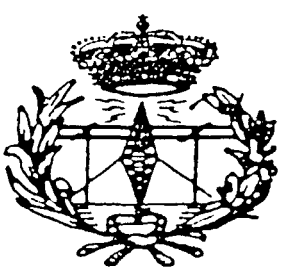
Introducción a la Inteligencia Artificial, estudio lenguaje Prolog.

Universidad Politécnica de Canarias



Escuela Universitaria de Telecomunicaciones

Tutor.: Don Roberto Rodríguez Domínguez

Alumno: Fernando Pérez de Blas

Indice general

Introducción del autor 0.0

INTRODUCCION A LA IA.

Introducción.....1-1

Concepto de la IA. 1-2

Proceso de datos en IA.....1-5

Algorítmica y Heurística. 1-7

Diseño de ordenadores para IA. 1-9

Factores comunes a todos los tipos.....1-10

Sistemas para desarrollar aplicaciones.....1-11

Computers para desarrollo de aplicaciones.....1-12

Grandes ordenadores con multiproceso.....1-13

Miniordenadores.....1-13

"Workstations" de uso general.....1-14

Ordenadores para I+D.....1-15

Superordenadores para I+D.....1-16

Workstations en I+D: máquinas LISP.....1-16

Evolución histórica de la IA.....1-17

Situación actual.....1-21

Perspectivas futuras.....1-25

El Método.....1-29

EXPLORACION DE ALTERNATIVAS.

Componentes de un sistema de búsqueda.....2-37

Representación en árbol.....2-38

Razonamiento hacia delante y hacia atrás.....2-39

Estructura de los Arboles And/Or. 2-40

Explosión Combinatoria. 2-46

Estrategias de Control. 2-48

Determinación de un camino. 2-50

Búsqueda del Camino Optimo. 2-59

Búsqueda en árboles And / Or. 2-68

SISTEMAS DE PRODUCCION.

Introducción al cálculo de Predicados....3-74

Fórmulas Atómicas. 3-76

Conectores. 3-79

Cuantificadores....3-80

Fórmulas bien formadas. 3-82

Reglas de Inferencia. 3-84

Representación mediante cláusulas. 3-86

Cláusulas de Horn. 3-87

Unificación y Sustitución. 3-89

Resolución por Refutación. 3-91

Programación en Lógica. 3-91

Resolución declarativa de problemas. 3-92

Ventajas e inconvenientes de la Lógica.....3-95

LENGUAJE PROLOG PARTE 1.

El Fundamento Lógico del Prolog.....4-97

Introducción 4-97

La Programación Lógica.....4-98

El Principio de Resolución.....4-101

Las Cláusulas de Horn.....4-102

El Prolog.....4-103

La Máquina Prolog. 4-106

LENGUAJE PROLOG PARTE 2.

Los predicados.....5-110

Predicados Creados.....5-110

Predicados Instalados 5-111

Predicados Principales y Secundarios.....5-112

Los Hechos y La Base de Hechos 5-113

El diálogo 5-116

Las variables 5-120

Definición de las variables 5-120

Propiedades de las Variables.....5-121

Las Conjunciones y los Modificadores 5-124

AND 5-124

OR.....5-125

NOT.....5-126

IDÉNTICO 5-127

LAS REGLAS 5-127

Definición.....5-127

LENGUAJE PROLOG PARTE 3.

Observaciones sobre las reglas. 6-131

El encadenamiento hacia atrás 6-132

El encadenamiento hacia adelante 6-132

Los Caracteres 6-133

Las constantes.....6-134

Los Atomos 6-134

Los Enteros 6-136

Seguimiento o traza de un programa 6-137

TRACE 6-138

CALL 6-138

RETURN o EXIT 6-138

REDO 6-139

FAIL 6-139

NOTRACE 6-139

Ayuda a la depuración de un programa 6-139

pp 6-140

listing 6-140

Ejemplo de traza de un programa 6-144

Los elementos compuestos 6-145

Arboles 6-147

Las Listas. 6-152

Los Operadores **6-156**

La Posición **6-157**

La Prioridad **6-158**

La Asociatividad **6-159**

Las comparaciones y las operaciones aritméticas **6-159**

Igualdad 6-160

No Igualdad 6-161

Comparaciones de mayor y menor 6-161

Las Operaciones Aritméticas 6-163

La Recursión **6-164**

La manipulación de listas **6-168**

La relación de pertenencia 6-168

Concatenar 6-170

El último elemento de una lista 6-172

Dos elementos consecutivos 6-173

Invertir una lista 6-174

Borrar un elemento de una lista 6-175

Sustituir un elemento de una lista 6-175

Sublistas 6-176

Correspondencia entre listas 6-177

Conjuntos 6-177

Conceptos 6-177

Relación de pertenencia 6-179

Operaciones con conjuntos 6-180

Las bases de datos 6-182

Predicados que manipulan la base de datos 6-183

Otros predicados 6-187

El Control 6-190

La Marcha Atrás 6-191

LENGUAJE PROLOG PARTE 4.

El Corte **7-196**

Las Metarreglas o Maxirreglas **7-199**

Las Paradas **7-200**

Wait 7-200

Stop 7-200

Exit 7-201

Aclaraciones sobre la marcha atrás **7-201**

Predicados instalados **7-202**

Predicados que manipulan las bases de datos 7-204

Predicados que influyen en el control **7-208**

TRUE 7-208

FAIL 7-209

Otros Predicados 7-209

Predicados que influyen en la marcha atrás 7-210

Predicados de tipificación 7-211

Impresión y lectura de términos 7-212

Impresión y lectura de caracteres 7-214

Operaciones de E/S con los archivos 7-216

Predicados para utilizar la pantalla 7-220

Predicados que manipulan cláusulas 7-220

INTRODUCCION A LOS SISTEMAS EXPERTOS

Componentes de un Sistema Experto 8.222

Base de conocimiento 8.223

Motor de inferencia 8.224

Interfaz de usuario 8.225

Características de un Sistema Experto 8.226

Desarrollo de un Sistema Experto 8.227

Identificación 8.228

Conceptualización **8.230**

Formalización **8.231**

Automat. en la creac. de la b.c. **8.233**

Implementación **8.236**

Prueba **8.237**

Comentario sobre el programa ejemplo **8.238**

APENDICE I: Programa ejemplo

APENDICE II: Bibliografía.

1. INTRODUCCION A LA IA.

1.1. Introducción.

La Inteligencia Artificial (IA) es una ciencia y una tecnología que se ocupa de la comprensión de la inteligencia y del diseño de máquinas inteligentes, entendiendo por tales, aquellas que presentan características asociadas al entendimiento humano, como el raciocinio, la comprensión del lenguaje hablado y escrito, el aprendizaje, la toma de decisiones y otras similares.

La IA es un campo científico reciente con poco menos de 30 años de historia. Su nacimiento data de 1956, pero hasta el comienzo de la presente década no ha cobrado verdadero auge.

Actualmente las expectativas son enormemente prometedoras, con un número creciente de aplicaciones prácticas que se están comercializando o se van a comercializar en breve plazo.

1.2. Concepto de la IA.

La IA, como tecnología que trata de producir sistemas inteligentes, se engloba dentro de la ciencia e ingeniería informática, aunque los métodos que utiliza no tienen que emular necesariamente el comportamiento de la mente humana.

Aunque la IA se basa en programas de ordenador, al igual que lo que se podría denominar informática tradicional o convencional, los procedimientos de resolución de problemas y las técnicas de programación de IA presentan una serie de características que los diferencia de los de la informática convencional.

Así encontramos, en primer lugar que, mientras en la informática tradicional los objetos que manejan las máquinas son datos, de tipo numérico o alfa-numérico, en IA, **los objetos son ideas y conocimientos** y se ajustan más a una descripción de tipo simbólico.

En la informática convencional los procesos a los que se someten los datos son de tipo **algorítmico,** perfectamente definidos y estructurados a priori, con una secuencia de operaciones predefinida y que se repite en su totalidad ante las mismas condiciones de partida.

En IA, por el contrario, el proceso es de tipo **lógico abierto**, en el cual, el ordenador dispone de unas **reglas de inferencia** y de una **base de conocimientos** y en función de ambas y de la información que adquiere, o se le suministra, inicia un proceso de búsqueda empírico o "*heurístico*" con notables dosis de complejidad, incertidumbre y ambigüedad,

propias, por otra parte, de los problemas de decisión con que se enfrentan las personas en el mundo real.

Así como en el cálculo científico se requieren respuestas exactas y se buscan las mejores soluciones posibles, en IA, los programas no siempren garantizan respuestas correctas y se aceptan normalmente respuestas "satisfactorias", tal como sucede con los problemas humanos en los que incluso ocurre a veces que las decisiones que se toman son equivocadas.

Un aspecto importante de los programas de IA es el del área o dominio del conocimiento en que se van a presentar los problemas para su resolución.

El dominio de conocimiento debe estar claramente acotado y estructurado para poder ser manejado. Por ello en IA es usual tener por separado los conocimientos y el mecanismo que controla la búsqueda de soluciones (al contrario de lo que ocurre con los programas convencionales).

Como consecuencia los programas de IA son, normalmente, fáciles de modificar, actualizar y ampliar, ya que se puede cambiar de dominio cambiando la base de conocimientos y utilizar el mismo método de búsqueda.

Debido a esta organización y al hecho de que el sistema informático puede inferir nuevas reglas, desaparece en buena parte, en el caso de IA, el **determinismo algorítmico** que caracteriza a la programación convencional.

En IA el programa indica al ordenador lo que debe hacer sin especificarle cómo debe hacerlo. En las diferencias que se establecen entre el "**cómo**" y el "**por qué**" de llegar a determinadas conclusiones están, según **R. Davis** del **MIT**, las características que separan a ambas formas de programación.

La **representación del conocimiento**, es por tanto, un punto clave en los problemas de IA y es una de las líneas de investigación más activas hoy en día en IA. Cuando se quiere resolver un problema complejo, los conocimientos que se requieren manejar son numerosos y por ello se necesitan métodos eficientes para representarlos y manejarlos.

El otro punto clave en los problemas de IA es el del proceso de **búsqueda de soluciones**. La resolución de un problema supone frecuentemente una serie de elecciones entre varias alternativas hasta llegar a la meta final.

Este método se representa mediante una estructura jerárquica de árbol y la solución sigue un camino desde el nodo de origen hasta el nodo de destino o meta final. Para problemas de gran complejidad, resulta demasiado complicado explicitar todos los posibles caminos, para determinar el que conduce a la mejor solución. Es preciso disponer de métodos de búsqueda eficientes, que, reduzcan el abanico de posibilidades, empleando reglas empíricas o razonamientos, que no consideren todas las ramas del árbol sino solamente las más probables.

Como consecuencia el camino seguido y la solución encontrada pueden no ser los óptimos, pero sí razonablemente aceptables con la ventaja de reducir enormemente el proceso de búsqueda. De esta forma los nuevos sistemas se aproximan más a la forma de comportarse de las personas que, en su proceder diario o en sus procesos de decisión, lejos de buscar respuestas precisas y soluciones exactas manejan reglas aproximadas y hechos imprecisos para alcanzar conclusiones útiles, y aceptables.

1.3. Proceso de datos en IA.

En el cálculo convencional, un programador crea un programa, formado por instrucciones para el ordenador, que sigue un camino de solución para cada situación; este camino está completamente planeado por el programador. Cualquier sorpresa en el transcurso del proceso es un error que debe ser eliminado. Este método convencional de usar pasos predecibles puede ser muy potente, ya que capacita para resolver problemas que requieren el procesamiento de una gran cantidad de datos y la repetición de muchos pasos . A lo largo de los años se han hecho progresos en el cálculo convencional.

- **Primero:** se pueden ejecutar muy rápidamente con el hardware disponible.
- **Segundo:** se han reducido el tiempo y los esfuerzos requeridos para escribir los programas, por el desarrollo de conjuntos de instrucciones y lenguajes de programación que requieren escribir pocas líneas de código por parte del programador.

Sin embargo, muchos problemas importantes no se resuelven fácilmente por el cálculo convencional, aunque la velocidad de ejecución sea muy rápida. Estos problemas incluyen decisiones basadas en la interacción compleja de muchos factores que deben ser considerados como un todo, más que como una serie de pasos. Pero la IA utiliza una técnica que es útil en la solución de estos problemas complejos: el **procesamiento simbólico**.

Los símbolos procesados por los programas de IA a menudo representan entidades del mundo real, y en lugar de efectuar simplemente cálculos, los programas de IA manipulan o "*piensan*" sobre las relaciones entre los símbolos.

En el procesamiento de datos tradicional, el sistema procesa el contenido de variables. Los sistemas de IA pueden hacer esto, pero pueden también manipular símbolos independientemente de sus valores. Esto hace posible resolver un problema cuando el valor de una variable no se conoce hasta un momento antes de que la respuesta sea necesaria. Por ejemplo, un automóvil que se está construyendo debe tener un color, pero dicho color no es importante hasta que se entregue.

En proceso de datos, es el programador, no la máquina, quien determina todas las relaciones entre los símbolos. Pero en el procesamiento de símbolos, en un sistema basado en IA, el programa puede determinar las relaciones entre los símbolos que no estaban establecidas explícitamente por el programador. Esta manipulación de relaciones entre parte de los datos es una característica importante de la programación en IA.

El aspecto distintivo de la IA es el énfasis en el almacenamiento y manipulación de las relaciones entre símbolos. Así, la tecnología de la IA tiene que ver con la preservación, no exactamente de datos, sino más bien con el conocimiento encarnado en las relaciones entre los datos. Las sentencias que expresan áreas particulares de conocimiento son tratadas como datos en un programa de IA.

En la resolución de la mayoría de los problemas en IA, el diseño de la solución no puede ser conocido de antemano. En

lugar de eso, se recurre a una programación exploratoria, usando varias técnicas de resolución e intentando producir una solución prototipo para una pequeña parte del problema de manera rápida.

Mejorando continuamente la capacidad del prototipo, el diseñador moldea el diseño para un sistema que por fin acaba haciendo lo que se pensaba. En el transcurso de todo este proceso, los sucesivos prototipos proporcionan un beneficio adicional: **el sistema es útil en el paso intermedio, estando lejos del sistema convencional de partida.**

Un sistema experto cuidadosamente construido, por ejemplo, puede ser útil al 30-50 % del nivel final. Los resultados del uso del sistema permiten a los usuarios proporcionar realimentación en la ejecución del trabajo. La realimentación se usa entonces para mejorar la ejecución, y el proceso continúa hasta que el sistema alcanza el nivel deseado.

Un beneficio adicional del último prototipo es que el sistema es capaz de mostrar sus utilidades y limitaciones potenciales, antes de que una gran cantidad de recursos se hayan puesto en juego. Un proyecto que no va a producir claramente los resultados pedidos, puede ser detenido antes de que consuma una cantidad mayor de recursos.

1.4. Algorítmica y Heurística.

La característica principal de la IA respecto de la informática convencional, reside en la utilización de la heurística, sin renunciar por ello al método científico. Para

entender el significado del término "*heurística*", realizaremos una serie de definiciones.

