursiva es simplemente aquella que se utiliza a sí misma como argumento en su propia definición. Estas funciones son realmente útiles, pues existen casos en los que una acción debe ser ejecutada repetitivamente. Las funciones recursivas son la base para contruir programas en estilo denotacional, estas hacen posible que las definiciones con un número finito de términos puedan describir objetos de un tamaño variable, pues es muy común para un programador el no conocer de antemano el número de datos o iteraciones que necesitará una función para obtener un resultado esperado.

Los programas que contienen un número finito de símbolos, deben describir la relación entre los datos de entrada y los resultados de tamaño variable. Existen varias funciones predefinadas en LISP con esta característica. Por ejemplo, la función **append** entrega un resultado cuyo tamaño depende del tamaño de los datos de entrada, los cuales no son sujetos a ninguna regla definida de **append**. Por lo tanto, podemos decir que una función que utilize **append** puede describir cualquier un resultado de cualquier tamaño. Otras funciones, tal como **reverse, cons, cdr** y otras, también tienen esta característica.

Ninguna de las funciones mencionadas transforma los datos de entrada en una forma de componente-componente. Ya analizamos la función **mapcar**, que aplica una función a cada componente de una lista de argumentos y entrega un resultado con el mismo número de componentes que la lista inicial. Si necesitamos un resultado con más (o menos) componentes, lo podrémos formar usando las funciones **car, cdr, append, list**, y otras en combinación.

Sin embargo, **mapcar** no es la herramienta más conveniente para describir todos los tipos de transformaciones a una lista. Al visualizar el resultado en una forma circular (recursiva) podemos contruir funciones más simples. Pongamos por ejemplo, que hay una colección de datos agrupados como una lista cuyos componentes son atoms representando nombres de la forma inicial-apellido. Esto es, los atoms aparecen en pares, el primer elemento de cada par es la inicial del nombre y el segundo el apellido, tal como: (F Gomez C Cardenas L Aguilar).

Ahora supongamos que queremos una versión transformada de esta lista, en la cual los inicial/apellido sean agrupados como componentes simples. Esto es, cada componente de la lista final será una lista de dos componentes, el primero una inicial, y el segundo un apellido. La funciones recursivas definidas en LISP generalmente tienen la característica de que en cualquier punto de la recursión, los datos de entrada habrán sido simplificados en alguna forma.

La función **pares** mostrada expresa una definición recursiva en notación de LISP:

```
=> (defun pares (nombre)
          (if (null nombre)
               '()
               (cons (list (car nombre) (cadr nombre))
                     (pares (cddr nombre)))
          ))
```

Ejemplo:

```
=> (pares '(F Gomez   C Cardenas   L Aguilar))
( (F Gomez) (C Cardenas) (L Aguilar) )
=> (pares '(R Lopez S Glez))
( (R Lopez) (S Glez) )
```

Esta definición especifíca que el resultado podrá ser de una de dos formas posibles: A) una lista vacía o B) una lista cuyo primer componente es una lista de dos componentes construída de la lista de dos componentes de los datos de entrada, y el resto de los componentes que son el resultado de aplicar la función a una lista más simple, la cual es el argumento original expecto que los primeros dos componentes han sido removidos.

Se puede considerar el definir funciones recursivas muy similar a la construcción de pruebas por medio de inducción matématica, si se entiende una, se podrá entender la siguiente. La mejor forma de entender las funciones recursivas es a base de experimentar y crear nuevas funciones.

Ejemplo: Este ejemplo extrae el primer bloque de componentes de una lista. La función tendrá dos argumentos: el primero nos dirá cuantos componentes extraer del segundo argumento. Si el segundo argumento no tiene el número de componentes especificado, entonces el resultado debería ser idéntico al segundo argumento.

El resultado buscado es simple cuando el bloque es una lista vacía (nil). En ese caso, el resultado debería no tener componentes, o sea, ser una lista vacía. El resultado es también sencillo cuando el segundo argumento es una lista vacía, por lo cual el resultado sería una lista vacía a su vez. Si ambos, el bloque y el componente son no-vacíos, entonces el primer componente del resultado será el primer componente del segundo argumento, y el resto de los componentes del resultado deberían ser tomados del segundo argumento y así sucesivamente para el resto de los componentes. En notación LISP esto sería:

```
=> (defun prefijo (bloque lista)
     (if  (null bloque)  '()
          (if (null lista)  '()
               (cons (car lista)
                    (prefijo (cdr bloque)
                             (cdr lista)))
          )))

=> (prefijo '( 1 2 3 ) '(d i s n e y))
(d i s)
=> (prefijo '( 1 2 ) '(numero (dos) sesenta))
(numero (dos))
=> (prefijo '() '(cuantos elementos? )
nil
```

Ejemplo: La siguiente función es una versión de la función de LISP **append**. Esta manera de construirla muestra lo útil que la recursión es al contruir funciones en la cual no se sabe el número de iteraciones que llevará a la función para encontrar la respuesta buscada.

```
=> (defun append(l1 l2)
     (cond
          ((null l1) l2)
          (t(cons (car l1) (append (cdr l1) l2)))))
```

La función **append** mostrada, unirá la lista l1 y con la lista l2, la función recursiva parará cuando la lista l1 sea la lista vacía (nil), en donde será mostrada la lista l2. Resulta claro el uso de las funciones **cons** y **car** para alcanzar el objetivo de juntar ambas listas.

Ejemplo: La siguiente función borrará el elemento **e** de la lista l en forma recursiva. El uso de la función **equal** es muy necesario y claro de enteder, pues simplemente comparará el elemento e con los elementos de la lista.

```
=> (defun borra (e l)
     (cond
          ((null l) '())
          ((equal e (car l)) (delete e (cdr l)))
          (t (cons (car l)(delete e (cdr l))))))
```

Ejemplo: La siguiente función, **intersecta**, dará como respuesta la intersección de dos listas. La intersección de dos listas es la lista de los elementos comunes a ambas listas, asumiendo que las

listas no tienen elementos repetidos. El uso de la función member es claro, y las llamadas recursivas utilizan la función cdr, que remueve un elemento de la lista que pasa como argumento.

```
=> (defun intersecta (l1 l2)
     (cond
          ((null l1) nil)
          ((member (car l1) l2)
               ((cons (car l1) (intersecta (cdr l1) l2)))
          (t (intersecta (cdr l1) l2))))

=> (intersecta '(a e d c b) '(e z t r b))
(e b)
```

C. CORRESPONDENCIA DE PATRONES.

La correspondencia de patrones (pattern matching) es un método muy efectivo de resolver problemas relacionados con Inteligencia Artificial, el cual implica tomar patrones o descripciones y encontrar los elementos en una base de datos que corresponda a los patrones. Esta técnica es la base del PROLOG y también la base para sistemas de producción, que han demostrado ser muy útiles al implementar modelos de conocimiento humano y sistemas expertos en Inteligencia Artificial.

Como un ejemplo de correspondencia de patrones, consideremos el siguiente conjunto de siete factores que han sido unidos por una lista:

```
=> (setq database '((Jorge   es_papa_de Pedro)
                    (Jorge   es_papa_de Ricardo)
                    (Maria   es_mama_de Pedro)
                    (Pedro   es_papa_de Alicia)
                    (Diana   es_mama_de Tim)
```

```
(Ricardo es_papa_de Miguel)
(Alicia  es_mama_de Juan)))
```

Es fácil observar que cada factor ha sido codificado como una lista encadenada, la cual se le llamará claúsula. Ahora bien, si queremos saber quienes son los hijos de Jorge, simplemente necesitaremos buscar cada claúsula en la base de datos en la cual el primer elemento es Jorge y el segundo elemento sea **es_papa_de**. En otras palabras, necesitamos encontrar cada claúsula que concuerde con el siguiente patrón:

```
(Jorge es_papa_de <hijo>).
```

Por lo tanto, un patrón es un tipo de plantilla, es una descripción de la información que estamos buscando en la base de datos. En reconocimiento de patrones, se accesa información en una base de datos al encontrar claúsulas que correspondan a dichos patrones. Cuando se encuentra una claúsula que corresponde al patrón, el elemento que aparece en la posición de <hijo> será el nombre de uno de los hijos de Jorge en la base de datos.

Es importante notar que hay dos tipos de componentes en el patrón arriba mencionado. Algunos componentes, como **Jorge** o **es_papa_de**, sólo pueden corresponder a atoms idénticos en la base de datos, y son llamados **patrones constantes.** El otro componente, <hijo>, es como un comodín, puede corresponder a cualquier elemento en la base de datos. Este tipo de patrón es conocido como **patrón**

variable. Los patrones variables serán representados por atoms que inician con un signo de igual, como =x; cualquier otro atom que aparezca en un patrón es una constante. Así, un patrón que podría ser utilizado para encontrar los hijos de Jorge sería:

(Jorge es_papa_de =x),

y este patrón correspondería a dos claúsulas en la base de datos:

(Jorge es_papa_de Pedro).

(Jorge es_papa_de Ricardo).

Ahora escribamos una función para encontrar patrones, esta función aceptará un argumento, un patrón y encontrará las claúsulas que corresponden al patrón en la base de datos, guardada en la variable global **database.** Un patrón no está fijado a una sola variable, sino que se pueden usar dos variables, como:

(=papa es_papa_de =hijo),

que buscará en la base de datos quien es hijo de cada papá. A continuación se presentan unas llamadas a la función **match**, y la información que se proporciona por la misma:

```
=> (match '(Jorge es_papa_de =x))
(((=x Pedro)) ((=x Ricardo)))
=> (match '(=x es_mama_de =y))
(((=x Maria)(=y Pedro)) ((=x Diana)(=y Tim))((=x Alicia)(=y Juan)))
```

Estos ejemplos muestran que la función **match** regresará una lista en la cual cada correspondencia es representada por una lista. Cada correspondencia es llamada también una **instancia** del patrón. En el primer ejemplo, se obtuvieron dos instancias, por

lo cual **match** regresó como respuesta dos listas. El patrón sólo era la variable **=x**. En el segundo ejemplo, el patrón contiene dos variables, por lo cual se esperan dos instancias para cada una de las tres claúsulas en la base de datos.