Entendemos por **Algoritmo** a *un procedimiento general de resolución de algún problema, tal que su validez no puede ser cuestionada por basarse en principios científicos, todo lo más podrá cuestionarse su coste.* Un algoritmo, como procedimiento general que es, siempre conduce a la resolución del problema para el cual fue creado. Como ciencia que estudia la construcción y propiedades de los algoritmos tenemos a la **Algorítmica**.

Por **Procedimiento Heurístico** entendemos *un procedimiento de resolución de un problema tal que se basa unicamente en la intuición o en la experiencia, y que sin embargo en la práctica resuelve el problema de forma correcta* y lo que es más importante existen procedimientos heurísticos que poseen costes más reducidos que el mejor de los algoritmos de tal problema.

La **Complejidad de un Problema** mide *la cantidad de tiempo y espacio que se precisan para la resolución del mismo.* El tiempo se mide en unidades de ejecución de una operación elemental, mientras que el espacio se mide en unidades de información, es decir bytes. Los problemas pueden clasificarse en función de la *evolución del coste en tiempo y espacio según evoluciona el tamaño del problema.* Podemos considerar que los problemas pertenecen a las siguientes clases generales:

- **Clase P:** *Problemas que poseen coste de tiempo polinomial respecto del tamaño del problema.*
- **Clase NP:** *Problemas que poseen coste de tipo exponencial respecto del tamaño del problema.*

- **Clase NP-Completo:** *Problemas con coste de tiempo polinomial y coste de espacio exponencial.*

Los algoritmos NP no suelen ser satisfactorios, ya que como procedimientos generales de resolución presentan una aplicación limitada debido a que un aumento del tamaño del problema convierte en no práctico el algoritmo.

Los procedimientos heurísticos posibilitan la resolución de problemas no prácticos, en forma práctica. Esta es la causa por la cual la incorporación del conocimiento heurístico en la técnica, y de una forma racional, es de vital importancia.

1.5. Diseño de ordenadores para IA.

Los ordenadores usados en IA se pueden clasificar en tres categorías:

- Los utilizados para el desarrollo de aplicaciones prácticas.
- Los que se entregan a los usuarios para aplicaciones prácticas.
- Los que se usan para investigación y desarrollo.

Pero los criterios para construir el soporte de *hardware* del trabajo en IA, difieren en las tres categorías. Así, en la

última clasificación se buscan las máximas prestaciones de los ordenadores.

Por esto la razón del proceso en la investigación está relacionada con la facilidad con que los investigadores pueden interaccionar con el ordenador y con la relación con la que pueden obtener y corregir respuestas del sistema de desarrollo. En I + D, el proceso de desarrollo de aplicaciones está mejor soportado por un *hardware* con buenas características y facilidades de programación.

1.5.1 Factores comunes a todas los tipos.

Los programas en IA requieren ordinariamente gran cantidad de espacio para direccionamiento, lo cual es una gran ventaja, pues permite direccionar por software distintas posiciones (de memoria, etc.), sin necesidad de recurrir al hardware para realizar gran cantidad de operaciones que necesitan procesos de conmutación, lo que haría que el proceso global resultase muy lento, de forma que así los procesos son rápidos, limpios y transparentes. Cara al futuro, el tamaño adecuado de la palabra en los ordenadores será de 32 bits. Con esto se puede direccionar todo lo que se desee con un programa de tamaño proporcionado.

Actualmente, los ordenadores personales más comunes tienen una memoria principal menor de 1 Mbyte, lo cual no es suficiente para la mayor parte de los sistemas expertos, aunque se están introduciendo ya ordenadores personales con memorias de hasta 16 Mbytes que son ya capaces de soportar sistemas expertos reales.

También los programas de IA se caracterizan por consumir una gran cantidad de ciclos de máquina,

comparados con los programas tradicionales. Esto es debido a que los programas de IA deben unir problemas complejos como búsquedas o modelados, por eso la velocidad es una de las características vitales entre las que deben considerarse en un ordenador que vaya a usarse en IA.

Asimismo, es necesario elegir un *hardware* que ejecute dialectos particulares de lenguajes de inteligencia artificial, sobre todo si se está trabajando con diversos problemas tratados con ordenadores distintos, en cuyo caso hay que recurrir a una distribución de las aplicaciones mediante la elección del hardware para interconexión a distintos niveles de redes, permitiendo distribuir la aplicación de forma pronta y conveniente teniendo acceso a los datos almacenados de forma centralizada

1.5.2 Sistemas para desarrollar aplicaciones.

Un factor importante a la hora de elegir el *hardware* para desarrollo de aplicaciones, es la facilidad y rapidez con que se pueden acortar las etapas de desarrollo y entrega del producto final. La compatibilidad y uso del mismo sistema, tanto para desarrollo como para terminar el producto, asegura que la aplicación pueda ejecutarse sin modificaciones.

Normalmente los criterios más comunes a seguir en la compra del hardware de aplicaciones son: precio, servicio, facilidad de instalación y mantenimiento, capacidad de mejorar el sistema así como soporte, entrenamiento y consulta por parte del vendedor.

1.5.3 Computers para desarrollo de aplicaciones.

Un ordenador de aplicación o propósito general puede ser la mejor solución cuando:

- 1.-*Además de servir para investigación y desarrollo, puede ejecutar programas de IA en aplicaciones reales, integrando códigos de IA con algoritmos tradicionales en el mismo programa.*
- 2.-*Tenga acceso multiusuario o concurrente a una gran base de datos.*
- 3.-*De forma análoga, un superordenador o bien tiene más hardware del que se necesita o del que puede usarse.*

Se deben tener en cuenta, para máquinas de aplicación general, las siguientes características:

a.- Deben tener herramientas para desarrollo tales como editores de lenguajes sensitivos, sistemas de gestión codificados y facilidades de depuración en multilenguajes.

b.- Espacio para direccionamiento virtual de 32 bits.

c.- Utilidades orientadas a pantalla, capacidad para manejar ventanas múltiples, así como programas para incrementar la productividad.

d.- Un buen conjunto de lenguajes estándar de desarrollo que se puedan usar como lenguajes ordinarios de interfaces, unido a la existencia de sistemas operativos para servicios y ayudas.

e.- Sistemas de bases de datos accesibles a todos los lenguajes, tanto relacionales como de gestión.

f.- Grandes redes operativas, incluyendo facilidad para comunicar con máquinas de otras marcas, para que los usuarios puedan compartir y distribuir recursos.

1.5.4 Grandes ordenadores con multiproceso.

Hasta hace poco, la mayor parte de la investigación y desarrollo en IA se hacía en "*mainframes*", debido a sus características de velocidad y flexibilidad, tanto para trabajos de cálculo tradicional como para IA. Lo mismo sucede con sus lenguajes, que son aptos para trabajar en los dos campos.

Recientemente, el precio de los miniordenadores y las estaciones de trabajo, ha bajado a un nivel tal que los "*mainframes*" han quedado para trabajos en "*batch*" (**procesamiento por lotes**) y soporte de grandes bases de datos relacionados con el proceso de desarrollo.

1.5.5 Miniordenadores.

Ofrecen rapidez y gran espacio para direccionamiento, así como una gran flexibilidad a un costo por usuario moderado, y desde luego a menor precio que los anteriores, por lo que los usuarios e investigadores los prefieren como ordenadores dedicados a este tipo de trabajos.

Tienen una amplia gama de software disponible para ejecutar porciones de algoritmos como soluciones de IA, unido a una gran cantidad de lenguajes para ser pasados bajo diferentes sistemas operativos.

1.5.6 "Workstations" de uso general.

Su uso es una tendencia generalizada entre los usuarios de investigación y desarrollo de IA, debido a que:

1.- Pueden usar toda la potencia de cálculo y facilidades necesarias para optimizar al máximo cualquier tarea de IA, desarrollada por cualquier usuario.

2.- Individualmente, aseguran suficientes recursos para soportar gráficos con técnicas de mapeado de memoria bit a bit, lo que requiere gran potencia de procesamiento.

3.- Tienen disponibles herramientas y utilidades generales como: sistemas y utilidades UNIX, elección de diversos editores, procesado de textos estándar, etc.

4.- Pueden usarse tanto en miniordenadores como en "mainframes", tanto para desarrollo como para productos acabados.

Una estación de trabajo para IA debe tener una alta resolución en pantalla además de grandes opciones de apertura de ventanas para poder crear, llenar, mover, reformar, recortar y superponer. Debe poseer también utilidades de diseño, para programación de sistemas en el lenguaje que se elija.

Aun cuando es importante la velocidad y que se pueda trabajar en ambientes críticos, es más importante aún el costo por usuario, teniendo en cuenta que la mayoría de los sistemas finales requieren capacidades interactivas, con grandes pantallas, además de poder tratar gráficos. Debe considerarse, además, la necesidad de estar conectado a diversas redes para que así la aplicación esté eficientemente distribuida y los resultados compartidos convenientemente.

1.5.7 Ordenadores para I+D.

Abarcan un amplio espectro que va desde los superordenadores hasta las máquinas especiales (máquinas LISP), "*mainframes*" de uso general, miniordenadores y estaciones de trabajo.

Hay que considerar diversos factores en el momento de elegir un ordenador para I+D en IA entre los distintos tipos existentes, tales como: **costo, características** y **software**. No obstante, el primer factor a tener en cuenta son las características de funcionamiento y el rendimiento; y dentro de estas últimas la velocidad, dado que el tiempo requerido para la fase de investigación y diseño de un proyecto depende, al menos de un modo parcial, de la rapidez con que puedan realimentarse los lazos correspondientes al camino crítico y pruebas por fallos del sistema. De ahí la importancia que tiene a la hora de la elección del ordenador para los sistemas I+D.

En estas etapas son más exhaustivas las exigencias de gran capacidad de memoria principal, dado que para IA se requiere al sistema para tareas de compilación, edición, depuración, etc. Estas facilidades conllevan grandes cantidades de espacio en disco, espacios de memoria virtual, y de ciclos de máquina, aun cuando la memoria final necesaria dependa del tipo de aplicación. Debido a esto, se sobredimensionan los sistemas de respuesta mediante pantallas de alta resolución, prediciendo la respuesta temporal, o eligiendo cuidadosamente el ancho de banda de los canales de entrada/salida.

A menudo se usan también, a través de sus respectivas interfaces, sistemas que ejecutan eficientemente tanto software tradicional como el que se usa en IA. Esta compatibilidad permite usar métodos tradicionales para desarrollar porciones de algoritmos de programas de IA, mezclando su uso con programas y bases de datos ya existentes, dentro de programas mayores de IA.

1.5.8 Superordenadores para I + D.

Actualmente estos equipos, tales como **CRAY II, Fujitsu,** etc., se usan en problemas específicos que requieren una gran velocidad. Los superordenadores no tienen ni el software ni los sistemas operativos que conducen al desarrollo de IA, y tampoco aquéllos están disponibles de una forma amplia.

1.5.9 Workstations en I + D: máquinas LISP.

Están en el mercado desde **1981** y realizan procesamiento simbólico pasando el lenguaje **LISP**. Las máquinas **LISP** no son más que estaciones de trabajo con **arquitecturas hard** diseñadas para ejecutar **LISP**.

Sus memorias principales varían entre 4 y 16 Mbytes, y el precio de estas configuraciones varía entre 80.000 y 170.000 dólares. Las ventajas de un ordenador optimizado para trabajar con LISP estriban en su gran memoria física y virtual, sus métodos de gestión de la misma, su rapidez, pero, sobre todo, en su arquitectura optimizada para pasar **LISP**.

Normalmente estas máquinas tienen: *mapeado de bits, gráficos de alta resolución con ventanas solapables* (ayuda muy útil para el desarrollo de programas), permitiendo representar una actividad discreta de más de un proceso en una pantalla como si se tuviese en ella más de un puesto de trabajo. También permiten cambiar de un entorno a otro sin pérdidas ni abandono del contexto; se usan en ellas también los *gráficos* para ilustrar las informaciones y relaciones.

La primera **máquina LISP** se construyó en el Laboratorio de Ciencia de Ordenadores del **MIT** y en el Centro de Investigaciones de la firma **Xerox Corporation** en **Palo Alto** (California, U.S.A.).

1.6. Evolución histórica de la IA

El nacimiento de la IA tuvo lugar en 1956 en una conferencia de diez prestigiosos científicos en el **Dartmouth College (U.S.A.)** que discutieron las posibles maneras de simular el comportamiento humano mediante los ordenadores.

Entonces predijeron que en 25 años los ordenadores serían capaces de hacer todo el trabajo de las personas, que estarían dedicadas fundamentalmente a actividades de ocio. En **1981**, en la *Conferencia Internacional sobre IA en Vancouver (Canadá)* un panel de cinco de aquellos científicos reconocieron el exceso de optimismo que reflejaban aquellas predicciones.

En los 15 años que van desde 1956 hasta 1970 el principal esfuerzo se concentró en la **traducción por ordenador.** Se pensaba entonces que la traducción del lenguaje natural se podría llevar a cabo mediante el uso de un diccionario bilinge y las correspondientes reglas gramaticales. Este enfoque fracasó estrepitosamente debido a una serie de factores tales como palabras dependientes del contexto, frases hechas y ambiguedades sintácticas.

En 1967 los resultados obtenidos eran tan desalentadores que la investigación en esta línea se abandonó. Y ha sido sólo muy recientemente cuando el tema de la traducción se ha vuelto a considerar.

Otros temas de investigación en este periodo fueron los de:

- Integración simbólica desarrollada en MIT y que sirvió como base a una serie de programas de matemática simbólica que culminaron en MACSYMA de uso en MIT y otras comunidades científicas a través de la red ARPA.
- Juegos tales como el ajedrez y damas, de uso frecuente hoy en día.
- Proceso de imágenes y reconocimiento de formas como precursores de la visión por ordenador. Este último tema se separó de la IA como área independiente, pero actualmente las dos disciplinas han vuelto a acercarse.
- Resolución de problemas y demostración de teoremas.

Al final de este periodo, en 1970, los investigadores se encontraron con que el éxito alcanzado en los temas de IA había sido muy restringido con un fracaso rotundo en alguno, como el de la **traducción automática**.

Los problemas ficticios de entretenimiento o los fácilmente planteables podían tratarse bien, pero cuando se

pretendía abordar problemas reales complejos, o no se encontraban soluciones adecuadas o bien el proceso de búsqueda tenía tantas combinaciones posibles que excedía la capacidad de los ordenadores de entonces.

Uno de los procesos intelectuales humanos que en aquel tiempo se intentó trasladar al ordenador originando continuos fracasos y frustraciones es el denominado *sentido común* (que se asume tienen la mayor parte de las personas). El sentido común supone un razonamiento de bajo nivel basado en una serie de experiencias que las personas van adquiriendo desde la niñez y que permiten aprender cosas tan obvias como que al soltar un objeto éste se cae (sin pensar en la ley de Newton sobre la gravedad). El mismo tipo de razonamiento nos indica la pauta de comportamiento ante las situaciones ordinarias de cada día.

Esto, que es trivial para las personas, resulta muy difícil de conseguir en IA ya que, contrariamente a lo que en su inicio se pensaba, este sentido común no es una forma de razonamiento que se apoye en un conjunto reducido de reglas universales, sino más bien el resultado de aplicar unas reglas empíricas y poco precisas sobre cantidades ingentes de conocimientos implícitos que, como decíamos, se van adquiriendo desde la niñez.

Como resultado de esta experiencia comenzó a trasladarse el énfasis de los sistemas de IA desde los conjuntos de reglas y procedimientos generales de inferencia, como ocurría en esta primera etapa, hasta la **acumulación organizada de masivas cantidades de conocimiento**.

Este cambio de énfasis fue el inicio del resurgir de la IA que trascendió públicamente hacia final de los 70 pero que comenzó a fraguarse desde principios de esa década.

Así pasamos a la siguiente etapa histórica que corresponde a la década de los 70.

En este período, la IA se veía desde fuera como una investigación poco prometedora, en cuanto a las posibilidades de obtener aplicaciones prácticas, tras el entusiasmo despertado en los años que siguieron a su descubrimiento.

No obstante las investigaciones continuaron capitalizando las experiencias anteriores. Así por ejemplo, y aparte del cambio de énfasis que mencionábamos con anterioridad, las técnicas de búsqueda maduraron y empezaron a aparecer nuevos métodos de representación del conocimiento.

Se construyeron prototipos, en muchos casos con fines demostrativos, en los temas de visión y de proceso y comprensión del lenguaje. También aparecieron unos programas de ordenador que trataban de capturar y utilizar la experiencia de especialistas y que se llamaron **"sistemas expertos"**.

Estos sistemas funcionaban en áreas de conocimiento bien delimitadas tales como cristalografía, diagnóstico médico, prospección geológica, etc., lo que motivó por otra parte, la *interacción de otras materias* con la IA.

Las experiencias de esta etapa resaltaron la importancia que el tema de **la adquisición y representación del conocimiento** tiene en la inteligencia, dando origen a un nuevo sector dentro de la IA denominada **"ingeniería del conocimiento"** que se asocia y se identifica con el desarrollo de **sistemas expertos**.

La última etapa histórica de la IA, que corresponde a la presente década, se caracteriza por la consolidación del resurgimiento que se apuntaba hacia el final de la década anterior y que en ésta comienza a trascender fuera de las comunidades investigadoras.

La novedad más importante de este momento se deriva del *éxito* de los **sistemas expertos** que han dejado de ser prototipos de demostración para pasar a la fase de comercialización y dar origen a numerosas empresas de IA -muchas de ellas formadas por investigadores- que obtienen importantes beneficios económicos con las aplicaciones creadas en sus laboratorios.

Compañías de ordenadores, electrónica, petrolíferas y otras han creado sus propios grupos de IA.

Aparte de los sistemas expertos, se han comercializado también sistemas de acceso a bases de datos en lenguaje natural, sistemas de visión simplificados con aplicaciones para robots y sistemas de síntesis y reconocimiento de voz.

1.7. Situación actual

La tecnología de la IA está evolucionando hoy rápidamente, de manera que, en la actualidad, es posible poner ya en práctica muchas de las aplicaciones que se pronosticaron en los comienzos de esta tecnología.

A su vez están apareciendo y desarrollándose nuevas aplicaciones y, como dijimos anteriormente, el número de compañías y grupos de IA prolifera de modo continuo.

Las aplicaciones que están teniendo más éxito son las de los **sistemas expertos** que multiplican su aparición en el mercado. Hoy en día hay sistemas funcionando en hospitales, para diagnóstico y tratamiento de enfermedades, en mantenimiento y reparación de cables y máquinas, en configuración de ordenadores, en perforaciones petrolíferas, y en prospecciones mineras.

Paralelamente a los logros conseguidos en IA, los avances conseguidos en microelectrónica, con un *grado de integración cada vez mayor* y un *menor coste,* han hecho posible el desarrollo de productos de IA, con lenguajes de programación específicos de esta tecnología tales como el **LISP** - y sus versiones - y el lenguaje **PROLOG,** cuyo estudio es el objetivo principal de este proyecto fin de carrera.

La importancia que la IA está cobrando en los países tecnológicamente avanzados ha hecho que en **Japón** el gobierno y las industrias nacionales hayan emprendido el proyecto de los "*Ordenadores de la 5ª generación*", que empezó en 1982 y terminará en **1992**.

El objetivo de este proyecto es desarrollar, para la década siguiente, una nueva tecnología de ordenadores distinta de la de Von Neumann y basada en la **lógica natural** en vez de en la lógica del tipo máquina.

Se pretende con ello establecer un nuevo concepto que incluya una **nueva arquitectura**, con **proceso en paralelo** a gran velocidad y un **nuevo tipo de software**, que produzca en

conjunto una máquina superior con una *lógica más parecida al razonamiento humano*, con la que se pueda dialogar en **lenguaje natural** para conseguir unas aplicaciones más inteligentes.

El reto planteado por el **Japón** ha hecho que en **U.S.A.**, donde la economía libre de mercado excluye la aparición explícita de ese tipo de proyectos financiados por el gobierno, hayan surgido iniciativas, como la de la empresa **MCC (Microelectronics and Computer Technology Corp.)** para competir con el **Japón** en la producción de la próxima generación de ordenadores.

Es importante destacar que **MCC** está formada por 13 empresas de ordenadores y semiconductores que son competidoras unas de otras en el mercado, pero que han decidido unir sus esfuerzos en este campo para poder afrontar con éxito una tecnología tan cambiante y costosa.

Por otro lado, agencias tales como la **DARPA** del Departamento de Defensa, la **Marina**, la **Aviación** y el **Ejército** de **U.S.A.** comienzan a dedicar notables esfuerzos y medios creando *centros de investigación* y financiando importantes proyectos en IA.

En **Europa**, los países de la **CEE** han decidido también aunar sus esfuerzos en el programa **SPRIT (European Strategic Program for Research & Development in Information Technology)** para, como dice textualmente su preámbulo, hacer frente al desafío japonés y norteamericano, con objetivos a diez años vista.

Una de las líneas del programa **SPRIT** es la denominada *Proceso Avanzado de la Información* que trata de sentar las bases para explotar industrialmente la transición de los

sistemas de proceso de datos a los de proceso del conocimiento, que corresponden a la próxima generación de ordenadores.

Concretamente los temas de investigación en esta línea incluyen *sistemas expertos* e *ingeniería del conocimiento*, *intercomunicación hombre-máquina* así como *técnicas avanzadas de software y hardware para el almacenamiento de la información y del conocimiento*.

Aunque es de esperar un considerable avance de esta tecnología en los próximos años y décadas, el estado actual de la misma, sin embargo, permite realizar ya ciertas aplicaciones que en etapas anteriores no pudieron realizarse con éxito. En concreto:

- Los sistemas expertos no son ya, en muchos casos, herramientas de investigación o piezas de laboratorio sino que, como dijimos, están superando la frontera comercial y transformándose en fuente de abundantes beneficios económicos. No obstante, están todavía restringidos a áreas de conocimientos bien delimitadas y requieren, para su construcción, una laboriosa interacción con especialistas en la materia y un largo proceso de depuración. Algunos de estos sistemas proporcionan, todavía, soluciones satisfactorias (no necesariamente óptimas) en un porcentaje de los casos solamente y en otros son incapaces, incluso, de dar una respuesta o bien ésta es totalmente errónea.

- Los sistemas basados en interface con lenguaje natural han superado las dificultades iniciales y algunos están en el mercado. Sin embargo todos utilizan subconjuntos del lenguaje natural y en algunos casos fracasan en la comprensión y el procesamiento de la información del usuario, que necesita un cierto grado de entrenamiento. Del mismo

modo, empiezan a aparecer algunos traductores automáticos en temas muy restringidos, tales como el de la predicción metereológica. Los intentos de emplearlas a temas más generales tropiezan todavía con serias dificultades y evidencian numerosos fallos. Los proyectos actuales se dirigen hacia la creación de instrumentos que sirvan de ayuda a especialistas humanos, que pueden incrementar así su productividad, de 2 a 10 veces dependiendo del tema que traten. Actualmente la comprensión del lenguaje y la generación del texto están aún en fase de investigación.

- La visión por ordenador ha superado ya la etapa de desarrollo y ha comenzado a entrar en los circuitos comerciales en los que, unas cuantas empresas, ofrecen sistemas sofisticados de visión que funcionan bien en circunstancias especializadas o en la verificación, inspección, reconocimiento y determinación de objetos concretos. La mayor parte de esos productos tratan imágenes en dos dimensiones.

En resumen, el estado actual de la IA es prometedor, aunque los sistemas actuales sean aún bastante restringidos en cuanto a su dominio de actuación, fuera del cual presentan bastantes fallos y deficiencias.

Por este motivo las aplicaciones de estos sistemas deben considerarse más dentro de un ámbito de instrumentos "*inteligentes*" de ayuda a las personas que no como verdaderos sistemas autónomos.

1.8. Perspectivas futuras

En el campo de la IA, las previsiones más optimistas para un futuro inmediato son, hacia los **sistemas expertos** que,

junto con los programas y equipos necesarios para desarrollarlos, tienen actualmente un mercado de varias decenas de millones de dólares.

Se estima que esta cifra pasará a 2.500 millones en 1993. Simultáneamente se espera desarrollar un mercado de software que permitirá mejorar notablemente los interfaces hombre/máquina con sistemas de comunicación en **lenguaje natural** y **visión por ordenador**.

Se prevé para este mercado una cifra de 1.800 millones de dólares en 1993.

El desarrollo de la microelectrónica, favorecerá la aparición de valiosas herramientas inteligentes para ayuda a los profesionales en la forma de **sistemas expertos, interfaces con lenguaje natural, sistemas de visión por ordenador y robots inteligentes.**

Se pretende que estos sistemas se asemejen más al comportamiento humano de manera que los fallos que puedan tener, como los que tienen las personas, no sean desastrosos sino que se produzca una degradación progresiva cuando se vayan aproximando a los límites de su conocimiento.

- En relación con los sistemas expertos se espera utilizar las herramientas que vayan surgiendo en su desarrollo para poner a disposición de cualquier persona, con un ordenador personal (probablemente de la nueva generación de microprocesadores de 32 bits), programas de asistencia médica, legal y financiera, añadiendo inteligencia a los sistemas de ayuda a la decisión. Se espera también la proliferación y mejora de sistemas expertos en las áreas antes mencionadas con énfasis principal en los sistemas de enseñanza por ordenador que ahora serán más inteligentes. Esto puede traer cambios sustanciales en el área de la educación y de la formación.

- En relación con el lenguaje natural, la tendencia es a mejorar el modo de acceso a los ordenadores mediante el reconocimiento de la voz. Uno de los problemas con que los investigadores se han encon trado es el de la repulsa psicológica que muchas personas experimentan hacia el hecho de dialogar con una máquina. Por lo tanto la superación de esta reticencia puede ser incluso más importante que la reducción del coste. Para ello parte de la investigación se dirige hacia la obtención de un flujo continuo de conversación así como de unos tonos y timbres de voz agradables. Se están desarrollando incluso sistemas en los que se pueda elegir ciertas características de la voz de la máquina interlocutora, tales como la de ser hombre, mujer o niño, o bien de un acento de una determinada región o país. La ventaja de usar el lenguaje natural como modo de acceso a los ordenadores es el incremento extraordinario de usuarios que este hecho proporcionaría. Sin embargo, aún tendrán que transcurrir varios años para poder mantener un diálogo sin restricciones. Se espera que para 1989 el acceso público a grandes bases de datos sea posible utilizando un conjunto restringido del lenguaje natural. Como consecuencia, se abrirán nuevos mercados de servicio de información, como los de compra y reservas, a los que se podrá acceder por redes telemáticas.

- En relación a la visión por ordenador se espera construir sistemas capaces de identificar, inspeccionar, localizar y verificar objetos, de tal forma que estos sistemas se conviertan en los sensores de los robots inteligentes. Probablemente al final de la presente década el 25% de los robots tendrán visión incorporada. Los sistemas de visión jugarán un papel importante en aplicaciones militares y espaciales y también en la ayuda a ciegos y minusválidos.

- Los robots industriales que existen actualmente, aunque funcionan con programas muy sofisticados, son dispositivos que actúan de modo repetitivo. Los avances en IA hacen posible el desarrollo de robots inteligentes que, no solamente permitirán mayor flexibilidad en los procesos sino que ampliarán su radio de acción fuera del entorno industrial, por ejemplo en exploraciones marinas y geológicas, construcción, extinción de incendios, etc. Uno de los mayores impactos de los robots inteligentes se espera que tenga lugar en aplicaciones espaciales como la exploración del espacio y la ayuda en la operación y mantenimiento de instalaciones de ese tipo. También se espera para la próxima década tener robots con dotes sensoriales e inteligencia como para ser útiles en las industrias de servicios y en aplicaciones domésticas.

- Otras aplicaciones de futuro serán las de planificación y automatización industriales. Además de producir robots más inteligentes la IA incidirá en el desarrollo de plantas industriales integradas mejorando la planificación automatizada, el control de procesos, el almacenamiento de existencias y el funcionamiento de robots, vehículos y máquinas de producción.

El impacto más importante que puede tener lugar, en el campo de la IA, puede venir como consecuencia del logro de que las máquinas sean capaces de aprender.

Actualmente algunos sistemas experimentales con cierta capacidad de aprendizaje han producido resultados y experiencias interesantes.

El objetivo es lograr que algún día las máquinas puedan aprender de la experiencia y ampliar sus conocimientos a lo largo de su vida, de manera que sean capaces de razonar con cierto sentido ante situaciones complicadas.

Esto abriría caminos espectaculares hacia nuevas aplicaciones en la industria, la oficina y los hogares.

No cabe duda que el desarrollo de la IA, unido al del resto de las técnicas informáticas y microelectrónica va a producir cambios importantes en la sociedad.

Así por ejemplo, es previsible que se produzca un desplazamiento gradual del empleo desde el entorno productivo y de fabricación hacia el de los servicios, de modo parecido al que tuvo lugar con el cambio de la agricultura a la industria durante la revolución industrial.

También afectará a las actividades de investigación, desarrollo, servicios y recreativas.

Al final la cuestión central será como reestructurar la sociedad, para que ese cambio sea positivo con mayor número de bienes y servicios y más tiempo para actividades de ocio.

1.9. El Método.

Por método entendemos el modo de realizar con orden la actividad científica y académica en una disciplina según su propia naturaleza. Es decir, el procedimiento que se sigue en cada rama de la ciencia y de la ténica para obtener nuevos conocimientos o nuevas aplicaciones y para saber transmitirlos de forma coherente, ordenada y sistemática.