Ahora que es comprendido el uso de la función **match**, se presenta el código de la función:

```
=> (defun match (patron)
     (mapcan # '(lambda (x) (match-clausula patron x)) database))
```

La función aceptará un argumento, el cual es un patrón. **match** utiliza la función **map**, que 'mapea' una sub-función **match-clausula**, para comparar el patrón de cada claúsula en la base de datos. Cuando una claúsula corresponde al patrón, **match-clausula** regresa una lista de las variables instanciadas; cuando un patrón no corresponde, **match-clausula** regresará nil. Así, si llamamos la función **match** con el patrón **(Jorge es_papa_de =x)**, **match-clausula** regresaría la siguiente serie de valores:

```
(((=x Pedro)))
(((=x Ricardo)))
nil
nil
nil
nil
nil
```

La función **mapcan** unirá estas listas y así borrará todos los **nil's**. Por lo cual, con esta función, **match** entregaría unicamente: **(((=x Pedro)) ((=x Ricardo))**. Ahora podemos mostrar la sub-función **match-clausula** de esta manera:

```
=> (defun match-clausula (pat dat)
```

```
(filter (list  (elem-match (car pat) (car dat))
               (elem-match (cadr pat) (cadr pat))
               (elem-match (caddr pat) (caddr dat)))))
```

La función **match-clausula** acepta dos argumentos, un patrón y una claúsula de la base de datos. A su vez, **match-clausula** llama a la sub-función **elem-match** tres veces para comparar cada componente del patrón con el componente correspondiente de la claúsula de la base de datos. Los resultados de las tres llamadas son colocados en una lista, y la función **filter** toma este resultado, checa por una correspondencia completa de los tres elementos, y si todos corresponden, construye una lista de las variables. Finalmente, la función **elem-match** es de la siguiente forma:

```
=> ((defun elem-match (patron data)
     (cond
          ((variablep patron) (list patron data))
          ((equal patron data) t)
          (t nil)))
```

D. SISTEMAS DE PRODUCCION.

Ahora que hemos generado algunas funciones que realizan correspondencia de patrones, incorporémoslas en un sistema completo, que pueda realizar tareas de razonamiento, tal sistema es conocido como sistema de producción , y tiene tres componentes básicos:

[1] Una base de datos de claúsulas, generalmente llamada memoria de trabajo.

[2] Un conjunto de reglas, llamadas producciones

[3] Un programa que aplica las reglas a la base de datos, llamado intérprete de reglas.

Cada producción es una regla **if-then** que es muy similar a la función **cond**. Una regla **if-then** especifica una prueba a ser realizada y una acción a tomar si la prueba se satisface. En una producción, la prueba toma la forma de un patrón a corresponder contra la memoria de trabajo, y la acción especifica una claúsula por añadir a la memoria de trabajo. El siguiente ejemplo muestra una regla de producción:

```
IF ((=x es_papa_de =y)
    (=z es_esposa_de =x))
THEN (=z es_mama_de =y)
```

El patrón, o sea la parte **IF** de la regla, correlaciona una situación en la cual una persona **=x** es papá de **=y** y esposa de **=z**. La acción, o la parte **THEN**, infiere por lo tanto, que **=z** es la mamá de **=y**. Esta producción será codificada por la siguiente estructura:

```
(p1
     ((papa =x =y)
      (esposa =z =x))
= =>
     (mama =z =y))
```

Como se puede ver, una producción toma la forma de una lista, que consiste de cuatro elementos:

1. El nombre de la producción, en este caso **p1**.
2. Una lista con uno o más patrones que especifican una prueba.
3. Una flecha doble que separa las pruebas de la acción, con el fin de que sean fácilmente distinguidos.
4. Una acción, o mejor dicho, un patrón que describe una claúsula a ser añadida a la memoria de trabajo.

Por ejemplo, supongamos que aplicamos la producción **p1** a la siguiente base de datos:

```
((papa Jorge Pedro)
 (esposa Alicia Jorge)
 (papa Enrique Maria)
 (mama Maria Alicia))
```

obtendríamos una sóla correspondencia, en la cual **=x** se relaciona con Jorge, **=y** con Alfredo, y **=z** con Alicia, por lo tanto, la producción añadiría la claúsula **(mama Alicia Pedro)** a la memoria de trabajo o base de datos.

Es importante notar que una producción puede corresponder a la base de datos en más de una manera. Por ejemplo, si añadiéramos la claúsula **(esposa Susana Enrique)** a la base de datos, entonces la producción **p1** correspondería dos veces, y creando los siguientes:

```
(((=x Jorge) (=y Pedro) (=z Alicia))
 ((=x Enrique) (=y Maria)(=z Susana)))
```

Hasta ahora, hemos discutido producciones simples, pero un sistema de producciones tendrá generalmente una colección de

producciones a ser aplicadas a la memoria de trabajo. Esta colección de producciones es conocida como **set de producciones.** La tabla 3.1 presenta producciones que trazan inferencias acerca de relaciones entre los familiares:

```
=> (setq producciones '(
     (p1
          ((papa = x =y)
          (esposa =z =x))
     = =>
          (mama =z =y))
     (p2
          ((mama =x =y)
          (esposo =z =x))
     = =>
          (papa =z =y))
     (p3
          ((esposa =x =y))
     = =>
          (esposo =y =x))
     (p4
          ((esposo =x =y))
     = =>
          (esposa =y =x))
     (p5
          ((papa =x =z)
          (mama =y =z))
     = =>
          (esposo =x =y))
     (p6
          ((papa =x =z)
          (mama =y =z))
     = =>
          (esposa =y =z))
     (p7
          ((esposo =x =y))
     = =>
          (hombre =x))
     (p8
          ((esposa =x =y))
     = =>
          (mujer =x))))
```

TABLA 3.1. Un conjunto de producciones.

1. El interprete de reglas: aplicando producciones.

Hasta ahora sólo se han visto aplicaciones a una sóla producción, pero es importante saber el funcionamiento de un sistema de producción para realizar tareas. Asumiremos que nuestro sistema emplea dos variables globales, una que guarda nuestro conjunto de producciones y otra que tiene memoria de trabajo. El intérprete de reglas es el programa que aplica las producciones a las claúsulas en la memoria de trabajo.

La unidad básica de procesamiento al aplicar producciones es el **ciclo**, el cual es equivalente a un paso en un ciclo. En cada ciclo, el intérprete de reglas realiza tres pasos: Primero, aplica todas las producciones del conjunto de producciones a la memoria de trabajo y encuentra todas las producciones cuyas condiciones son satisfechas. Este conjunto de producciones correspondidas es llamado el **set conflicto**. Después, el intérprete de reglas selecciona una producción del set conflicto, añadiendo la claúsula especificada a la memoria de trabajo. Finalmente, el intérprete de reglas imprime un registro de la producción y la claúsula que fue añadida a la memoria de trabajo. Después de realizar estos pasos, el intérprete de reglas procede al siguiente ciclo y las repite sucesivamente.

Para especificar al sistema de producciones cuando deje de ciclar, se necesita añadir un control, el cual no añadirá una claúsula a la memoria de trabajo si ya existe. Así, cuando una producción añadiera una claúsula a la memoria de trabajo que ya existe, la producción no puede seguir.

Así pues, hemos definido un sistema que acepta un conjunto de factores (memoria de trabajo) y un conjunto de reglas (el conjunto de producciones) y traza las posibles inferencias que pueden surgir en base a los factores y las reglas. Supongamos que asignamos la siguiente memoria de trabajo a la variable global **database**:

```
=> (setq database '((hombre Pedro)
                    (mama Diana Pedro)
                    (papa Jorge Pedro)
                    (papa Jorge Maria)
                    (esposa Maria Tomas)
                    (hija Sally Tomas)))
```

Veamos que pasaría cuando corrieramos el sistema de producciones con las ocho reglas de la tabla 3.1. En el primer ciclo, las siguientes producciones corresponderían: **p3, p5, p6 y p8**. Sólo una producción puede ser seleccionada por cada ciclo. Para simplificar las cosas, siempre seleccionaremos la primer producción en el set conflicto, por lo cual sólo **p3** sería seleccionada. Como resultado, la claúsula **(esposo Tomas Maria)** se añadiría a la memoria de trabajo, y nuestro sistema de producción imprimiría la siguiente expresión:

```
(firing p3 (esposo Tomas Maria))
```

Si dejamos al sistema correr libremente, la siguiente información aparecería en pantalla:

```
(firing p3 (esposo Tomas Maria))
(firing p5 (esposo Jorge Diana))
(firing p4 (esposa Diana Jorge))
(firing p1 (mama Diana Maria))
(firing p7 (hombre Tom))
(firing p7 (hombre Jorge))
(firing p8 (mujer Maria))
(firing p8 (mujer Diana))
done
```

En el noveno ciclo ninguna de las instancias en el set conflicto es capaz de añadir una claúsula nueva a la memoria de trabajo, por lo cual la ejecución del sistema de producción termina.

El uso de intérpretes de reglas y bases de datos es de gran importancia al aplicar la Inteligencia Artificial. Al mostrar las funciones básicas de LISP y algunas implementaciones adicionales, se facilita el entendimiento de la parte final de este capítulo, y es más claro el porqué los sistemas expertos pueden 'aprender' por sí mismos a partir de información adquirida a través de 'experiencias' e información almacenada en sus bases de conocimientos.