Un aspecto metodológico de la IA no clarificado, o sobre el que no existe acuerdo unánime, reside en la forma en que se debe incorporar la heurística en la IA, no la incorporación misma que es considerada por todos como útil y necesaria. Una tendencia consiste en incorporar el conocimiento heurístico dentro de un esquema formal coherente y sistemático, otra tendencia sin embargo, predica la utilización de esquemas heurísticos en si mismos. Es evidente que esta última tendencia solo puede representar una etapa inicial en la evolución de la IA, etapa de carácter empírico que es muy interesante para toda nueva técnica, como lo es la IA.

El objetivo básico de la IA es simular el conocimiento y pensamiento humano. Pero dado que el cerebro humano parece tan diferente de los computadores digitales convencionales, el citado objetivo entra en conflicro con la realidad práctica. Basándose en este aspecto, pueden

distinguirse dos aproximaciones metodológicas diferentes al mundo de la IA, denominadas teórica y práctica. Para los defensores de la última, en IA el único criterio para juzgar un sistema, es el funcionamiento correcto en las condiciones especificadas, mientras que para los defensores de la primera es preciso algún criterio más general. Los partidarios de las aproximaciones prácticas sostienen que la utilización de teorías psicológicas y otras basadas en el cerebro humano, no son adecuadas para implementaciones en computadores, por las siguientes razones:

- Los humanos utilizan procesos con poca computación pero con gran capacidad de conocimiento estructurado.
- Los computadores realizan procesos fundamentalmente repetitivos, utilizando bastante computación sobre datos altamente regulares.
- Los computadores son más adecuados para la computación rápida de algoritmos sencillos, que para la simulación de procesos del conocimiento.

En muchos problemas de IA, los algoritmos de resolución exigen un tiempo de cómputo de tipo exponencial. Las soluciones **heurísticas**, sin embargo, proporcionan buenas soluciones en tiempo polinomial. En muchas de las cuestiones de la IA que conducen a costes de computación de este tipo, por ejemplo los métodos de exploración, se adopta la metodología de la investigación heurística, que en algunos casos se basa en modelos humanos.

Muchos de los campos científicos, la IA entre ellos, se polarizan en aplicaciones prácticas y teóricas, aceptando ambas la necesidad de la otra, pero separándose más o menos

en las líneas de investigación y en sus **metodologías**, situación ésta reflejada en la IA como sigue:

> El debate se inició entre los "desaliñados", liderados por R. Schank y E. Feigenbaun, y los "pulcros", liderados por N. Nilsson. Estos últimos sostienen que la educación en la IA no está completa sin un componente teórico fuerte, conteniendo por ejemplo, cursos en lógica de predicados y teoría de autómatas. Los "desaliñados" mantienen que tal componente teórica no sólo es innecesaria sino que, sino hasta perjudicial.... El producto final de los desaliñados es un programa de computador funcionando, mientras que los "pulcros" no están satisfechos hasta que extraen una teoría del programa.

El punto de vista teórico en IA asume que unos principios elegantes pueden abarcar muchas de las manifestaciones de la inteligencia humana, y el descubrimiento de tales principios proporcionará la llave para los trabajos del conocimiento. Desde el punto de vista práctico, en IA se utilizan muchas aproximaciones "ad hoc", para muchas actividades diferentes, no pudiéndose encontrar principios generales ya que cada aplicación requiere una gran cantidad de **conocimiento específico al dominio**. Ambas partes de la controversia se atribuyen victorias, ninguna de ellas definitiva. Por un lado, los teóricos han encontrado técnicas generales para resolver los problemas de Deducción, Resolución de problemas y Planificación de tareas, pero la mayoría se ejecutan demasiado lentamente para las aplicaciones en tiempo real. Por otro lado, los prácticos se apuntan como logro la producción de sistemas que guian robots, reconocen objetos y mantienen conversaciones, pero no pueden transportar sus técnicas má allá de ligeras desviaciones de las condiciones iniciales.

La metodología de la IA debe constituirse tanto con el enfoque "***pulcro***" de construcción de modelos generales del conocimiento, como con el "*desaliñado*" de producción de

sistemas "*ad hoc*" con técnicas "*heurísticas*", que se desenvuelven con agilidad y eficacia en el mundo real. La conclusión final es que el método de la IA debe ser el de construir programas "*desaliñados*" y **analizar** los mismos con el objetivo de extraer algún nucleo de estructura y técnica "*pulcra*".

Otro aspecto metodológico de importancia en la IA, en lo que respecta a las herramientas de la misma, es decir, a los Métodos de Programación, que ha conducido a la controversia entre **declarativistas** e **imperativistas**, la cual se centra en el **qué** y el **cómo** de los problemas. Los primeros asumen que el conocimiento es una colección de hechos y reglas que se representan por proposiciones lógicas, grafos conceptuales u otras estructuras simbólicas. Los segundos, por el contrario, estiman que el conocimiento está contenido en los procedimientos de interpretación y manipulación del entorno. Ambas posturas son ilustradas por H. Simon, en las siguientes definiciones de un circulo:

- Un círculo es el lugar de los puntos que equidistan de uno dado.
- Un círculo es la figura que resulta de rotar un compás, con un extremo fijo, hasta que el otro extremo regrese al punto de partida.

La primera definición, de tipo **declarativista**, no define como construir un círculo, mientras que la segunda, de tipo **imperativo** o **procedural**, no expresa como caracterizarlo y/o reconocerlo.

La programación procedimental utiliza una descripción del problema basada en la especificación de un conjunto de órdenes o instrucciones que ejecutadas en un orden conducen

a la solución. La representación más general de un programa procedimental la constituye un diagrama de la secuencia de instrucciones. Una máquina que ejecute un programa procedimental es una máquina secuencial.

La programación declarativa o descriptiva realiza una descripción de los problemas en forma de las relaciones lógico funcionales de los componentes o datos del mismo. La programación declarativa no especifica la forma de alcanzar la solución, sino la relación que debe existir entre ésta y los datos, por esta causa una máquina que incorpore la programación declarativa debe realizar una inferencia de la solución a partir de los datos y de las relaciones especificadas. Generalmente toda máquina de inferencias realiza un proceso de exploración de alternativas.

La IA adopta de la informática, de la cual proviene, la metodología de separar los aspectos de **Representación** de los aspectos de **Control**. El primero se refiere a la presentación y almacenamiento de la información y el segundo a su manipulación. En la informática estos dos aspectos dan lugar a la **Estructura de Datos** y a la **Algorítmica**. Dentro de la IA los problemas de representación se estudian como **Representación del Conocimiento**, por ser este elemento la parte más importante de todos los sistemas de IA.

La mayor parte de los sistemas de IA utilizan una programación declarativa, lo cual conduce a que el procedimiento de control practicamente universal en IA sea el de **Exploración de Alternativas**. Procedimiento de control que por otra parte es el **más simple y general** de los conocidos por el ser humano.

De entre todos los sistemas de representación y control que se han desarrollado, destacaremos los **Sistemas de**

Produccion. Estos sistemas son un raro ejemplo en los cuales la representación y el control resultan altamente sencillos, en cuanto a utilización y diseño, y potentes en cuanto a prestaciones.

2. EXPLORACION DE ALTERNATIVAS.

La búsqueda o **Exploración de Alternativas** desempeña un papel crucial en casi todas las parcelas de la IA, constituyendo, como se verá más adelante, el nucleo central de la resolución de muchos de los problemas que exigen la selección de alguna opción entre un conjunto de posibilidades. Este es el caso que se presenta en problemas como:

- *Demostrar un teorema de lógica.*
- *Determinar el camino óptimo que debe seguir un robot móvil.*
- *Determinar las transformaciones necesarias para resolver una integral.*

En todos estos casos el papel de la búsqueda o exploración de alternativas es fundamental en la resolución del problema.

Un algoritmo o procedimiento general actúa generalmente en un sistema de representación o estructura de datos. Podemos considerar que cada situación de esta

estructura representa un estado en la solución de un problema, así la situación inicial del problema representa el estado inicial, representando la estructura los datos iniciales del problema.

La solución del problema consiste generalmente en una configuración de la estructura que es la meta u objetivo, denominandose a tal situación el **estado final**. La resolución del problema se consigue cuando se alcanza el estado final a partir del inicial, mediante el paso por un conjunto de estados intermedios.

El algoritmo de un problema establece la trayectoria en el espacio de estados, trayectoria que equivale a la secuencia que resuelve el problema.

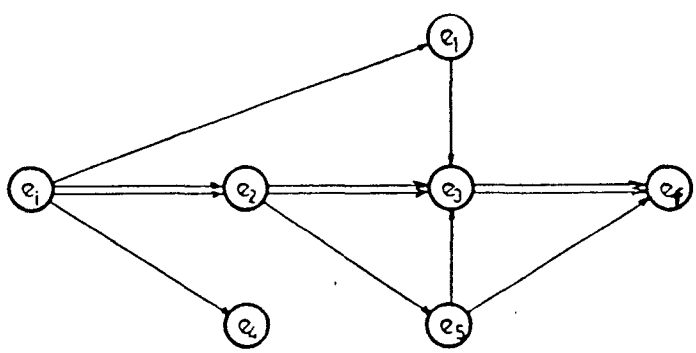


Figura 1

Existen muchos problemas para los cuales no puede establecerse un algoritmo, es decir, una trayectoria en el espacio de estados, esto se debe, generalmente, a la complejidad y volumen de los estados posibles. En estos casos la determinación de una trayectoria se establece en base a explorar distintos caminos, abandonando aquellos que se consideren poco adecuados. Este método de resolución de problemas, denominado **Exploración de Alternativas**, es basicamente el procedimiento de Ensayo-Error y constituye el procedimiento más general de resolución, y por tanto el

menos eficaz, pero por múltiples circunstancias es el único procedimiento posible.

2.1. Componentes de un sistema de búsqueda.

Los sistemas de **Solución de Problemas** (Problem Solving) como también se denomina a los sistemas de búsqueda o exploración de alternativas, pueden ser descritos, de un modo general, en base a tres componentes principales:

Base de datos o representación, que describe la situación del dominio del problema, así como el objetivo final que se desea alcanzar y la situación inicial en determinados casos. En muchos de los problemas de IA estas situaciones del problema pueden ser descritas en términos de **Estados.**

Operadores o reglas que describen la transformación del dominio del problema, es decir, transformadores de los estados-situaciones. Otro tipo de representación común en IA es aquella en la que la aplicación de un operador puede dividir un dominio o problema en varios subproblemas, presumiblemente de complejidad menor o de solución más sencilla. Este tipo de representación se denomina de **Reducción del Problema.** Siendo los operadores dependientes del tipo de representación, que se clasifican fundamentalmente en **Espacio de Estados** y **Reducción del Problema.**

Estrategias de control que definen los operadores a aplicar en cada situación del problema, eligiendo éstos entre un gran número de ellos. En general muchos de los problemas en IA se resumen en alcanzar un **Objetivo** por aplicación de una secuencia apropiada de operadores a partir de alguna situación o Estado inicial.

La aplicación de cada operador modificará la situación de un modo diferente, la aplicación de una secuencia de operadores puede conducir a la resolución del problema o no, siendo responsabilidad de la estrategia de **control** la selección de la secuencia de operadores que conducen a la consecución del objetivo según algún criterio, que simplemente asegure la consecución del objetivo o que lo haga de alguna forma óptima.

2.1.1. Representación en árbol.

Generalmente se puede pasar de la representacion en modo "**espacio de estados**" o "**reducción del problema**" a la representación en forma de Arbol, mediante el **árbol expandido del grafo de estados**. En casos de "reducción del problema", el árbol resultante será del **tipo And/Or**, este tipo de árbol se explica más adelante.

En la representación en árbol se designa cada situación o estado mediante un nodo, siendo los nodos hijos del mismo los que representen los estados resultantes de la aplicación de operadores de transformación. Sea **P** un nodo que representa un estado del problema, la aplicación de un operador T_i producirá un estado transformado que será representado por un nodo H_i, hijo del anterior, la aplicación del conjunto completo de operadores del problema, $(T_1,T_2,...,T_n)$, producirá los sucesores del nodo P $(H_1,H_2,..H_n)$.

En esta representación la resolución completa del

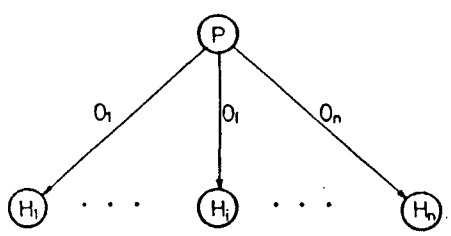


Figura 2

problema constará de un nodo raiz,que representa el estado

inicial del cual desciende un árbol, alguno de cuyos nodos descendientes coincidirá con el estado final u objetivo. **El número de operadores del problema determinará el grado de ramificación del árbol que representa el problema**. En determinados tipos de problemas se puede considerar la existencia de nodos prohibidos, correspondientes a estados que violan determinadas restricciones condiciones específicas del problema.

No debe olvidarse el hecho de que un árbol de búsqueda no representa a ningúna estructura de datos concreta, sino que se produce como consecuencia de la resolución del problema y es más bien una representación del problema desde el punto de vista del control o secuencia de determinación de la solución.

2.1.2. Razonamiento hacia-delante y hacia-atrás.

En el caso del problema anterior hemos resuelto el problema partiendo de la situación inicial aplicando operadores hasta encontrar el estado final. La aplicación de operadores a las estructuras de representación, tanto en el caso de "espacio de estado" como en el de "reducción del problema", partiendo de la situación inicial con el propósito de alcanzar la situación objetivo, es denominada **Razonamiento Hacia-Delante** (reasoning forward).

Una estrategia alternativa a la anterior es el **Razonamiento Hacia-Atrás** (reasoning backward), que implica la utilización de otro tipo de operadores, inversos a los anteriores, que se aplican en el objetivo y no en la situación inicial, de tal forma que se obtienen situaciones previas a la de resolución y por aplicación de éstas, llegar al estado inicial.

El problema tendrá solución en el primer caso si es posible alcanzar el objetivo a partir del inicio, mediante la aplicación de los operadores del problema, en el segundo caso la solución del problema es posible si por aplicación de los operadores inversos es posible alcanzar el inicio a partir de la situación objetivo.

Estas dos formas de contemplar el proceso de búsqueda se denominan también **Dirigidas por Datos o Bottom-Up** en el primer caso, y **Dirigidas a Objetivos o Top-Down** en el segundo, todos estos nombres se aplican por similitud con otros tipos de problemas en la informática convencional.

Muchos de los métodos humanos de resolución de problemas parecen estar conducidos por una estrategia de control del tipo hacia-atrás y por esta causa de proximidad con el método de razonamiento humano, muchos de los sistemas de IA usan esta estrategia.

2.1.3. Estructura de los Arboles And/Or.

En los esquemas de representación basados en la "reducción de problemas", la principal estructura de la base de datos es la descripción de problemas objetivos. Dada una descripción inicial del problema, se resolverá la misma ejecutando una secuencia de transformaciones que en último término, convierten la descripción en un conjunto de subproblemas cuya resolución sea inmediata. Estos últimos se denominan *sistemas primitivos.*

Un operador puede transformar un problema en varios subproblemas de forma que para dar solución al problema

principal, se deberá resolver todos los problemas hijos. Esta situación puede representarse con un nodo del cual nacen diversos nodos hijos, de tal forma que la resolución del nodo padre precisa la resolución de todos sus nodos hijos.