Actualmente LISP es el lenguaje de programación más popular en los Estados Unidos, en lo que a Inteligencia Artificial concierne, y más y más versiones mejoradas de LISP se encuentran

frecuentemente. Hacia donde marcha la Inteligencia Artificial? Que podrémos enseñar a las máquinas para aprender de ellas? Estas preguntas serán fácil de contestar en pocos años, cuando la Inteligencia Artificial y sus diversas ramas dejen de ser un área de investigación y pasen a ser una disciplina popular.

CAPITULO IV. PROLOG COMO LENGUAJE DE INTELIGENCIA ARTIFICIAL

PROLOG COMO LENGUAJE DE INTELIGENCIA ARTIFICIAL

El propósito de este capítulo no es discutir en una manera abstracta las grandes cualidades de PROLOG, sino demostrarlas de una manera sencilla, ayudandose de ejemplos ilustrativos. Cada uno de los ejemplos mostrados en las siguientes secciones es relativamente simple, pero al mismo tiempo, ofrece una visión del poder de este lenguaje y una idea de la variedad de posibilidades de aplicación real para lo cual PROLOG es una herramienta muy popular. El primer paso a dar es mostrar la terminología usada en PROLOG.

A. SINTAXIS Y TERMINOLOGIA DE PROLOG

Un programa en PROLOG consiste de un conjunto de procedimientos, cada uno de los cuales define una relación lógica en particular, comúnmente llamada predicado. Un procedimiento consiste de una o más declaraciones o claúsulas, que tienen la forma:

P0 :- P1, P2, ... Pn.

las cuales pueden ser leídas declarativamente como:

"P0 es verdadero si P1 y P2 y ... Pn son verdaderos"

o de una forma procedural:

"Para satisfacer la meta P0,
satisface la meta P1, después

satisface la meta P2, después

.

.

.

satisface la meta Pn".

Es importante notar que cualquier claúsula en PROLOG es terminada por un punto, **P0** es llamada la cabeza principal y el conjunto de metas **P1,P2 ... Pn** forman el cuerpo. Una claúsula sin un cuerpo, por ejemplo, una claúsula de la forma:

P.

es una claúsula unidad o factor, que significa:

"P es verdadero" o "la meta P es satisfecha"

Una claúsula sin un encabezado, tal como:

:- P1, P2, ... Pn.

es una directiva y es la forma por la cual un programa en PROLOG es invocado para executar las metas **P1, P2, ... Pn.** La directiva es interpretada como:

"Son P1 y P2 y ... Pn verdaderos?"

o como:

"Satisface la meta P1, después P2, después ... Pn"

Una forma especial de directiva, la cual es usada comunmente es la de interrogación, escrita como:

? P1, P2, ... Pn.

La diferencia entre estas directivas será explicada posteriormente, por ahora basta en saber la existencia de ambas. Antes de seguir con esta discusión de manera abstracta, veamos algunos ejemplos especificos de algunas claúsulas de PROLOG. Las siguientes son dos claúsulas de un procedimiento que define un predicado **estudia**, el cual asocia estudiantes y las materias que cursan:

```
estudia(jack,620).
estudia(jill,641).
```

La primer claúsula puede ser leída como "jack estudia la materia 620" y el segundo como "jill estudia la materia 641". A estas simples claúsulas podemos añadir una claúsula más general, de la forma:

```
estudia(Estudiante,611) :- año(Estudiante,1).
```

la cual puede ser ser leída como:

"La persona, estudiante, estudia la materia 611 si esa persona cursa el primer año"

o como:

"Para encontrar una persona que estudia la materia 611, encuentra quien esta en el primer año"

Estas claúsulas contienen ejemplos de los tipos de datos encontrados en PROLOG. Las claúsulas, sus metas componentes y los

argumentos de estas metas, llamados términos. En general un término es ya sea una constante, una variable o un término compuesto (una estructura).

Las Constantes son objetos definidos, tal como pronombres personales en lenguaje natural, y pueden ser ya sea enteros (611,1,620) o atoms (estudia,jack,:-).

Las variables son distinguibles por su primera letra siendo mayúscula (Estudiante) o por un caracter inicial de subrayado ("_") Las variables no representan localidades de almacenamiento como en la mayoría de los lenguajes de programación, sino que son nombres locales para algún objeto específico. Dos variables con el mismo nombre usadas en dos claúsulas diferentes son completamente diferentes.

Los términos compuestos son estructuras como **estudia(jack,620)** o como **año(Estudiante,1)**. Todas las estructuras en PROLOG son términos compuestos aunque existe una notación especial para el uso de las listas.

La operación básica que describe un programa escrito en PROLOG es la correspondencia o unificación de términos. Dos términos corresponden si sus funciones y todos sus argumentos corresponden o concuerdan. Además, si cualquiera de los argumentos son variables que no han sido instanciadas a un término, entonces estas

concordarán con cualquier argumento en la misma posición. Por ejemplo, **estudia(jack,620)** no correponderá a **estudia(jill,64)**, pero corresponderá a **estudia(X,620)**. Cuando aparece la correspondencia, la variable X es asignada a la constante **jack**.

Ahora que tenemos el vocabulario necesario para entender los términos más comunes usados en PROLOG, procedamos a analizar un ejemplo de un programa.

B. RECUPERACION DE INFORMACION DEDUCTIVA

Las tres claúsulas para **estudia** mencionadas anteriormente, son parte de un procedimiento que declara las materias estudiadas por estudiantes en una universidad. Este procedimiento, a su vez, es parte de un programa más largo que contiene información acerca de profesores enseñando varias materias, los cursos llevados por el estudiante y información referente a varias clases. En la realidad, este programa podría ser muy largo, pero abreviado luciría como:

```
enseña(jeff,611).
enseña(ken,620).
enseña(david,641).
enseña(jan,642).
enseña(ken,642).
enseña(graham,646).

estudia(fred,611).
estudia(jack,620).
estudia(jill,641).
estudia(jill,646).
estudia(henry,643).
```

```
estudia(henry,646).
estudia(X,611) :- año(X,1).

año(fred,1).
año(jack,2).
año(jill,3).
año(henry,3).

clase(611,m1000,1g1).
clase(611,w1300,1g1).
clase(646,tu1100,g24).
clase(622,th1000,418).
clase(643,tu1100,224).
```

En esta forma, el programa es esencialmente una base de datos, en la cual es importante notar que cada claúsula es completamente auto-contenida. Esto es, el programa no consiste en una secuencia de pasos o en un algoritmo como en todo lenguaje de programación convencional. Simplemente tenemos una colección de declaraciones de factores o relaciones lógicas y aún tendríamos un programa válido si las claúsula o todo el programa fueran intercambiados.

Pero, aún no sabemos como usar este programa. La manera más sencilla para hacerlo es haciendo una pregunta, tal como: Estudia jill la materia 641?, que en PROLOG sería:

```
? estudia(jill,641).
```

PROLOG responderá a esta pregunta escribiendo **true** si la meta dada es verdad dentro del contexto de este programa. Para encontrar si la meta es **true**, PROLOG tratará de relacionarla con la cabeza de la claúsula en el programa (buscando desde la parte superior del programa). Si se encuentra una correspondencia, la

claúsula correspondida es entonces activada al ejecutar, en turno, cada una de las metas en su cuerpo. La meta principal (cabeza) es verdadera si cada una de las metas en el cuerpo es también verdadera.

Así, en este ejemplo, PROLOG busca a través del procedimiento por la palabra **estudia** y encuentra que la meta dada corresponde una de las claúsulas en el procedimiento. Ya que el cuerpo de la claúsula está vacío, la meta es automáticamente satisfecha. Por lo tanto deberíamos recibir la respuesta:

true.

En esta forma, sólo estamos usando el programa para checar información acerca de objetos específicos. Una aplicación mucho más útil sería, por ejemplo, el encontrar todos los estudiantes en el año 3:

? año(Estudiante,3).

Aqui, **Estudiante** es una variable y PROLOG responde generando todas las instancias de la variable para la cual la meta es verdadera. Esto lo hace correspondiendo la meta con las claúsulas en el procedimiento **año**. Una variable sin instanciar corresponde con cualquier claúsula, por lo cual, la primer correspondencia será la claúsula **año(jill,3)**. PROLOG por lo tanto responde:

Estudiante = jill.

indicando que la meta es verdadera si **Estudiante** es instanciada a

jill. Si en cualquier momento PROLOG no encuentra una correspondencia para una meta, procede a efectuar encadenamiento regresivo.[5] Esto es, rechaza la claúsula activada más recientemente, deshaciendo cualquier substitución hecha por la correspondencia con la cabeza de la claúsula. Después reconsidera la meta original que activó la claúsula rechazada y trata de encontrar una claúsula subsecuente que también corresponda a la meta. Este mismo procedimiento también es aplicable cuando PROLOG ha encontrado con éxito una solución a la pregunta. Automáticamente retrocede a buscar soluciones alternativas. Al hacer esto, encuentra que la correspondencia siguiente (última) ocurre cuando:

Estudiante = henry.

Antes de ir más adelante, es necesario explicar la diferencia entre la pregunta que acabamos de hacer y la directiva:

:- año(Estudiante,3).

En ambos casos, estamos preguntando a PROLOG si **año(Estudiante,3)** es verdadero para cualquier **Estudiante** o, equivalentemente, encontrar un valor de **Estudiante** tal que **año(Estudiante,3)** sea verdadero. En el caso de la directiva, PROLOG encuentra un valor de **Estudiante**, si existe, pero no produce salida y no retrocede para encontrar valores alternos que quizás

---

[5] [5] Explicado en el capítulo no. 2

puedan satisfacer la meta. Si queremos imprimir el valor de **Estudiante**, necesitamos usar explícitamente un predicado aparte para hacerlo:

```
:- año(Estudiante,3),print(Estudiante).
```

Además, si queremos encontrar todos los valores posibles de **Estudiante**, tenemos que forzar el encadenamiento regresivo para que ocurra al hacer la meta fallar después de escribir el primer valor. PROLOG entonces tratará de re-satisfacer **año(Estudiante,3)**. Esto se puede hacer por medio del uso de otro predicado de PROLOG, llamado apropiadamente fail.

```
:- año(Estudiante,3),print(Estudiante),fail.
```

Cuando usamos un signo de interrogación al inicio de una directiva, estamos preguntando automáticamente a todos los valores variables que sean impresos y al encadenamiento regresivo que encuentre todas las soluciones posibles.

Ahora regresemos a nuestro programa y usemos preguntas un poco más elaboradas. Supongamos que queremos saber si **jill** y **henry** estudian materias en común. Preguntaremos de la siguiente manera:

```
? estudia(jill,X),estudia(henry,X).
```

PROLOG trata de satisfacer estas metas de izquierda a derecha, por lo cual encuentra una instancia de la variable X (por ejemplo, una materia) para la cual **estudia(jill,X)** es verdadera. La primer

correspondencia ocurre con X instanciada a **641**, por lo cual la primer meta es satisfecha y el valor **641** es substituido por la variable **X** a través de la claúsula. Después trataremos de satisfacer la segunda meta, la cual es

estudia(henry,641)

Obviamente esta claúsula no aparece en el programa, por lo cual la meta no se cumple. PROLOG entonces retrocede a la meta ejecutada más recientemente (en este caso sólo hay una meta precediendo) y busca por una correspondencia alterna. Esto es, deshace la instancia de **X** con **641** y busca una correspondencia alterna para **estudia(jill,X)**. La siguiente alternativa ocurre con **X** remplazada por **646** y la segunda meta es satisfecha cuando este valor es substituido por **X**. La meta completa es por lo tanto satisfecha y PROLOG presenta:

X = 646.

Al haber encontrado una solución, PROLOG nuevamente se encadena regresivamente para buscar las otras soluciones posibles. En este caso, no las hay, por lo cual la ejecución termina.

1. Deduciendo relaciones complejas.

Similarmente, podríamos seguir adelante y preguntar por todos los estudiantes que estudian una materia en particular, todas las materias impartidas por un maestro, etc. Sin embargo, no estamos

restringidos a hacer preguntas tan simples. PROLOG puede ser usado para deducir nuevas relaciones entre objetos. Por ejemplo, supongamos que queremos encontrar todos los estudiantes de un maestro. Podríamos hacerlo añadiendo al programa el predicado llamado **enseña** definido por:

```
enseña(Profesor,Estudiante) :-
     enseña(Profesor,Materia),
     estudia(Estudiante,Materia).
```

el cual implica que **Profesor** enseña **Estudiante** si **Profesor** enseña **Materia** y **Estudiante** estudia **Materia**. Entonces, si preguntamos:

```
? enseña(ken,Estudiante).
```

la respuesta será:

```
Estudiante = jack.
Estudiante = henry.
```

Similarmente, la pregunta:

```
? enseña(X,jill).
```

resultaría en:

```
X = david
X = graham
```

Es importante notar que en la primer pregunta, el primer argumento, **ken** fue la entrada y el segundo argumento presentó la salida. En la segunda pregunta, los roles de dos argumentos son invertidos. Podríamos ir aún más lejos al preguntar

```
? enseña(X,Y).
```

para encontrar todos los pares maestro-alumno.

Si estuvieramos usando esta base de datos con el fin de diseñar o modificar una tabla de horarios, quisieramos saber si habría conflictos, tal como el que un estudiante tomara dos clases al mismo tiempo, o que un salón de clases estuviera programado para dos clases a la vez. Para encontrar tales conflictos, podríamos añadir al programa un procedimiento como este:

```
conflicto(Salon,S1,S2,T) :-
                clase(S1,T,Salon),
                clase(S2,T,Salon),
                S1 /= S2.

conflicto(Estudiante,S1,S2,T) :-
                estudia(Estudiante,S1),
                clase(S1,T,_),
                estudia(Estudiante,S2),
                S1 /=S2,
                clase(S2,T,_).
```

En este procedimiento, S1 /=S2 es un predicado interno de PROLOG, el cual es verdadero si las variables S1 y S2 son instanciadas ya sea a enteros o atoms y las dos tienen valores diferentes. Un predicado escrito de esta forma es llamado un operador. Todos los comparadores usuales (=, /=, <, >, <=, >=) son disponibles como predicados internos en PROLOG.

La primera claúsula dice que un conflicto ocurre en el salón **Salon** a la hora **T** si la materia **S1** tiene lugar en **Salon** al mismo tiempo que dos materias que no son las mismas. La segunda claúsula dice que **Estudiante** tiene un conflicto si el estudia la materia **S1**, la cual tiene horario **T** y también estudia otra materia diferente, **S2** que tiene una clase al mismo tiempo. Notemos que

en este caso, no nos importa en cuáles salones ocurren estas clases, por lo cual PROLOG nos permite ahorrarnos nombres de variables al usar la variable anónima "_" como tercer argumento en las dos metas **clase**. Esta variable anónima difiere de todas las variables en que dos instancias de "_" dentro de la misma claúsula no comparten el mismo valor.

Hagamos la pregunta: "Hay algún conflicto entre salones o estudiantes?"

```
? conflicto(X,S1,S2,T).
```

resultaría en:

```
X = henry.
S1 = 643
S2 = 646
T = tu1100

X = henry
S1 = 646
S2 = 643
T = tu1100
```

lo que nos dice que un conflicto ocurrirá entre el estudiante **henry** el día martes a las 11 de la mañana (**tu1100**).

2. Uniendo términos por medio de listas

Hasta ahora, cuando hemos hecho preguntas como **? año(X,3)**, cada valor de **X** ha sido producido separadamente y no tenemos manera de manipular la colección de todas las instancias de **X** que cumplan con la meta. Por ejemplo, podemos pedir arreglar un grupo de

estudiantes en orden alfabético o contar cuántos estudiantes safisfacen cierta meta.

Una manera obvia de agrupar varias soluciones es por medio de listas. Una lista puede ser el atom "[]" representando la lista vacía[6]. Cada lista en PROLOG posee dos argumentos principales, la cabeza y la cola de la lista. Así, una lista conteniendo la primera letra del alfabeto sería representada por la estructura:

```
list(a,[]).
```

y una lista de los tres primeros elementos sería:

```
list(a,list(b,list(c,[]))).
```

Para simplificar el manejo de las listas, PROLOG provee una manera un tanto mejor a la de LISP. En esta notación, la estructura mostrada podría ser abreviada a:

```
[a,b,c]
```

y una lista cuya cola es una variable es escrita como:

```
[a,..L]  o [a,b,..L]
```

donde, en el segundo caso, **a** y **b** son los primeros dos elementos de la lista y **L** es la cola.

Ahora supongamos que queremos hacer una lista de todos los profesores de esta escuela o de todos los estudiantes que comparten alguna cualidad en particular. Nuestra implementación en PROLOG

---

[6] definida como **nil** en LISP

contiene un predicado llamado **findall**, el cual es definido para que la meta

findall(X,P,L).

construya una lista **L** constituida por todos los valores de la variable **X** para la cual la meta **P** es satisfecha. Los elementos son listados en el orden en que son encontrados. **findall** no es un predicado de primer orden en el sentido de que requiere que se hagan cambios en la base de datos durante la ejecución. Esto puede hacerse en PROLOG usando los predicados **assert** y **retract** [7] pero es fuera de la extensión de este estudio.

Así, para listar todos los estudiantes en el año 3, haríamos

:- findall(X,año(X,3),L),print(L).

y recibiríamos la respuesta:

[jill,henry].

Similarmente, para escribir una lista de todos los profesores,

:- findall(X,enseña(X,_),L),print(L).

y PROLOG respondería:

[jeff,ken,david,jan,ken,graham]

Ya que queremos una lista con todos los profesores, sin importar su materia, buscamos todos los valores de **X** para los cuales **enseña(X,_)** es satisfecha. Esto corresponderá con todas las

---

[7] Se recomienda ver ejemplos en Clocksin and Mellish, 1981

claúsulas para **enseña**, sin importar la materia en el segundo argumento. Es importante notar también que ya que **ken** enseña en dos materias, su nombre aparece dos veces en la lista.

C. RESOLVIENDO PROBLEMAS CON PROLOG.

Uno de los rompecabezas más comúnmente discutidos en programación es el problema de las "N reinas". La finalidad de este problema es encontrar todas las maneras de colocar N reinas sobre un tablero de ajedrez de N x N con tal de que ninguna reina atacará a otra, donde se dice que dos reinas se atacan si están posicionadas un una hilera en común, columna o diagonal.

Es posible resolver este problema usando lenguajes de programación convencionales, sin embargo, hay un esfuerzo considerable envuelto en determinar un procedimiento efectivo para generar configuraciones alternas para encontrar la solución y subsecuentemente retroceder para asegurarnos que todas las posibles soluciones fueron encontradas.

En contraste, ya que no necesita especificar la secuencia precisa de pasos requeridos para resolver un problema, sino que sólo definir las reglas a seguir y la meta a ser satisfecha, un programa en PROLOG para resolver este problema es casi trivial.

Antes de mirar al programa, consideremos la estructura de

datos que vamos a utilizar para representar las soluciones. Las dos posibles soluciones para un tablero de 4 x 4 son:

```
. Q . .        . . Q .
. . . Q        Q . . .
Q . . .        . . . Q
. . Q .        . Q . .
```

y claramente las podemos representar usando la lista:

[2,4,1,3] y **[3,1,4,2]**

donde el primer elemento da el número de la columna de la reina en el primer renglón, el segundo elemento da el número de la columna para el segundo renglón y así sucesivamente. Para cualquier valor de N, todas las soluciones posibles deben tener exactamente una reina en cada renglón y columna, por lo cual listas como las descritas son suficientes para describir todas las soluciones. Sin embargo, para que sea una solución válida, una configuración particular debe también satisfacer la condición de que dos reinas no residan sobre la misma diagonal (hacia arriba (/) o hacia abajo (\)).