Otra situación diferente se presenta cuando a un mismo sistema se aplican diferentes operadores, en este caso no será preciso resolver todos los subproblemas generados, sino que con la solución de un caso queda resuelto el principal, este caso se puede representar como un nodo del cuál nacen diversos nodos hijos resultante cada uno de ellos de la aplicación de diversos operadores, siendo preciso resolver solo uno de los hijos para obtener la solución del nodo principal.

En un caso genral se presentarán diversas combinaciones de ambos casos, por lo cuál la construcción de un árbol será problemática dada las diversas naturalezas de los nodos hijos. Para resolver este problema, se introduce un tipo de árbol denominado **Arbol And/Or**, definido en base a las siguientes reglas:

1. Cada nodo representa un problema simple o un conjunto de problemas a resolver.

2. Un nodo representando un problema-primitiva es un nodo terminal.

3. Por cada posible aplicación de un operador a un problema se genera un conjunto de subproblemas de solución alternativa, produciendose un nodo Or, cuya representación es:

En la que los diferentes operadores aplicados sobre P producen los nodos A, B y C con carácter alternativo. Los nodos A, B y C se denominan nodos Or.

4. Cuando la aplicación de un operador genera diversos subproblemas, siendo necesaria la resolución de todos ellos, se produce un nodo And, representado por:

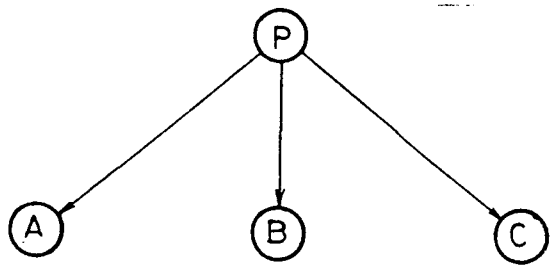


Figura 3

Donde P se descompone en los problemas A, B y C, denominados nodos And.

5. Se puede realizar una unión de las reglas 3 y 4 en el caso de que sólo sea posible aplicar un único operador, como por ejemplo:

Esta regla se denomina de **Nodo Or Intermedio.**

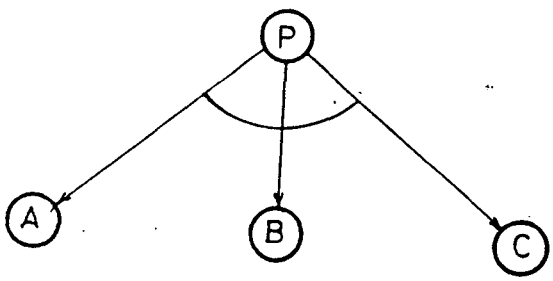


Figura 4

Para la resolución de un problema en forma de árbol and/or, será preciso resolver el nodo raiz. En general, cualquier nodo tendrá solución, es decir es resoluble, si se verifica:

1. Es un nodo primitiva, o:

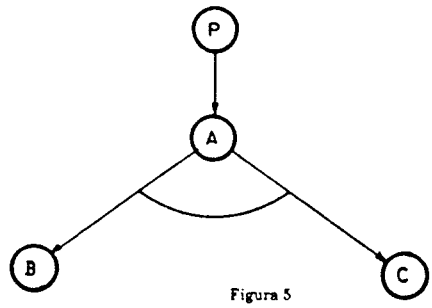


Figura 5

2. Es un nodo no terminal cuyos sucesores son nodos And y todos ellos son resolubles, o:

3. Es un nodo no terminal cuyos nodos sucesores son nodos Or y al menos uno de ellos es resoluble.

Igualmente cualquier nodo será resoluble si se verifica:

1. El nodo no es primitiva y es terminal, o:

2. El nodo posee como sucesores a nodos And y al menos uno de ellos es irresoluble, o:

3. El nodo posee como sucesores a nodos Or y todos ellos son irresolubles.

Un sencillo caso de problema de solución trivial desde el punto de vista de la IA, que admite este tipo de presentación es el de las torres de Hanoi, sea H tal problema para una torre de n elementos:

Tal problema tendrá solución si se resuelven los subproblemas:

- *A: Mover la torre de n-1 elementos desde 1 a 2.*
- *B: Mover un elemento desde 1 a 3.*
- *C: Mover la torre de n-1 elementos desde 2 a 3.*

En tal caso la solución se representará por:

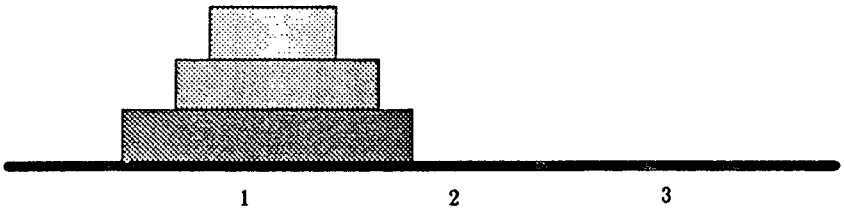


Figura 6

El problema B es **primitiva** y por tanto no precisa una descomposición posterior, mientras que los problemas A y C no son terminales. Tomemos como ejemplo un caso sencillo con n = 3, y representando el problema con el convenio 111→333, indicando el paso de las tres piezas desde 1 hasta 3. El árbol **And/Or** del problema completo estará dado por:

Esta formulación del problema de la Torre de Hanoi está basada en un conocimiento específico de la descomposición

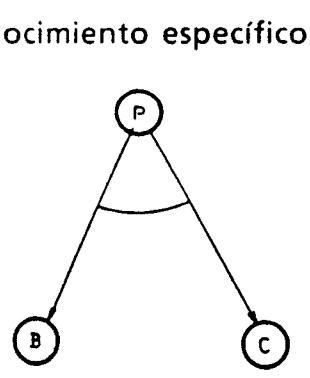


Figura 7

del problema en varios subproblemas, conocimiento que deriva sin duda de un análisis del mismo. Este problema puede plantearse también en el espacio de estados, en cuyo caso no se precisa ningún conocimiento que conduzca a la solución, salvo las leyes que lo rigen. El grafo que interrelaciona los diferentes estados, y a partir del cual se puede construir un árbol de búsqueda, será el siguiente:

Otro clásico ejemplo de "reducción de problema" es el que deriva de la lógica de predicados, por ejemplo:

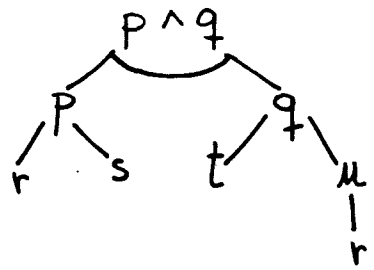


Figura 8

Representación : p, q, r, s, t, u : predicados lógicos.

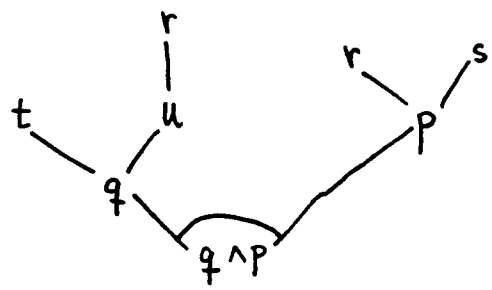
1. $r \vee s \rightarrow p$.

2. $t \vee u \rightarrow q$.

3. $r \rightarrow u$.

4. $r \rightarrow$ primitiva.

Operadores : Teoremas de resolución de cláusulas.



Formulación del problema de la Torre de Hanoi. Figura 9

Objetivo : Demostrar p ^ q.

El árbol And/Or que se deduce por un razonamiento hacia-atrás, es decir, dado el objetivo como nodo raiz aplicar diferentes operadores hasta obtener una verificación, será el siguiente:

Aplicando una estrategia hacia-delante, partiríamos de un predicado cierto y avanzaríamos por aplicación de operadores hasta demostrar el objetivo.

2.1.4. Explosión Combinatoria.

El problema de conducir un sistema hacia un estado que cumpla con los objetivos, puede ser formulado como un problema de búsqueda en un árbol hasta encontrar un nodo asociado con el estado objetivo. Un proceso similar se verifica con la reducción de problemas. El principal problema que se presenta en la resolución de problemas por búsqueda en el espacio de estado reside en la complejidad del problema, definida como los recursos en tiempo y memoria que una máquina precisa para resolver el problema. Por ejemplo en el

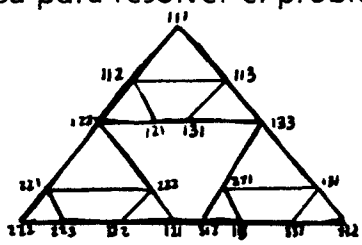
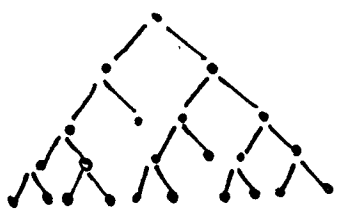


Figura 10

juego del ajedrez se establece que el número de jugadas posibles de una partida es de 10^{120}, aunque sólo un número limitado de ellas conduce a resultados satisfactorios.

Figura 11



En determinados problemas en los que se puede establecer una dimensión del mismo, tal como n, se puede presentar la circunstancia que la cmplejidad del mismo sea exponencial con n, es decir con el tamaño del problema. Este fenómeno de crecimiento desmesurado del coste de la resolución de un problema es lo que se denomina **Explosión Combinatoria**.

Uno de los métodos más eficaces en la reducción del coste reside en visitar sólo un número reducido de nodos en cada proceso de ramificación, basándose en un **conocimiento específico** del problema, expresados generalmente en forma de principios heurísticos, denominándose tal proceso de búsqueda como **búsqueda heurística**, en la cual no se visitan sistemáticamente todos los nodos hijos, sino algunos seleccionados, este proceso también se denomina de "*poda del árbol*". En muchos tipos de problemas este procedimiento puede evitar la explosión combinatoria y convertir en práctica la solución del mismo.

2.2. Estrategias de Control.

Dentro de las estrategias de control contemplaremos los métodos que se utilizan con mayor generalidad para recorre los árboles de búsqueda, con el fin de verificar el objetivo del problema o lo que es lo mismo, dado que el árbol es una abstracción del problema, para determinar la secuencia de operadores que conducen al objetivo. La clasificación de los métodos de control puede realizarse en base a diversos parámetros. Una clasificación puede realizarse en función del carácter definitivo de la exploración de ramas, de tal forma:

Búsqueda irrevocable : en la cual el proceso se realiza a través del árbol descendiendo por los nodos sin posibilidad de replantear las decisiones tomadas.

Búsqueda tentativa : en la cual se realiza una exploración de las ramas del árbol en forma exploratoria posibilitando una Vuelta-Atrás (Backtracking) a algún punto de partida en caso de que la exploración

actual no sea lo suficientemente satisfactoria o no se encuentre la solución al problema.

Otra forma de clasificación de los métodos de búsqueda puede establecerse en función del carácter sistemático o intuitivo de las estrategias de control:

Búsqueda completa : en la cual se realiza una exploración completa, o que asegura la consecución del objetivo, si el problema tiene solución. Esta cualidad de certeza tiene su contrapunto en que tales métodos pueden en determinados casos ser muy costosos, siendo su complejidad de tipo exponencial.

Búsqueda incompleta : el proceso se realiza en función de un conocimiento específico, heurístico, del problema y en base a principios intuitivos o basados en la experiencia pero que carecen en algunos casos de rigor, a pesar de lo cual suelen proporcionar resultados adecuados. Esta desventaja inicial en cuanto a la certeza de los métodos está en muchos casos ampliamente compesada por un ahorro computacional importante.

Otra clasificación que se puede establecer se basa en el grado de precisión de la solución encontrada, según que el método proporcione un camino o un camino óptimo:

Búsqueda de un camino : en estos métodos se obtiene una solución cuando se encuentra un camino al objetivo del problema, independiente de que puedan existir otros.

Búsqueda del camino óptimo : en la que no se plantea la determinación de un camino, sino del camino óptimo, en base a algún criterio específico al problema.

Por último se puede establecer una clasificación en función del tipo de representación del problema:

Búsqueda en el Espacio de Estado : que implicará que la búsqueda se realiza sobre un árbol.

Búsqueda en "Reducción del Problema" : en este caso la búsqueda se realiza sobre un árbol And/Or.

Para ilustrar los diferentes métodos de búsqueda se utlizará como ejemplo el problema de encontrar el camino entre dos nodos de un grafo con ponderación en sus ramas, problema que se asemeja a la determinación de caminos en una red de carreteras que unen diversas ciudades.

En este grafo se define un objetivo consistente en determinar el camino que conduce del nodo S al G. Este problema se puede plantear en el espacio de estado, representando cada nodo un estado en la resolución del problema e interpretando los nodos S y G como los estados inicial y objetivo. Asociado con este grafo tenemos un árbol que coincide con el árbol expandido:

2.2.1. Determinación de un camino.

En esta sección se revisan diversos métodos que determinan un camino hacia el objetivo.

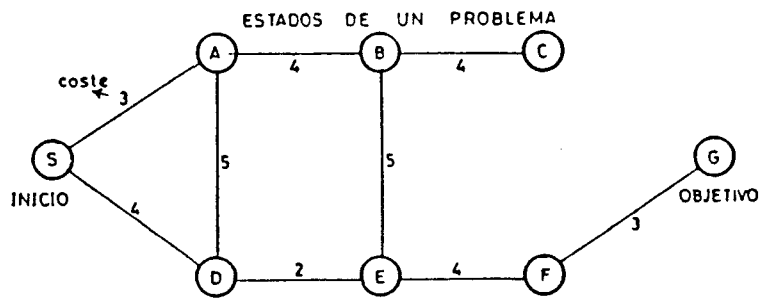
A) Búsqueda a lo profundo (Deep-first).

Este método de búsqueda es de naturaleza *algorítmica*, es decir, sistemático asegurando por tanto que se recorre todo el árbol y que se determinará por tanto un camino si existe solución.

En este procedimiento se visita primero los nodos hijos en dirección a los niveles más bajos del árbol, es de aquí de donde proviene el nombre del método. El procedimiento está

perfectamente descrito en los textos de algorítmica y puede ser descrito utilizando una lista:

Estados de un problema. Figura 12

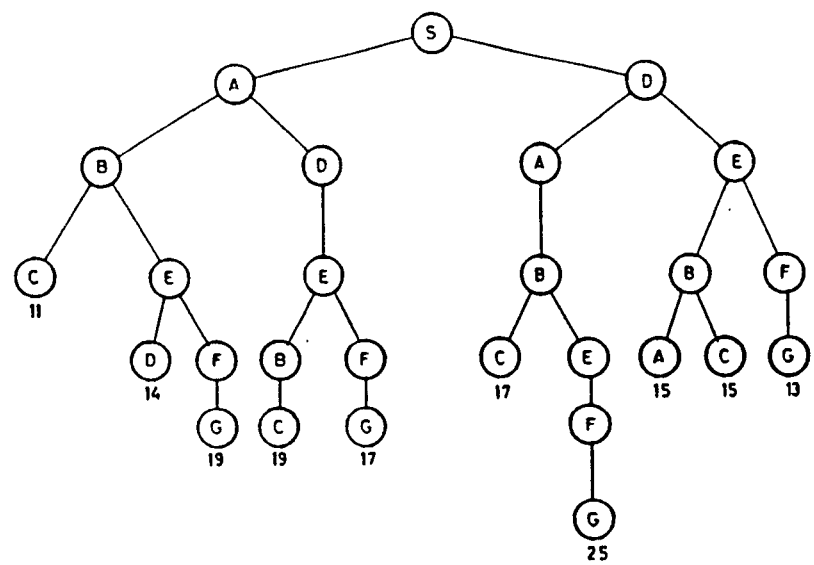


1.- Introducir en la lista el nodo raiz.