Por lo tanto, como vamos generando la lista de número de columna representando una solución particular, necesitamos ser capaces de checar si cualesquiera dos elementos en la lista se atacan a sí mismos a lo largo de sus diagonales.

A lo largo de una diagonal hacia arriba, la diferencia entre números en renglón o columna es constante; a lo largo de una diagonal hacia abajo, la suma es constante. La tarea de checar

si una configuración de reinas en particular es válida, puede por lo tanto ser simplificada si remplazamos la lista de los números de columnas presentada anteriormente por una lista de términos de la forma:

```
square(C,U,D).
```

donde C representa la columna, U la diagonal hacia arriba y D la diagonal hacia abajo. Así, las soluciones sería representadas:

```
[square(2,-1,3), square(4,-2,6),square(1,2,4), square(3,1,7)]
```

y

```
[square(3,-2,4), square(1,1,3), square(4,-1,7), square(2,2,6)].
```

Veamos ahora como atacaríamos el problema haciéndolo manualmente para un tablero de N x N:

(1) Colocar una reina en algun lugar del primer renglón

(2) Ir al siguiente reglón y colocar una reina en una de las columnas que todavía este libre. Checar que esta posición no sea atacada a lo largo de cualquier diagonal y después repetir el paso (2) hasta que N reinas hayan sido colocadas en el tablero.

Si en cualquier momento colocamos una reina en un lugar inseguro, entonces retrocedemos a la última selección que hicimos y tratamos otra alternativa. Similarmente, cuando hemos encontrado una solución completa, retrocedemos a través de varias opciones que hemos hecho para generar todas las soluciones posibles.

Esto representa obviamente un procedimiento tedioso al ser seguido manualmente, particularmente para un tablero de 8 x 8, donde puede haber 92 soluciones. Este procedimiento es fácilmente seguido en PROLOG, donde el conjunto de reglas es fácilmente definido. El siguiente programa refleja esta simplicidad. De hecho, el programa es aún más simple que la discusión anterior, pues no se hace mención del encadenamiento regresivo, sino que procede automáticamente.

```
resuelve(Input,Output,[Renglón,..R],Columnas) :-
          escoge(Col,Columnas,C),
          Up is   Row - Col,
          Down is Row + Col,
          Seguro(Up,Down,Input),
          resuelve([square(Col,Up,Down),..Input],Output,R,C).
resuelve(L,L,[],[]).

escoge(X,[X,..Y],Y).
escoge(X,[H,..T1],[H,..T2]) :-
          escoge(X,T1,T2).

seguro(U1,D1,[square(_,U,D,),..Rest]) :-
          U1 /= U,
          D1 /= D,
          seguro(U1,D1,Rest).
seguro(_,_,[]).
```

Para un tablero de 4 x 4, por ejemplo, podríamos invocar al programa con:

```
? resuelve ([],Solución,[1,2,3,4],[1,2,3,4]).
```

Para seguir la forma en que trabaja el programa, iniciamos con el predicado **resuelve**. Esta meta tiene cuatro argumentos: una lista, **Input**, de cuadros que han sido ocupados, una lista, **Output**,

de cuadros ocupados en la solución final y dos listas que contienen el número de renglón y columna que aún estan libres. Inicialmente, **Input** es la lista vacía, **Output** es una variable sin instanciar y las otras dos listas contienen los enteros 1, ... N.

Para satisfacer la meta **resuelve**, PROLOG primero escoge una columna entre las disponibles al momento, y después de avanzar al siguiente renglón, calcula los parámetros **Up** y **Down** que caracterizan las diagonales. Después checa que esta posición sea segura, la inserta en la lista de posiciones ocupadas y llama a **resuelve** nuevamente para colocar la siguiente reina.

Cuando finalmente una reina ha sido colocada en cada uno de los **N** renglones, la lista **R** en la llamada recursiva a **resuelve** estará vacía. Ya que [] no corresponde [Row,..R], la llamada a **resuelve** no podrá encontrar la cabeza de la primera claúsula, pero, por el contrario, corresponderá con el segundo. La función de esta segunda claúsula es completar la solución al pasar su primer argumento, el cual ha estado acumulando la lista de cuadros seguros al segundo argumento, el cual muestra la solución final. Para el caso de un tablero de 4 x 4, este es:

```
Solución = [square(3,1,7),square(1,2,4),square(4,-2,6),
square(2,-1,3)]
```

La meta **escoge(Col,Columnas,C)** significa que el elemento **Col** es escogido de la lista **Columnas** (conteniendo el número de columnas

vacías), dando como resultado la lista reducida, C. La primera claúsula del procedimiento **escoge** selecciona la cabeza de la lista. Sin embargo, si la columna representada por este elemento no corresponde a una posición segura, entonces PROLOG retrocederá a **escoge** y la segunda claúsula será correspondida. Esta claúsula selecciona un elemento de la cola de la lista por la llamada recursiva a **escoge**.

La segunda y tercera meta en el cuerpo de **resuelve**, que calcula los valores de **Up** y **Down** usa el predicado interno **is**, además de dos operadores internos **+** y **-**, por lo cual es necesario una explicación breve de la forma en que PROLOG realiza procesamientos aritméticos.

Además de los operadores de comparación mencionados, PROLOG cuenta con operadores internos para la suma entera (+), resta (-), multiplicación (*), división (/), exponenciación (^) y modulo (mod), además con su usual precedencia y sus reglas de precedencia. Una expresión como **A * B** es una estructura simple en PROLOG, ***(A,B)**. Sin embargo los operadores aritméticos son llamados **funciones evaluables** por que PROLOG provee un mecanismo para evaluar expresiones ariméticas. Este mecanismo interno es **is**.

Después de esta breve descripción, dejamos la aritmética y volvemos al procedimiento final usado por **resuelve**, llamado **seguro**. La meta **seguro(Up,Down,Input)** significa que el cuadro caracterizado

por diagonales **Up** y **Down** es seguro si ningún cuadro en la lista **Input** reside en esas diagonales. La primera claúsula de **seguro** dice que los valores de **U1** y **D1** son seguros si el cuadro a la cabeza de la lista **Input** tiene diferentes valores de **U** y **D** y si **U1** y **D1** son también seguros de los elementos restantes de la lista. La segunda claúsula dice que **seguro** es verdadero para cualquier valor de **Up** y **Down** si la lista **Input** es vacía.

Esencialmente, esto completa la solución del problema de las N-reinas. Sin embargo, la entrada y salida de nuestro programa son un tanto inconvenientes, la entrada porque tenemos que teclear dos listas largas y la salida porque una lista como **Soln** es un tanto difícil de visualizar como una configuración en un tablero de ajedrez. Sin embargo, los procedimientos mostrados ofrecen una solución acertada y nos dejan apreciar las características que PROLOG ofrece.

Este programa quizás no sea más rápido en tiempo de ejecución que su programa equivalente escrito en PASCAL u otro lenguaje, pero la sencillez de su definición y la manera tan accesible de modificar las reglas, hacen de PROLOG un lenguaje muy poderoso para procesar Inteligencia Artificial.

D. ENCADENAMIENTO REGRESIVO Y CONTROL

Hasta ahora se ha mencionado que la característica de PROLOG de retroceder automáticamente (backtracking) provee la generación de soluciones múltiples. Antes de seguir adelante, examinemos este proceso con un poco más de detalle, pues es una de las cualidades principales de este lenguaje.

Los programas escritos en PROLOG generalmente implican éxito o falla en el sentido de que una condición se cumpla. Cuando una condición se cumple, el flujo del control sigue en la manera convencional, hacia adelante.

Por el contrario, cuando una condición falla, el control retrocederá en el intento de encontrar otras formas de alcanzar el éxito. Veamos un ejemplo de este proceso en acción. Supongamos que queremos determinar si hay dos elementos combinables en algún lugar de una lista. Usaremos la función mostrada, **append**, para especificar una relación que verifique esta condición, y después analizaremos el proceso de retroceso relacionado en satisfacer el procedimiento **elementos_multiples** para una lista dada.

Primero, mostremos el procedimiento **append**:

```
append([ ], P, P).
append([A|X], Y, [A|Z]) :- append(X,Y,Z).
```

Ahora observemos el siguiente principio: Si hay dos elementos idénticos en una list L, entonces:

```
? append(_,[E|M],L), append(_,[E|_],M).
```

es verdadero para algunos valores de E y M. Esto justifica el siguiente procedimiento:

```
elementos_multiples(E,L) :-   append(_,[E|M],L).
                              append(_,[E|_],M).
```

Básicamente, elementos_multiple dice que:

" Existe un valor para E, precedido por elementos arbitrarios, que es seguido por una lista que contiene una ocurrencia del mismo valor en la lista L."

Esto es, ocurrencias múltiples del mismo elemento existen en la lista. Para clarificar como funciona el proceso de retroceso, veamos una llamada a la función elementos_multiples:

```
? elementos_multiples(Elemento,[a,b,c,b]).
```

los cálculos para este query son presentados en forma esquemática:

```
1    <Goal>     elementos_multiples(Elemento,[a,b,c,b])
|
1    <Unify>    elementos_multiples(E,L)
|
1               Elemento=E,[a,b,c,b]=L
|
|2        <Goal>     append(_,[E|M],[a,b,c,b])
|2        <Unify>    append([],P,P)
|2                   E=a, M=[b,c,b]
|2        <Goal>     append(_,[a|_],[b,c,b])
|2        <Unify>    append([A|X],Y,[A|Z])
|
```

```
2                    A=b, X=_, Y=[a|_], Z=[c,b]
 3            <Goal>     append(_,[a|_],[c,b])
 3            <Unify>    append([A|X],Y,[A|Z])
 3                       A=c, X=_, Y=[a|_],Z=[b]
    4         <Goal>     append(_,[a|_],[b])
    4         <Unify>    append([A|X],Y,[A|Z])
    4                    A=b, X=_, Y=[a|_],Z=[]
      5       <Goal>     append(_,[a|_],[])
      5       <FAIL>     append(_,[a|_],[])
    4         <re-try>   append(_,[a|_],[b])
    4         <FAIL>     append(_,[a|_],[b])
 3            <re-try>   append(_,[a|_],[b])
 3            <FAIL>     append(_,[a|_],[c,b])
2         <re-try>   append(_,[a|_],[b,c,b])
2         <FAIL>     append(_,[a|_],[b,c,b])
2         <retry>    append(-,[E|M],[a,b,c,b])
2         <unify>    append([A|X],Y,[A|Z])
2                    A=a, X=_, Y=[E|M], Z=[b,c,b]
2         <Goal>     append(_,[E|M],[b,c,b])
2         <Unify>    append([],P,P)
2                    E=b, M=[c,b]
2         <Goal>     append(_,[b|_],[c,b])
2         <Unify>    append([A|X],Y,[A|Z])
2                    A=c, X=_, Y=[b|_], Z=[b]
 3            <Goal>     append(_,[b|_],[b])
 3            <Unify>    append([],P,P)
```

```
|2        <EXIT>     append(_,[b|_],[c,b])
|
1    <Exit>     elementos_multiples(b,[a,b,c,b])
```

El encadenamiento regresivo en este procedimiento se cumple porque el algoritmo básicamente adivina un elemento que podrá tener múltiples ocurrencias, después checa por alguna segunda ocurrencia de ese elemento. Si esa búsqueda falla, entonces debe re-intentar el proceso con un nuevo valor. El seguimiento mostrado de un retroceso relativamente sencillo, es suficiente para entender como funciona y los alcances que tiene.

Una ventaja importante de los lenguajes de programación denotacionales sobre los operacionales es que ponen su atención en la relación input/output en lugar de los procesos computados. Después de dar seguimiento a estos algoritmos, es esperada una comprensión del funcionamiento del encadenamiento regresivo.

E. PROLOG EN DISEÑO POR COMPUTADORAS (CAD)

En esta sección, será demostrado el uso de PROLOG en un muy pequeño sistema de diseño CAD. Aunque la discusión es acerca de una area en especial, las técnicas usadas resultan de interés y el entendimiento de estas puede ayudar a crear y comprender nuevos programas escritos en PROLOG, que es la finalidad de este estudio.

Queremos escribir un programa, el cual, al darle una secuencia lógica arbitraria tal como:

~a ^ b v c ^ ~d => e ^ f

donde los símbolos representan las operaciones **not(~)**, **and(^)**, **or(v)** e **implica(=>)**, entregará como salida, una secuencia de instrucciones para construir un circuito eléctrico equivalente usando únicamente compuertas **nand**, donde **nand(x,y)** es equivalente a **~(x ^ y)**.

El problema es dividido en dos partes. La primera es usar varias reglas lógicas para 'traducir' la expresión en otra diferente usando únicamente el operador **nand** y la segunda es 'codificar' esta expresión en una secuencia de instrucciones listas para ser usadas. Tomaremos como modelo la salida de los programas de "Wirit" de Hayes and Carrington[8]. Una simple expresión como

a v b

es transladada a

nand(nand(a,a), nand(b,b))

y codificada como

```
gate(1)   7400
          in(1)  a
          in(2)  a

gate(2)   7400
          in(1)  b
          in(2)  b

gate(3)   7400
```

[8] Vickers, 1981

```
            in(1)   gate(1)/out
            in(2)   gate(2)/out

final       gate(3)/out
```

El circuito consiste de tres compuertas, las cuales son contenidas en chips 7400 que son compuertas **nand**. Para cada compuerta, dos entradas deben ser especificadas. Así **gate(3)** recibe entradas de las salidas de las compuertas **gate(1) y gate(2)**. Además, el resultado final es especificado como la salida de la tercer compuerta. La fase de traducción se lleva a cabo de la siguiente manera:

```
:- op(700,xfx,=>).
:- op(600,yfx,v).
:- op(500,yfx,^).
:- op(400,fx,~).

traduce(X,Y) :-
          trans(X,A),
          termina(X,A,Y).

trans(~A,nand(B,B)) :- !, translate(A,B).
trans(nand(nand(B,B),nand(B,B)),B) :- !.
trans(A=>B, nand(C,nand(D,D))) :-!,
          traduce(A,C),
          traduce(B,D).
trans(A ^ B,nand(nand(C,D),nand(C,D))) :- !,
          traduce(A,C),
          traduce(B,D).
trans(A v B,nand(nand(C,D),nand(C,D))) :- !,
          traduce(A,C),
          traduce(B,D).
trans(nand(A,B),nand(C,D)) :- !,
          traduce(A,C),
          traduce(B,D).
trans(A,A).

termina(X,X,X) :- !.
termina(X,Y,Z) :- traduce(Y,Z).
```

Las cuatro primeras claúsulas en el programa son directivas

que definen las atoms '=>', v, ^ y ~ que sean operadores. Los argumentos asociados con la función **op** especifican las reglas de precedencia o asociatividad obedecidas por los operadores. Por ahora es suficiente decir que los operadores lógicos satisfacen todas las reglas lógicas usuales[9].

En realidad traducimos una expresión lógica **X** en una forma equivalente, **Y**, usando únicamente operaciones **nand** al satisfacer la meta **traduce(X,Y)**. El procedimiento **traduce** primero llama al procedimiento **trans** para transformar **X** en una expresión equivalente **A** y después llama un procedimiento **termina** para checar si **A** puede ser simplificada por una traducción posterior.

Cuando la meta **trans(X,A)** en el procedimiento principal es satisfecha, la expresión **A** será lógicamente equivalente a **X** e incluirá únicamente operaciones **nand**. Sin embargo, **A** quizás no este en su forma más simple. Por ejemplo, si preguntamos

```
? trans(a v a, X).
```

la respuesta será:

```
X = nand(nand(a,a), nand(a,a))
```

donde sabemos que esto puede ser simplificado considerablemente. En general, una expresión **A** se encuentra en su forma más simple si la meta **trans(A,Y)** es satisfecha con **Y = A**.

---

[9] Una discusión más a fondo de la definición de operadores lógicos puede ser encontrada en Clocksin and Mellish, 1981.

En nuestro programa, el procedimiento **termina** es usado para checar si una transformación ha ido lo más lejos posible. La meta **termina(X,A,Y)** en el procedimiento **traduce**, corresponderá con la cabeza de la primera claúsula en **termina** si **A = X**. En ese caso, sabemos que la traducción es completada, por lo cual **Y** es instanciada al valor de **A** y este valor es regresado como respuesta final. Por el otro lado, si **A** no es igual a **X**, entonces nuestra meta corresponde la cabeza de la segunda claúsula en **termina**. El cuerpo de esta claúsula llama a **traduce** otra vez, y este procedimiento iterativo continúa hasta que no hay más traducción posible.

Al final de la fase de traducción, la expresión lógica **a v b** habrá sido transformada en la estructura **nand(nand(a,a),nand(b,b))** como pedimos. La tarea de codificar esta traducción en una secuencia de instrucciones para construir el circuito, es realizada por los siguientes procedimientos:

```
wirit(nand(A,B), gate(M)/out,N,M) :-!,
      wirit(A,OutA,N,NA),
      wirit(B,OutB,NA,NB),
      M is NB + 1,
      print_nand(M,OutA,OutB).
wirit(X,X,N,N).

print_nand(GateNumber,Input1,Input2) :-
      print(gate(GateNumber),'7400'),
      print('in(1) ',Input1),
      print('in(2) ',Input2).
```

El primer argumento de **wirit** es la estructura producida por **traduce**. El segundo argumento representa la salida de la

compuerta. El tercer y cuarto argumento son usados para numerar las compuertas. N será el número usado para iniciar la secuencia de numeración y M será el último número en la secuencia.

Para seguir la forma de trabajar de **wirit**, se muestra un seguimiento de la ejecución de la meta:

```
wirit(nand(nand(a,a),nand(b,b),Final,0,M)
wirit(nand(nand(a,a),nand(b,b)),gate(M)/out,0,M)
     wirit(nand(a,a),gate(M)/out,0,M)
               wirit(a,a,0,0)
               wirit(a,a,0,0)
               M is 0 + 1
               print_nand(1,a,a)
     wirit(nand(b,b),gate(M)/out,1,M)
               wirit(b,b,1,1)
               wirit(b,b,1,1)
               M is 1 + 1
               print_nand(2,b,b)
     M is 2 + 1
     print_nand(3,gate(1)/out,gate(2)/out)
```

Cada línea en el seguimiento representa una llamada al procedimiento. La llamada final es al procedimiento **print_nand**, que acepta como entrada el número de la compuerta **nand** y las dos entradas. Si definimos el predicado **diseña** como

```
diseña(x) :-
     traduce(X,Y),
     wirit(Y,Final,0,_),nl,
     print('final  ',Final).
```

entonces la directiva

```
:- diseña(a v b).
```

resultará en la descripción dada anteriormente. Para aprender como construir un circuito representando nuestra "expresión lógica arbitraria", E, simplemente preguntamos:

```
:- diseña(~ a ^ b v c ^~ d => e ^ f).
```

y recibimos la respuesta

```
gate(1) 7400
     in(1)   a
     in(2)   a

gate(2) 7400
     in(1)   gate(1)/out
     in(2)   b

gate(3) 7400
     in(1)   d
     in(2)   d

gate(4) 7400
     in(1)   c
     in(2)   gate(3)/out

gate(5) 7400
     in(1)   gate(2)/out
     in(2)   gate(4)/out

gate(6) 7400
     in(1)   gate(5)/out
     in(2)   gate(6)/out

final   gate(7)/out
```

Si diseñamos cada una de las expresiones, encontramos que mientras la primera conduce a la configuración más simple de **nand**, la segunda forma requiere seis operaciones **nand**. Esto es consecuencia del hecho de que nuestro programa de traducción es muy simple y que no hemos incluido información tal como leyes distributivas para operaciones lógicas. Sin embargo, el propósito de este estudio es ilustrar qué se puede hacer con un simple programa, y no caer en grandes detalles.