2.- Hasta que la lista esté vacía o se encuentre el objetivo, examinar el primer elemento de la lista:

2a.- Si el primer elemento es el objetivo, salir del bucle.

Figura 13



2b.- Si el primer elemento no es el objetivo, eliminar el primer elemento y añadir, en el caso de que existan, los hijos de éste al principio de la lista.

3.- Si se encuentra el objetivo se finaliza el proceso, en caso contrario el problema no tiene solución.

Este método de búsqueda es ciertamente sistemático y asegura el encontrar el camino, pero su utilización puede ser peligrosa en el caso de que existan nodos con descendientes a niveles muy profundos. El tipo de recorrido que se presentaría en nuestro problema sería del tipo:

B) Búsqueda en escalada (Hill-Climbing).

La eficacia en la búsqueda del método anterior puede ser aumentada considerablemente si se dispone de alguna evidencia que nos permita ordenar los nodos hijos, de tal forma que se exploren primero aquellos que presenten mayor expectativa de éxito. Este tipo de selección ordenada de los nodos puede hacerse en función de algún parámetro que resulte de un conocimiento específico del problema. En el ejemplo siguiente supongamos que una estimación heurística del coste restante desde el nodo actual al de destino viene dada por:

- A-G = 10.4
- D-G = 8.9
- E-G = 6.9
- B-G = 6.7
- F-G = 3.0
- C-G = 4.0

Cuanto más coste, hasta el final, presenta un nodo debe suponerse que es menos adecuado como próximo nodo a explorar. Este procedimiento es una simple variación del anterior salvo en que los nodos hijos no se añaden a la lista, tal y como se disponen en el árbol, sino que se añaden de forma

ordenada, introduciendo primero los menos prometedores. El procedimiento se describe como:

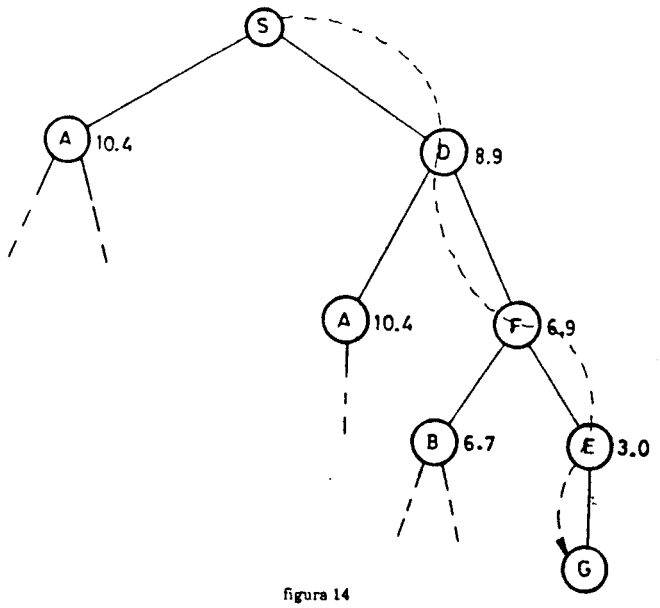


figura 14

1.- Introducir en la lista el nodo raiz.

2.- Hasta que la lista esté vacía o se encuentre el objetivo, examinar el primer elemento de la lista:

2a.- Si el primer elemento es el objetivo, salir del bucle.

2b.- Si el primer elemento no es el objetivo, eliminar el primer elemento y añadir, en el caso de que existan, los hijos de éste al principio de la lista ordenados en base a las expectativas de éxito, introduciendo en último lugar el menos prometedor.

3.- Si el objetivo es encontrado se finaliza el proceso, en caso contrario el problema no tiene solución.

Por ejemplo en el caso del grafo, un parámetro significativo puede ser la distancia en línea recta desde cada nodo al nodo objetivo, resultando el proceso de la figura 15:

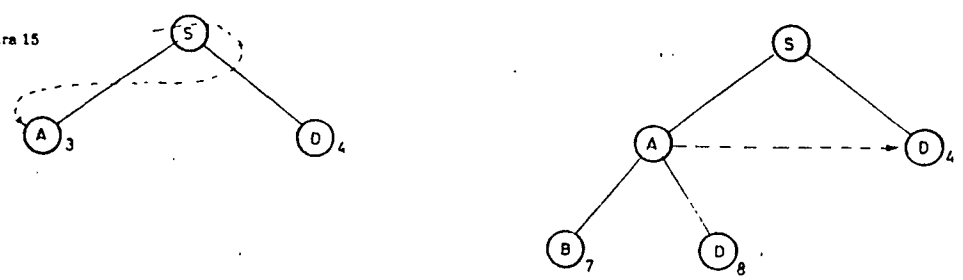
Este método es equivalente a realizar un movimiento en el espacio del parámetro. El nombre del método procede del procedimiento de escalar montañas basandose en tomar en cada punto de la escalada la dirección de máxima pendiente, esto asegura que se asciende lo más rapidamente posible y que se alcanza la cima o meta en el menor tiempo posible. El procedimiento tiene diversas dificultades entre las que se destacan las siguientes:

Los problemas descritos no suelen ser muy frecuentes en sistemas en los que la dimensión del espacio de parámetros heurísticos es escasa. Tales problemas pueden ser muy frecuentes si el número de parámetros es considerable.

C)Búsqueda a lo Ancho (Breadth-First).

Este método se basa en visitar primero todos los nodos de un mismo nivel, antes de visitar a sus nodos hijos. Este método proporciona cierta uniformidad en la búsqueda y conduce a mejores resultados en el caso de que existan nodos con muchos sucesores a niveles muy profundos. La estrategia de búsqueda es semejante a las anteriores salvo en un detalle, los nodos hijos de cada nodo visitado no se añaden al principio de la lista sino al final de la misma. El procedimiento se describe como:

Figura 15

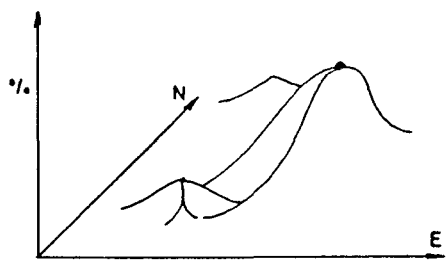


1.- Introducir en la lista el nodo raiz.

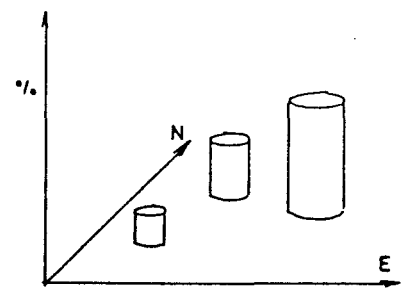
2.- Hasta que la lista esté vacía o se encuentre el objetivo, examinar el primer elemento de la lista:

2a.- Si el primer elemento es el objetivo, salir del bucle.

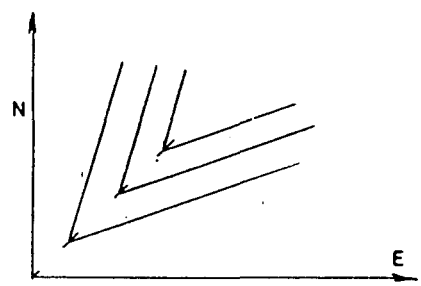
2b.- Si el primer elemento no es el objetivo, eliminar el primer elemento y añadir, en el caso de que existan, los hijos de éste al final de la lista.



Existencia de varios máximos locales (Foothill problem), lo cual sólo asegura una optimización local pero no global.



Existencia de áreas planas (Plateau problem), lo que implica la no existencia de direcciones privilegiadas que conduzcan con seguridad al máximo.



Existencia de aristas (Ridge problem) o discontinuidades en la pendiente, provocando una optimización muy local.

Figura 16

3.- Si el objetivo se ha encontrado finaliza el proceso, en caso contrario el problema no tiene solución.

Esta estrategia es adecuada cuando se presenten árboles con profundidades desiguales y las soluciones se encuentren a escasa profundidad, en cambio será poco eficaz en el caso de árboles bastante equilibrados. En el ejemplo descrito el recorrido se realizará de la forma:

D) Búsqueda por Haces (Beam search).

Este procedimiento es una derivación del caso anterior y se basa por tanto en una exploración por niveles, pero en este caso no se desciende por todas las ramas, sino por un conjunto de ellas seleccionado o "*haz*", ignorándose el resto. La selección de las ramas adecuadas se realiza en base a criterios heurísticos específicos al problema produciéndose una especie de "*poda de ramas*" poco prometedoras. Esta estrategia permite tener acotado, de un modo considerable, el número de nodos que se visitan en el proceso, evitando en cierta forma la posibilidad de una *Explosión Combinatoria.*

En el ejemplo de trabajo tomando como parámetro heurístico la distancia en línea recta al nodo objetivo y limitando a dos el número de ramas sin "podar" se producirá el siguiente resultado:

E) Búsqueda Primero-el-Mejor (Best-first S.)

Esta estrategia se asemeja a la de Hill-Climbing salvo en el detalle de que la exploración no continua forzosamente por el mejor nodo hijo del nodo actual, sino por el mejor nodo hijo de los presentes en la lista.

El camino encontrado por este procedimiento parece más obvio que en el resto de procedimientos porque se actua

sobre el nodo más idoneo o más cercano al objetivo, siendo la ramas más prometedoras las seleccionadas por el elemento heurístico. En cualquier caso, la eficacia del procedimiento estará en función de la idoneidad del elemento heurístico empleado. La estrategia se describe como:

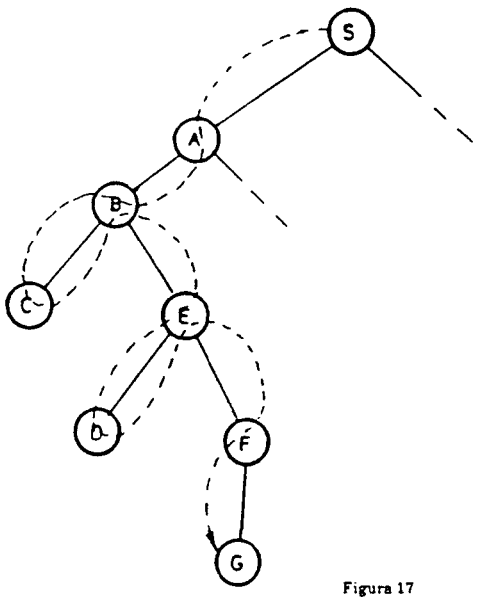


Figura 17

1.- Introducir en la lista el nodo raiz.

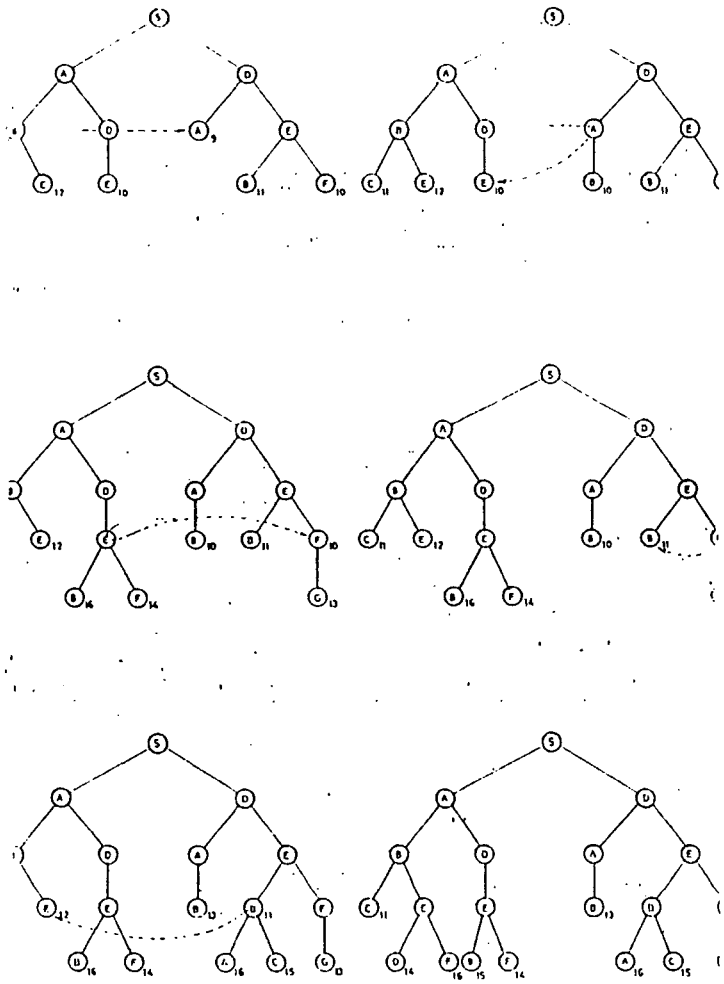
2.- Hasta que la línea esté vacía o se encuentre el objetivo, examinar el primer elemento de la lista:

2a.- Si el primer elemento es el objetivo, salir del bucle.

2b.- Si el primer elemento no es el objetivo, eliminar el primer elemento y añadir, en el caso de que existan, los hijos de éste *a la lista ordenando la lista entera en base a las expectativas de éxito situando en primer lugar el elemento más prometedor.*

3.- Si el objeto se ha encontrado el proceso finaliza, en caso contrario el problema no tiene solución.

El papel de la heurística en las estrategias de control que hemos expuesto se puede concretar en dos aspectos:



- Determinación de qué nodos deben ser visitados en primer lugar, en función de ser los de máxima espectativa de éxito. Este proceso implica la tarea de ordenar los nodos en la lista de nodos pendientes de ser visitados, por esta causa se suelen denominar como Búsqueda Ordenada.

- Realización de una "Poda" selectiva del ancho del árbol en cada nivel manteniendo un *ancho de búsqueda* constante o restringido.

2.3. Búsqueda del Camino Optimo.

En esta sección se expondrán algunas estrategias para la determinación de caminos óptimos. En principio el método más sencillo, y por supuesto más costoso, de determinación del camino óptimo será determinar todos los caminos, y elegir el más adecuado. Esta estrategia es demasiado costosa en la mayoría de las aplicaciones y no se considera como una estrategia razonable. Existen otras estrategias de determinación del camino óptimo sin precisar un coste tan elevado.