F. CONCLUSIONES

Con la ayuda de ejemplos específicos, se han ilustrado muchas de las características del lenguaje de programación PROLOG. Sin embargo, en una discusión tan breve, hay muchas características que no han sido mencionadas, como la habilidad para definir y retractar claúsulas dentro de un programa en PROLOG, o bien, la variedad de procedimientos internos disponibles de entrada/salida. Estas características internas no son esenciales para entender la manera en que PROLOG trabaja, pero ayudan a incrementar el poder del lenguaje al ser usado en aplicaciones prácticas.

CAPITULO V. LISP vs PROLOG.

LISP vs PROLOG

Las funciones en matemáticas son definidas como subconjuntos de productos Cartesianos de conjuntos de dominio y rango[10]. Tal subconjunto determina una función sí y sólo si contiene a lo máximo un par ordenado para cada elemento del conjunto dominio. El conjunto dominino determina argumentos potenciales para la función, y el conjunto rango determina valores potenciales de la función. Cuando **(x,y)** es un par ordenado en una función, **y** denota el valor de la función correspondiendo al argumento **z**.

Por ejemplo, si el conjunto dominio contiene los elementos **d** y **e** y el conjunto rango contiene los elementos **r,s,** y **t,** entonces el conjunto **{(d,r),(d,s),(e,s)}** es una función, al igual que los conjuntos **{(d,r),(e,r)}** y **{(d,t),(e,t)}**, que son funciones constantes, pues entregan el mismo valor para cada argumento.

Por otro lado, el conjunto **{(d,r),(d,s),(e,r)}** no es una función porque contiene dos pares ordenados diferentes con el mismo dominio, **(d,r)** y **(d,s)**, pero sin embargo, es una relación.

---

[10] El producto Cartesiano del conjunto dominio y rango es el conjunto de todos los pares ordenados en el cual el primer componente viene del conjunto dominio y el segundo del rango.

Los programas funcionales discriminan cuidadosamente las entradas computacionales (argumentos o funciones, correspondiendo a elementos del dominio de la función en la formulación matemática) y las salidas (valores de funciones, elementos del rango). Por otro lado, los programas relacionales no designan componentes particulares como componentes de entrada y otros como componentes de salida, sino que cada componente puede tomar cualquier rol, dependiendo de la naturaleza de la invocación.

La función **reverse**, que invierte el orden de los elementos de las listas, servirá ampliamente como ejemplo. Como función, en el sentido matemático, **reverse** tiene el conjunto de secuencias finitas como su dominio y el mismo conjunto como su rango. Esto la hace un subconjunto del producto Cartesiano del conjunto de secuencias finitas dentro de sí misma. Este mismo subconjunto también forma una relación en el sentido matemático.

Como función, en el sentido computacional, expresada en LISP, **reverse** entrega una nueva secuencia como valor sólo si una invocación ofrece una secuencia como argumento.

Función definida en LISP	Invocaciones	Resultados

`=>(defun rev(x)`
`  (if (null x)`
`  x`
`  (append (rev (cdr x)`
`  (list(car x)))))`
	`(rev '(a b c))`
	`(rev '(c b a))`
		`(c b a)`
		`(a b c)`

Por el otro lado, como una relación computacional expresada en PROLOG, una invocación a **reverse** puede ofrecer un valor de entrada en cualquier componente y el resultado entregará un valor para el otro componente que satisfaga la relación **reverse**. O bien, una invocación puede ofrecer valores de entrada en ambos componentes, en cuyo caso el resultado indicará si el par ordenado indicado satisface la relación **reverse**. En esta forma, PROLOG cumple la noción matemática de una relación al no distinguir entre componentes de entrada y componentes de salida como pares ordenados.

Relación definida en Prolog	Invocaciones	Resultados
`rev([].[]).`	`rev([a,b,c],Y).`	`Yes. Y=[c,b,a]`
`rev([W3X],Y) :-`		
`  rev(X,Z),`	`rev(X,[c,b,a]).`	`Yes. X=[a,b,c]`
`  append(Z,[W],Y)`	`rev([a,b,c],[c,b,a])`	`Yes.`

La forma de tratar input/output en los componentes, señala una diferencia fundamental entre programas escritos en LISP y PROLOG. La flexibilidad de PROLOG en esta área proviene del potencial para computaciones extras en el proceso de resolución, que provee la base para interpretar programas en PROLOG, comparado con el proceso de substitución que forma la base para computaciones en sistemas basados en LISP.

PROLOG también tiene otra ventaja sobre LISP al ofrecer

reconocimiento automático de patrones entre invocaciones y definiciones. La definición de una relación en PROLOG toma la forma de una colección de factores y reglas, cada una involucrando un patrón diferente de valores componentes. Para interpretar un query, el sistema PROLOG usa el factor o la regla cuyo componente corresponde al mismo del query. Por ejemplo, para interpretar el query **rev([],[])**, PROLOG usaría el primer factor en la definición mostrada de la relación reverse porque los componentes en el query corresponden a los del factor.

Por el otro lado, PROLOG no podría usar el factor para contestar el query **rev([a,b,c],[c,b,a])** porque los componentes en el query, siendo listas no-vacías, no pueden corresponder a estos del factor. Sin embargo, estos componentes corresponden con los componentes de la siguiente, y PROLOG usa esa regla para responder al query.

Los lenguajes funcionales modernos incorporan características de reconocimiento de patrones similares. Esto hace posible el definir funciones en términos de listas involucrando ciertos argumentos sin pruebas **if-then** encadenadas para los casos en el cuerpo de la función.

Ya que la correspondencia de patrones es incompatible con la programación funcional, la vemos aquí como una conveniencia de notación de PROLOG, pero no como una diferencia fundamental entre

programación relacional y funcional. Sin embargo, la simetría de input/output de PROLOG en diferencia del uso de argumentos en LISP, sólo es una diferencia sintáctica.

Por el otro lado, LISP y la notaciones funcionales en general, tienen algunas ventajas sobre PROLOG. Una de ellas es la capacidad para combinar funciones que hagan expresiones más complejas, y otra la capacidad de funciones de alto orden.

Por ejemplo, una función merge-sort involucra una combinación funciones de dividir y unir. Un programa en LISP expresa esto por medio de funciones de composición, pero PROLOG carece de este concepto y debe designar variables intermedias para comunicar resultados parciales entre estados. El siguiente programa en PROLOG usa las variables intermedias **A,B,SA y SB**, mientras que el programa en LISP usa una variable intermedia, **pair**, correspondiendo a **A** y **B** del programa en PROLOG, para evitar el re-escribir la expresión que parte la secuencia de entrada en dos partes, pero el programa en LISP combina funciones de unir y sortear en una expresión, sin tener que usar variables intermedias **SA** y **SB** para llevar a cabo subsecuencias sorteadas al punto de unión.

Función en LISP

```
(defun sort(seq)
     (if (null seq)
          '()
          (let ((pair (split seq)))
            (merge  (sort (car pair))
                    (sort (cadr pair)))
```

Relación en PROLOG

```
sort([],[]).
sort(Seq,Srt) :- split(Seq,A,B),
                 sort(A,SA),
                 sort(B,SB),
                 merge(SA,SB,Srt).
```

```
)))
```

Las funciones de alto nivel de LISP, le dan otra ventaja sobre PROLOG. La recursión no puede ser eludida en un sentido general, pero una función que utiliza funciones de alto nivel como argumento, puede ser comprendida con mayor facilidad por no-programadores, por lo cual los programas pueden suprimir las porciones recursivas al invocar estas funciones.

Por ejemplo, la función de LISP **mapcar** aplica una función dada a cada elemento de una lista y entrega esa lista como resultado. El patrón común de recursión es el construir una lista de resultado cuyo primer elemento es el valor de una función dada aplicada al primer elemento de una lista dada y el resto de cuyos elementos son aquellos entregados por la misma fórmula aplicada al resto de los componentes de una lista dada.

```
=> (defun mapcar (f seq)
     (if (null seq)
          '()
          (cons (f (car seq)) (mapcar f (cdr seq)))
     ))
```

Un buen programador de lenguajes funcionales decidirá qué patrones comunes de recursión cada cómputo requiere, codificar los de funciones de alto nivel, y usar las funciones de alto nivel en lugar de repetir las ecuaciones recursivas. Esta forma de expresar cómputos, hace posible para la gente que quiere entender el programa, el invertir el tiempo requerido para entender un patrón recursivo sólo una vez, y después hacer uso de ese tiempo

en varios lugares dentro del programa. Esta economía debe resultar familiar a cualquier programador.

Hasta ahora se han mencionado dos ventajas de PROLOG sobre LISP (reconocimiento de patrones y trato simétrico de input/output) y dos ventajas de LISP sobre PROLOG (composición de funciones y funciones de alto nivel).

Ya que PROLOG usa resolución en lugar de substitución como base para cálculos, los programadores de PROLOG pierden algo del control que los programadores en LISP tienen sobre el proceso de interpretar el programa. Esto es, los programa en LISP tienden a ser más específicos, mientras que en PROLOG dejan más opciones abiertas al intérprete. Un intérprete astuto será capaz de tomar ventaja de estas opciones y producir programas más económicos.

Las ventajas y desventajas mencionadas sobre programación funcional o relacional dejan ver las características de ambas, pero aún es difícil decidir cual sistema presenta mayores ventajas en general. La historia favorece a los lenguajes que son menos específicos como los más rápidos, así como los más poderosos en poder de expresión. Ya que ambos, LISP y PROLOG tienen ventajas de este tipo de las que otros lenguajes carecen, parece rasonable adivinar que alguna combinación de ideas de ambos lenguajes será lo más acertado para los lenguajes denotacionales del futuro.