A) Búsqueda Expansión-y-Salto.

Esta estrategia se basa en utilizar la información que proporciona una función de coste, o distancia, acumulado desde el inicio o raiz hasta cada nodo. El método consiste, en líneas generales, en mantener un conjunto limitado de nodos que determinan caminos incompletos y en expandir, o ramificar, el nodo de trabajo actual con todos sus nodos hijos, calculando la función de coste acumulada para los hijos y posteriormente seleccionar el nodo de menor coste calculado de entre todos los nodos ya visitados, convirtiendo el nodo seleccionado en el de trabajo. Este proceso se repite hasta dar con el nodo objetivo. Una descripción más precisa de la estrategia es la siguiente:

1.- Formar una lista de caminos parciales. Inicializar tal lista con un camino de longitud nula que comienza en la raiz del árbol.

2.- Hasta que la lista esté vacía o se encuentre el objetivo, analizar el primer camino en la lista:

2a.- Si el camino termina en el nodo objetivo, se finaliza el bucle.

2b.- Si el primer camino no termina en el nodo objetivo:

- **2b1.** *Eliminar este primer camino de la lista.*
- **2b2.** *Formar nuevos caminos a partir del camino eliminado, añadiendo los nodos hijos del nodo final del camino eliminado.*
- **2b3.** *Añadir estos nuevos caminos a la lista.*
- **2b4.** *Ordenar la lista de caminos en base al costo acumulado de cada uno, colocando el de mínimo coste al inicio de la lista.*

3.- Si se alcanza el nodo objetivo, el problema tiene solución y se determina el camino óptimo, en caso contrario no existe solución.

El coste computacional de este procedimiento en algunos casos puede ser bastante elevado dado que en muchos casos se pueden llegar a recorrer bastantes caminos. A continuación se muestra el recorrido a que daría lugar esta estrategia en el ejemplo de espacio de estado que se utilizó en ejemplos anteriores. Se utiliza como función de coste la ponderación o distancia entre nodos.

B) Búsqueda Expansión-y-Salto con Subestimación.

El método de expansión y salto puede muy mejorado, utilizando una estimación del coste, o distancia total al nodo objetivo a través del camino en estudio. Para ello se supone que el coste total será de la forma:

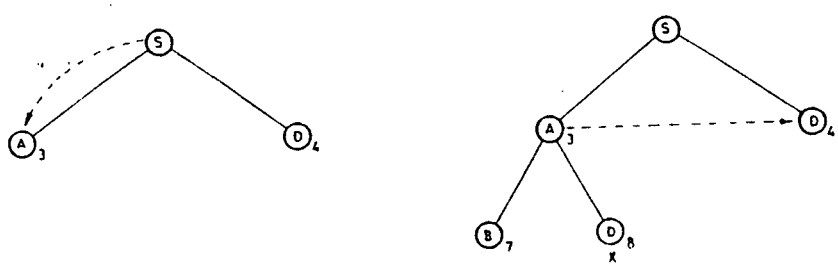
$$f^*(\text{total del camino}) = g(\text{ya recorrido}) + h^*(\text{restante}).$$

estimación conocido estimación

Si $h = 0$ este método coincide con el anterior.

En donde g (ya recorrido) es el coste acumulado hasta el punto de trabajo y que es conocido, h* (restante) es una estimación del coste restante hasta el nodo objetivo y que por tanto es desconocida. Esta estimación del coste restante se establecerá en base a algún conocimiento específico del problema. La estrategia combina por tanto un conocimiento determinista del problema, procedente de la historia anterior reflejada en los caminos recorridos, con un conocimiento heurístico reflejado en la estimación del coste restante. Se supone que todos los costes tanto reales como estimados son positivos.

Puede correrse el riesgo de ignorar permanentemente, es decir, no expandir un nodo en el camino óptimo si a tal nodo se le asignara un coste superior al que realmente tiene, puesto que un camino no óptimo podrá mantener un coste total, acumulado más estimado, inferior al del camino óptimo, en este caso cualquier camino no óptimo presentará siempre en algún momento un coste total superior al óptimo, por esta causa es preciso incluir una subestimaciones del coste restante, para prevenir la eliminación del verdadero camino óptimo. La



descripción exacta del procedimiento es la siguiente:

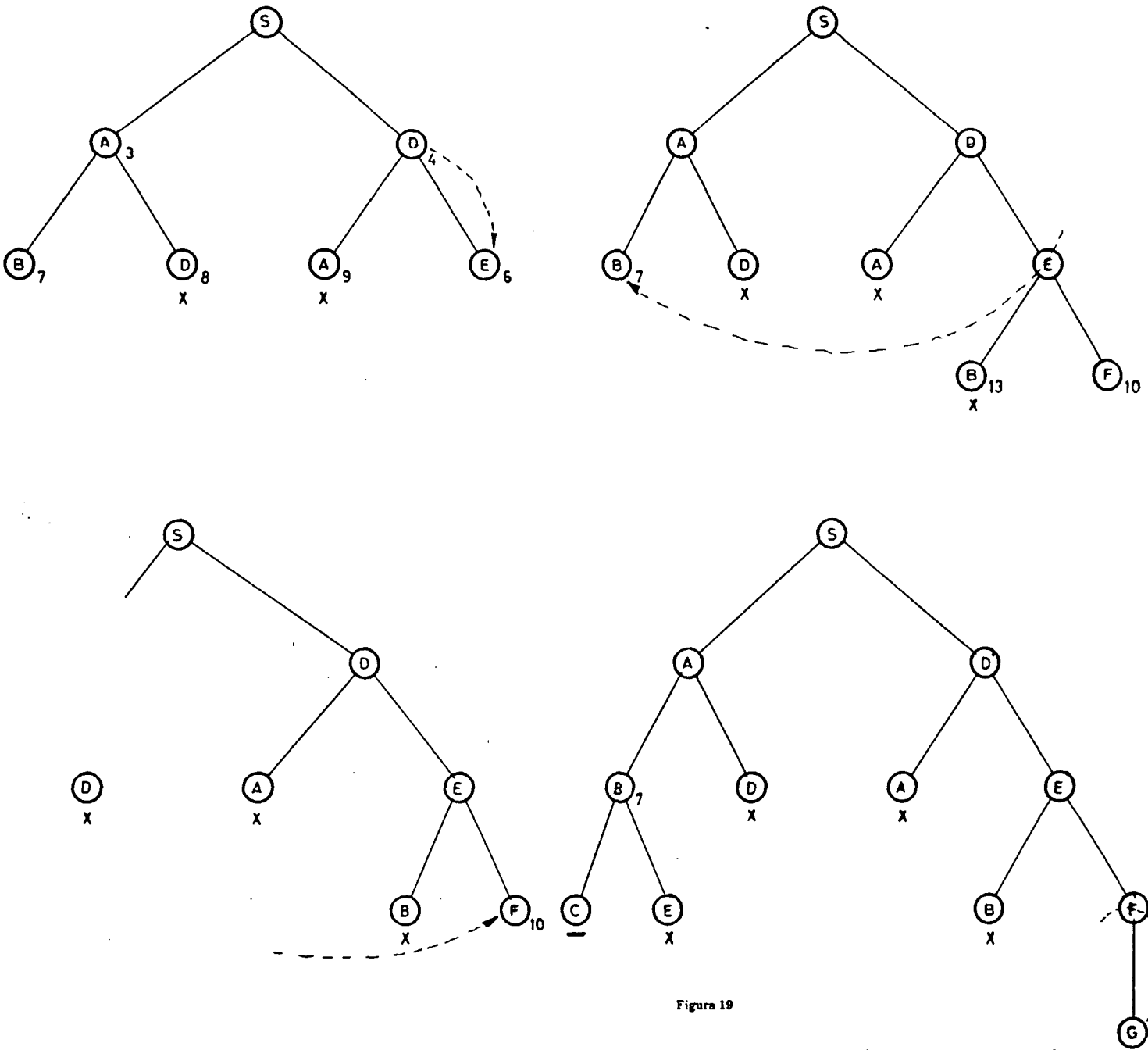


Figura 19

1.- Formar una lista de caminos parciales. Inicializar tal lista con un camino de longitud nula que comienza en la raiz del árbol.

2.- Hasta que la línea esté vacía o se encuentre el objetivo, analizar el primer camino en la lista:

2a.- Si el camino termina en el nodo objetivo, se finaliza el bucle.

2b.- Si el primer camino no termina en el nodo objetivo:

- **2b1.** Eliminar este primer camino de la lista.
- **2b2.** Formar nuevos caminos a partir del camino eliminado añadiendo los nodos hijos del nodo final del camino eliminado, si los tiene.
- **2b3.** Añadir estos caminos nuevos a la lista.
- **2b4.** Ordenar esta lista de caminos en base a *la suma del coste acumulado y una subestimación del coste restante* para cada uno, colocando el de mínimo coste al inicio de la lista.

3.- Si se alcanza el nodo objetivo, el problema tiene solución y se determina el camino òptimo, en caso contrario no existe solución.

En el ejemplo propuesto utilizando como medida estimativa la distancia en línea recta, el proceso de búsqueda será:

C) Búsqueda Expansión-y-Salto con Programación Dinámica.

Esta estrategia se basa en la utilización del principio de Programación Dinámica el cual establece que en un camino óptimo todos los subcaminos son también óptimos. Por tanto, si en la lista de caminos recorridos hasta la situación actual del problema nos encontramos con varios caminos que tienen el último nodo común, significará que sólo el más óptimo de los mismos podrá llegar, si acaso, a ser el camino óptimo del problema, por tanto es posible eliminar los caminos de la lista que tengan el mismo nodo final y que el resultante no sea el más óptimo de entre los que verifican esta propiedad. La descripción exacta del procedimiento es la siguiente:

1.- Formar una lista de caminos parciales. Inicializar tal lista con un camino de longitud nula que comienza en la raiz del árbol.

2.- Hasta que la lista esté vacía o se encuentre el objetivo, analizar el primer camino en la lista:

2a.- Si el camino termina en el nodo objetivo, se finaliza el bucle.

2b.- Si el primer camino no termina en el nodo objetivo:

- **2b1.** Eliminar este primer camino de la lista.
- **2b2.** Formar nuevos caminos a partir del camino eliminado añadiendo los nodos hijos del nodo final del camino eliminado, si tiene descendientes.
- **2b3.** Añadir estos nuevos caminos a la lista.
- **2b4.** Ordenar la lista de caminos en base al coste acumulado de cada uno, colocando el de mínimo coste al inicio de la lista.

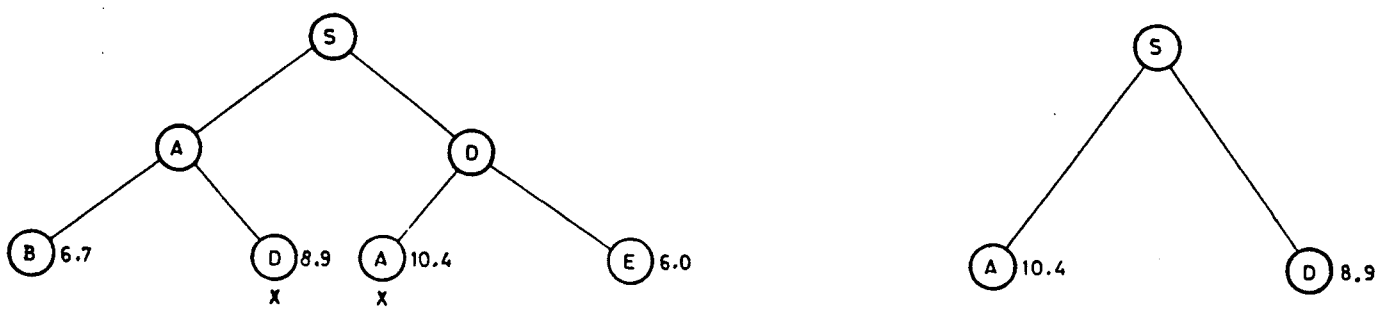


Figura 20

- **2b5.** Si dos o más caminos acaban en un nodo común, borrar los mismos, excepto el que posee mínimo coste de entre ellos.

3.- Si se alcanza el nodo objetivo, el problema tiene solución y se determina el camino óptimo, en caso contrario no existe solución.

En el problema ejemplo se producirá el siguiente proceso de búsqueda:

D) Procedimiento A*.

El procedimiento A* es una combinación del método **Ramificación-y-Salto** y los métodos de **Subestimación** y de **Programación Dinámica**, incorporando por tanto todas sus ventajas. Se describe de la siguiente forma:

1.- Formar una lista de caminos parciales. Inicializar tal lista con un camino de longitud nula que comience en la raiz del árbol.

2.- Hasta que la lista esté vacía o se encuentre el objetivo, analizar el primer camino en la lista:

2a . Si el camino termina en el nodo objetivo, se finaliza el bucle.

2b . Si el primer camino no termina en el nodo objetivo:

- **2b1.** Eliminar este primer camino de la lista.
- **2b2.** Formar nuevos caminos a partir del camino eliminado añadiendo los nodos hijos del nodo final del camino eliminado.
- **2b3.** Añadir tales nuevos caminos a la lista.

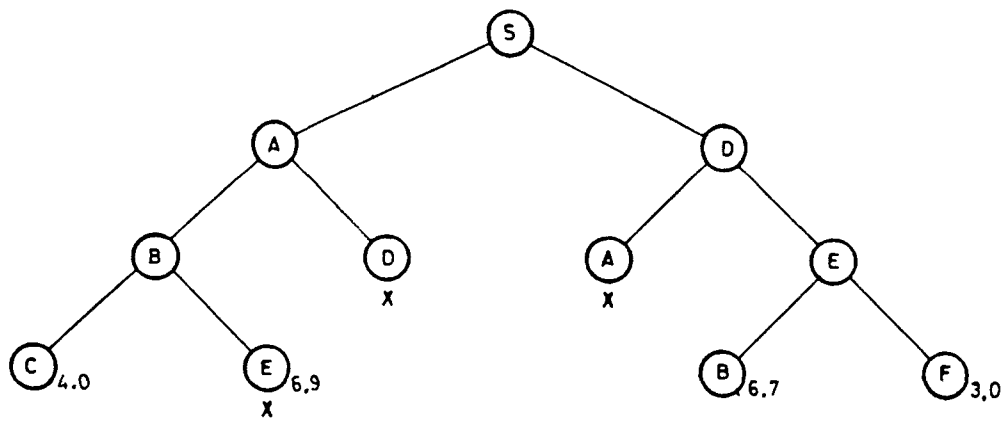
- 2b4. Ordenar la lista de caminos en base a *la suma del coste acumulado y una subestimación del coste restante* para cada uno, colocando el de mínimo coste al inicio de la lista.

- 2b5. *Si dos o más caminos acaban en un nodo común, borrar los mismos, excepto el que posee mínimo coste.*

3.- Si se alcanza el nodo objetivo, el problema tiene solución y se determina el camino óptimo, en caso contrario no existe solución.