VI. CONCLUSIONES

PROLOG no es el único ni mejor lenguaje de desarrollo de Inteligencia Artificial, de hecho, hasta fechas recientes, LISP había sido el estándard. Desde el inicio de esta rama, los investigadores de Inteligencia Artificial han usado LISP para procesamientos simbólicos como lenguaje de procesamiento natural.

Sin embargo, al ser desarrollado PROLOG, y especialmente despues de que los científicos japoneses lo seleccionaron como el lenguaje para su proyecto de la quinta generación, algunos investigadores empezaron a dudar del dominio de LISP.

LISP fue aceptado por la comunidad de Inteligencia Artificial por su gran utilidad. LISP exede a los lenguajes convencionales en el manejo de símbolos, donde listas de símbolos son evaluadas con sencillez.

Con LISP, el programador es capaz de escribir programas que usan símbolos en combinación con listas y funciones definidas, las que a su vez pueden ser relacionadas con otras listas. Esta habilidad es muy útil en el procesamiento de lenguajes naturales (como inglés o español).

Por el contrario de LISP, un programa en PROLOG consiste en

una serie de oraciones en inglés escritas en forma lógica. Ambos, programa y datos estan unidos por factores y reglas, lo que hace una simple tarea el formar relaciones entre palabras e incorporar estas relaciones en el programa. En LISP, por el otro lado, una función deberá ser definida antes de que estas relaciones puedan efectuarse.

Este estudio ha mostrado varios tópicos relacionados con Inteligencia Artificial, así como un sondeo de los lenguajes de programación LISP y PROLOG. Ambos tiene sus cualidades, ambos superan a los lenguajes convencionales en el procesamiento simbólico, a la vez que uno supera al otro en áreas especificas.

La controversia continuará mientras que las investigaciones en Inteligencia Artificial continúen progresando, y ambos mejorarán sus formas de trabajo. La última opción para un lenguaje de Inteligencia Artificial la tendrá el lector después de haber comprendido este estudio, esta opción reflejará sus gustos y preferencias hacia diferentes lenguajes de programación.

Se espera que este estudio haya cumplido sus metas, el dejar conocer al lector las bases y aplicaciones de la Inteligencia Artificial, de LISP y PROLOG, y que el mismo haya gozado al leer estas páginas.

VII. GLOSARIO TECNICO

Argumento Un valor que es pasado a una función

Algoritmo El método general o plan para alcanzar el resultado deseado en una función

Atom Una cadena de caracteres, números o signos de puntuación

Busqueda Procesar una estructura de datos con el fin de encontrar uno o más elementos de un tipo particular

Ciclo Una estructura en la cual las operaciones son realizadas repetitivamente, hasta que alguna condición de salida se cumple

Ciclo Una acción en un sistema de producciones que consiste en corresponder producciones al seleccionar una instancia

Claúsula Una aserción que existe en una base de datos

Contador Una variable que es usada para contar el número de veces que se ejecuta un ciclo

Correspondencia de patrones Selección de un elemento de acuerdo a la similaridad con otro

Ejecución en paralelo Acciónes que ocurren simultáneamente

Elemento Un atom o lista que es miembro de una lista

Encadenamiento regresivo Uno de los varios métodos usados en el procesador de deducciones para seleccionar y operar en componentes de la base de conocimientos

Expresión Un atom o una lista

Función iterativa Una función que realiza acciones repetidas en un ciclo

Función Procedimientos que operan en argumentos para producir un valor esperado

Función recursiva Una función que se llama a sí misma

Instancia Una representación de una manera por la cual una patrón puede corresponder en una base de datos

Inteligencia Artificial Un área de las ciencias de la computación que tiene como primer fin el modelamiento del comportamiento humano por sistemas computacionales

Intérprete de reglas Un programa que corresponde producciones en una base de datos

Lenguaje de alto nivel Un lenguaje donde el diseño e implementación es provisto automáticamente por el compilador, en lugar de ser provistos manualmente por el programador

Lenguaje natural Un área de la investigación en Inteligencia Artificial que se concentra en permitir a las computadoras el interpretar lenguaje natural, como inglés o español

Lista Una secuencia de elementos encerrados entre paréntesis

Lista vacía Una lista sin elementos, conocida como nil

Lógica algorítmica Lógica basada en fórmulas, donde el orden de las operaciones puede ser predecido por adelantado

Memoria de trabajo La base de datos en un sistema de producciones

Nil Denota una lista con cero elementos o lista vacía

Parametros de una función Las variables en la definición de una función que toman los argumentos como valores cuando la nueva función es llamada

Patrón Un conjunto de una o más claúsulas constituidas de variables and/or

Predicado Una función que realiza una prueba y regresa un resultado que puede ser interpretado como verdadero o falso

Procedural Reglas que representan control directo sobre una secuencia de operaciones en un programa de computación

Procesador de deducciones El componente de los sistemas basados en conocimientos que provee el razonamiento inicial y controla estrategias para operar el sistema

Producción Una regla if-then en un sistema de producciones

Prototipo Un programa que no ha sido completado para ser considerado operacional al cien por ciento, pero es suficientemente operacional que su operación puede ser demostrada

Recursión Posibilidad de una función de llamarse o usarse a sí misma. Las funciones recursivas son particularmente útiles para busquedas o manipulación de estructuras

Regla IF-THEN Una regla que consiste en una patron que tiene la forma: **si** prueba es satisfecha, **entonces** realiza la acción especificada

Reglas de producción Una expresión de una relación condicional,

generalmente de la forma **IF-THEN**

Reglas de decisión Uno de los varios términos para describir los procedimientos incluidos en varios sistemas expertos

Representación de conocimiento Los métodos y técnicas usadas para modelar y codificar factores y relaciones en una base de conocimientos

Sistema basado en reglas Un sistema en el cual el conocimiento es representado en forma de reglas

Sistema de diagnósticos Un sistema que puede determinar la causa o falla de problemas al ser alimentado con observaciones y resultados de pruebas

Sistema de producciones Un sistema constituido por una base de datos, un conjunto de reglas y un intérprete de reglas que aplica estas reglas a la base de datos

Sistema experto Un término usado para describir un programa de computadora que podría remplazar algún aspecto del desarrollo normal del experto humano

Sistema operativo Un programa de computadora que provee las

operaciones básicas de computación, tal como archivar, imprimir, leer entre otras

Sort Organizar un grupo de elementos conforme a alguna dimensión

Sub-función Una función que realiza parte de las acciónes de una función de mayor orden

Usuario Una persona interactuando con un programa

Variable global Una variable a la cual se le asigna su valor por medio de **setq** fuera de cualquier definición de función y retiene su valor hasta que es renovado

Variable local Una variable que retiene su valor sólo durante la ejecución de una función particular

Variable Un atom al cual se le ha asignado un valor

BIBLIOGRAFIA

ANDERSON, JOHN, CORBETT, ALBERT: **Essential LISP**, Addison-Wesley Publishing Company, Inc.,1987.

ANDRIOLE STEPHEN J.: **Applications in Artificial Intelligence**, Petrocelli Books, Inc.,1985.

BANERJI, RANAN B.: **Artificial Intelligence, a Theoretical approach**, Elsevier North Holland, Inc.,1980.

BARKOVSKY, ALVIN: **Lisp versus Prolog**, Computers and Electronics, v23, Jan 1985 p71.

CHOFARAS, DIMITRIS: **4th and 5th Generation Programming Languages**, Mc Graw Hill, 1986.

CLOCKSIN W.F., MELLISH, C.S.: **Programming in Prolog**, Springer-Verlag,1981.

COLMERAUER, ALAIN: **Opening the Prolog III Universe**, Byte v12, Aug 1987, p177.

D'AMBROSIO, BRUCE : **Expert systems - myth or reality? Artificial intelligence is applied**, Byte v10, Jan 1985 p275

HAUGELAND, JOHN: **Artificial Intelligence: The very idea**, Massachussetts Institute of Technology, 1985.

HAYES, J.E.: **Intelligent Systems, the Unprecedent Opportunity**, Ellis Horwood Limited, 1983.

HIGHBERGER, D., EDSON, D.: **Intelligent computing era takes off**, Computer design, September 1984.

HINSHAW DOROTHY: **The quest for Artificial Intelligence**, Harrout Brace Jovanovich publishers, 1986.

JACKSON, PHILIP C.: **Introduction to Artificial Intelligence,** Petrocelli books, 1974.

KARAMJIT S. GILL: **Artificial Intelligence for Society,** John Wiley & Sons, Great Britain, 1986.

MICHIE, DONALD: **On machine Intelligence,** Ellis Horwood Limited, 1986.

MICHIE, DONALD: **Machine Intelligence and related topics,** Gordon and Breach Science Publishers, 1982.

RANDALL DAVIS, LENAT DOUGLAS: **Knowledge-based Systems in Artificial Intelligence,** Mc. Graw Hill Inc.,1982.

SEYMOUR SCHOEN, SYKES, WENDELL: **Putting Artificial Intelligence to work,** John Wiley & Sons, Inc., 1987.

SIMONS, GEOFF L.: **Are computers alive?,** The thetford Press Ltd, Thetford, Norfolk, 1983.

VERITY, J. W.: **The LISP race heats up,** Datamation, August 1986.

VERITY, JOHN: **Prolog vs. Lisp; the two artificial intelligence programming language are squaring off in what wome say is a "religious battle",** Datamation, v30, Jan 1984, p50

VICKERS, T.: **A Connection Descriptor and Query System,** Computer Science Honours Thesis, University of New South Wales, 1981.

WELNER, JAMES: **Logic Programming and Prolog; the language that Japan has chosen for its Fifth Generation Project is based on mathematical logic,** Computers and Electronics v23, Jan 1985, p68