El procedimiento **A*** ha sido ampliamente analizado y con muy interesantes propiedades. Consideremos la función coste para cada nodo **n** expresada por:

Figura 21



$$f^*(n) = g^*(n) + h^*(n)$$

En donde **g* (n)** representa la estimación del coste dado hasta el nodo **n**, coincidente con el coste real dado **g (n)**, **h* (n)** es una estimación del coste **h (n)** desde el nodo **n** hasta el objetivo o coste restante.

Del procedimiento **A*** puede derivarse como caso límite el procedimiento anterior cuando se verifica que **h* (n) = 0**.

La función **h* (n)** está ligada con un conocimiento heurístico del problema, dado que esta función puede variar entre cero y su límite superior **h (n)**, se dice que el procedimiento está más informado del problema real cuanto mayor sea el valor de **h* (n)**, siempre inferior al real.

Otra importante propiedad que cumple este procedimiento es la de **Optimalidad** según la cual si el procedimiento **A1** está más informado que **A2**, aplicados al mismo árbol y siendo ambos admisibles, que equivale a, **h1*(n) > h2*(n)**, entonces **A1** nunca expande un nodo que no haya sido expandido por **A2**. Esto quiere decir que los procesos menos informados expandirán más nodos y que el proceso con mayor información posible - en el cual se verifica que **h*(n) = h(n)** - será el que menos nodos expanda de entre todos, ya que es el nodo óptimo.

Para finalizar con este procedimiento de control se hará notar que en muchos casos puede ser ser costoso estimar adecuadamente **h*(n)**, por lo cual es preciso considerar este costo computacional durante toda la evaluación, y puede ocurrir que sea preferible encontrar un camino no óptimo con coste total más reducido, que determinar el camino óptimo, ya que éste exigirá mayor coste de cómputo.

2.4. Búsqueda en árboles And / Or.

En contraposición con la búsqueda en el **espacio de estados**, que utilizan con mayor frecuencia el razonamiento **hacia-delante**, la búsqueda según la **reducción de problemas** utiliza el **razonamiento hacia-atrás**. La principal diferencia que se presenta en la búsqueda en árboles **And/Or** estriba en la presencia de nodos **And**, ya que en este caso es preciso

recorrer, o resolver, todos los nodos **And** descendientes directos de un mismo nodo. En la resolución de nodos **And** puede presentarse un problema derivado de la posible interdependencia de los problemas asociados con los mismos. Así, si tales problemas son independientes, se pueden resolver en el orden que se desee, si por el contrario los problemas son dependientes, será preciso incluir una estrategia de control que se adapte a tal dependencia, estrategia que es ajena a la de búsqueda general.

En las búsquedas relativas al espacio de estado la condición de terminación del proceso estaba determinada por el hallazgo del nodo objetivo, mientras que en este tipo de búsqueda la condición de finalización estará regida por la resolución, o no, del nodo raiz.

Debido a que las reglas de resolución de nodos se establecen en función de la resolución de los nodos hijos y dado que el resultado final del proceso será la resolución de la raiz, será preciso mantener un puntero en cada nodo dirigido hacia su nodo padre para inferir la solución del mismo una vez hecho lo propio con el hijo y repitiendo este proceso hasta inferir la relación con la raiz. Este proceso de **vuelta-atrás** está relacionado con el hecho de que los nodos descendientes carecen de un significado concreto, siendo pasos obligados en la resolución de la raiz. Esta circunstancia no se presenta en el espacio de estado, puesto que allí cada nodo tiene un significado concreto que se corresponde con un estado en la evolución del problema, y el nodo final tiene un significado preciso que no tienen los nodos terminales en los árboles **And/Or**.

A continuación se expondrán diversos procedimientos para realizar la búsqueda en árboles **And/Or**, suponiendo

siempre la independencia de los problemas. En líneas generales son adaptaciones de sus equivalentes en árboles.

A) Búsqueda a lo Ancho And/Or.

Este procedimiento realiza una búsqueda recorriendo primero los diversos niveles, por tanto la solución, o no, del problema se establecerá en función al camino menos profundo. El procedimiento se puede describir por:

1.- Poner el nodo raiz en una lista de nodos no expandidos, que inicialmente esté vacía.

2.- Hasta que se determine la resolubilidad o irresolubilidad del nodo raiz, analizar el primer nodo n de la lista:

2a . Eliminar n de la lista.

2b. Expandir n, generando todos sus sucesores directos incorporando a cada nodo generado un puntero hacia su antecesor.

- **2b1.** Para todos los nodos generados:

  * Etiquetar todos los nodos generados como resolubles.
  * Si de la solución de cada nodo terminal se deduce, por las reglas definidas, la solución de sus antecesores, etiquetar los mismos como resolubles.
  * Si el nodo raiz resulta resoluble finalizar con éxito.
  * En caso contrario añadir todos los nodos generados al final de la lista.

* Eliminar de la lista todos los nodos resolubles y sus descendientes.

- **2b2.** Si no se genera ningún nodo en la expansión:

* El nodo n es irresoluble.

* Si de la irresolubilidad de n se deduce la de sus antecesores, etiquetar a éstos como irresolubles.

* Si el nodo raiz resulta irresoluble, finalizar con fallo.

* En caso contrario eliminar de la lista todos los nodos irresolubles y sus descendientes.

3.- El proceso finaliza con resolución del problema o con la declaración de su irresolubilidad.

B) Búsqueda a lo profundo And/Or.

El procedimiento es similar al descrito anteriormente salvo que los nodos generados y aún no expandidos se añaden al principio de la lista:

1. Poner el nodo raíz en una lista de nodos no expandidos, inicialmente vacía.

2. Hasta que se determine la solubilidad o insolubilidad del nodo raiz, analizar el primer nodo, n, de la lista:

2a. Eliminar n de la lista.

2b. Expandir n, generando todos sus sucesores directos incorporando a cada nodo generado un puntero hacia su antecesor.

2b1. Para todos los nodos generados:

* Etiquetar los nodos terminales como resolubles

* Si de la solución del mismo se deduce la solución de sus antecesores, etiquetar los mismos como resolubles.

* Si el nodo raíz resulta resoluble finalizar con éxito.

* En caso contrario añadir todos los nodos generados al **principio** de la lista.

* Eliminar de la lista todos los nodos resolubles y sus descendientes.

2b2. Si no se genera ningún nodo en la expansión:

* El nodo n es irresoluble.

* Si de la irresolubilidad de n se deduce la de sus antecesores,etiquetar a éstos como irresolubles.

* Si el nodo raíz resulta irresoluble, finalizar con fallo.

* En caso contrario eliminar de la lista todos los nodos irresolubles y sus descendientes.

3. El proceso finaliza con resolución del problema o con declaración de su irresolubilidad.

Tanto en este procedimiento como en el anterior se puede ahorrar un nivel de búsqueda en determinados casos si se eliminan los nodos **Or** intermedios, así el paso **2b** puede ser sustituido por:

2b. Expandir **n**, generando todos sus sucesores directos, si algún **m** de los mismos genera nodos **And**, expandir **m**, incorporando a cada nodo generado un puntero hacia su antecesor **n**.

C) Exploración Exhaustiva And/Or.

En este procedimiento se muestra un método para determinar todas las formas posibles de resolver el nodo raiz. Denominaremos un camino a un sub-arbol del árbol And/Or original, tal que permite la demostración del nodo raiz. En un camino sólo existirán descendientes and, y en los nodos con

descendientes or, sólo aparecerá uno de ellos. En el siguiente ejemplo se muestran varios caminos de un árbol And/Or:

Un procedimiento de exploración que construye todos los posibles caminos, es el siguiente:

1.- Formar una lista de caminos, inicialmente vacía, y añadir el nodo raiz.

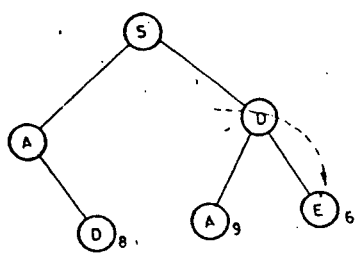
2.- Hasta que la lista de caminos esté vacía, hacer:

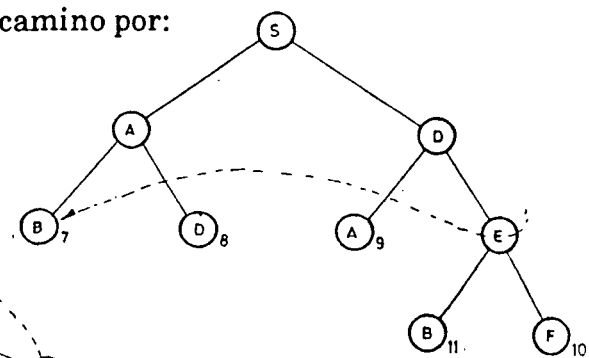
2a . Repetir para el primer camino:

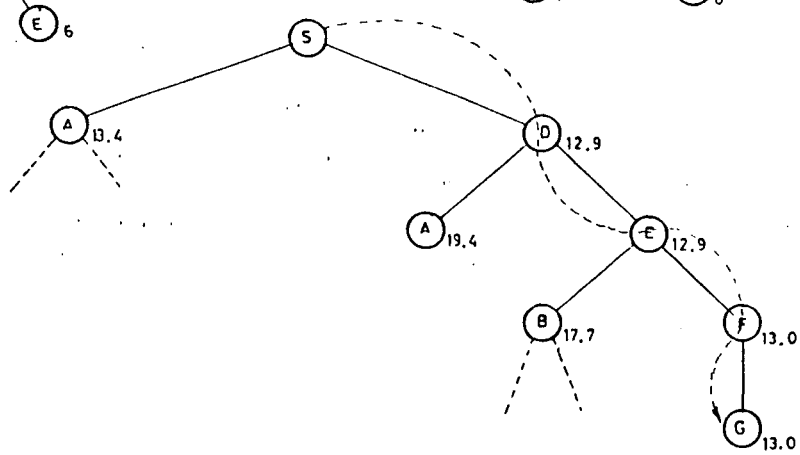
2a1. Si el primer nodo no visitado es terminal, entonces considerar el mismo nodo como visitado.

- **2a2.** Si el primer nodo no visitado no es terminal y no tiene sucesores, entonces este camino no es válido y se elimina de la lista.
- **2a3.** Si el primer camino no tiene nodos no visitados, entonces este camino es una solución y se elimina de la lista.
- **2a4.** Si el primer nodo no visitado tiene sucesores, entonces se sustituye este camino por:

b) Sucesores Or.







3. Sistemas de Producción.

3.1. Introducción al cálculo de Predicados.

Por **Sistemas de Producción** o **Sistemas basados en Reglas,** se entiende, en el ámbito de la **Inteligencia Artificial,** a los sistemas que incorporan bases de datos o estructuras de control que se fundamentan en el Cálculo de Predicados y en sus métodos de **Resolución.** En general, se pueden considerar como sistemas de producción, a aquellos cuya descripción del conocimiento o estrategia de control se basan en mecanismos del tipo:

IF condición1 AND condición2 ... THEN consecuencia.

La utilización de estructuras de este tipo, tanto para especificar y resolver problemas, como para representar conocimientos de algún ámbito, presenta las siguientes ventajas:

* Simplicidad conceptual de la estructura.

* **Modularidad**, si se considera un sistema compuesto de esta estructura simple.

* **Proximidad** al modo de concebir el razonamiento humano.

* **Estructuración**, dado que condición-i y consecuencia pueden ser estructuras complejas, pudiéndose de esta forma construir sistemas que incorporan jerarquías conceptuales.

Sin embargo no todas las propiedades de los sistemas de producción pueden considerarse como ventajas. Algunos esquemas de representación de problemas o conocimientos no se adaptan fácilmente a esta estructura, pero la mayor parte de ellos si que lo hacen, de tal forma que la utilización de la estructura IF-THEN simplifica considerablemente la programación y construcción de sistemas de información en el ámbito de la IA.

Dado que los sistemas de producción se basan en la utilización de **predicados lógicos**, y el control en tales sistemas se fundamenta en los **métodos de resolución** o **demostración de teoremas lógicos**, se procederá a una breve descripción de los elementos en **Cálculo de Predicados**.

El **cálculo de predicados** es una herramienta formal utilizada por la *Matemática* y por tanto en nuestro contexto de IA sólo se considerarán sus *aspectos fundamentales*. El **cálculo de predicados de primer orden**, que es el que nosotros consideramos, puede ser contemplado como un **lenguaje formal**, cuyo discurso son las **fórmulas**. Las fórmulas se componen de elementos más simples o fórmulas sencillas, que denominaremos **átomos**, y de operadores o **conectores**, que los unen para formar fórmulas más complejas.

3.1.1 Fórmulas Atómicas.

Las fórmulas atómicas, o átomos, son los componentes básicos de las fórmulas. Para la construcción de éstas consideraremos los siguientes componentes elementales del lenguaje del cálculo de predicados:

- Símbolos **Predicados**.
- Símbolos **Variables**.
- Símbolos **Constantes.**
- Símbolos Funciones o **Functores**.

Un **símbolo predicado** es un símbolo cuyo valor se encuentra en el dominio lógico, es decir "*Cierto o Falso*" en un contexto de relaciones entre objetos o entidades. A estos objetos se les denomina **términos**, pudiendo ser de tres tipos: **Constantes, Variables** y **Functores**.

Un **símbolo constante** representa un término con un valor semántico determinado en el contexto del problema o situación. Así, por ejemplo, la fórmula atómica siguiente:

pintó (picasso, el_guernica).

Expresa el hecho cierto de que "picasso pintó el guernica". Tanto "picasso" como "el_guernica", representan símbolos constantes y "pintó" es un predicado lógico. Este ejemplo de predicado posee, por su propia semántica, dos términos: *el pintor y el cuadro*.

Un **símbolo variable** representa un *término* cuyo valor no está inicialmente determinado. El dominio de las variables debe coincidir con el de las constantes. En este contexto, y para avanzar en coherencia con temas posteriores, se supondrá que las variables son strings que comienzan con

mayúscula, mientras que las constantes comienzan con minúscula. En el siguiente ejemplo aparece una variable:

pintó (Quien, el_guernica).

"Quien" representa una variable, pintor, de forma que en este contexto deberia asumir el valor "picasso". Este proceso se denominará más adelante **Unificación**.

Un **símbolo función** o **functor** es un símbolo que posee un valor en el dominio de las constantes y variables, pero que no es ni una variable ni una constante. En el ejemplo siguiente:

pintó (el_mejor (pintor, españa), el_guernica).

"el_mejor" representa un functor de los términos "pintor" y "españa", refiriéndose al pintor de "el_guernica". A pesar de que un functor tiene la misma estructura que un predicado, no se puede considerar como tal, al menos en el dominio de la *Lógica de Primer Orden*, que es la que estamos considerando. Se denomina **Universo del Discurso**, *al conjunto de constantes, functores y predicados que describen una semántica.*

Esta concepción de los predicados lógicos tiene una relación inmediata con el discurso humano, causa ésta por la que representa un mecanismo formal que nos es muy próximo. Para pasar de una sentencia a un predicado lógico, se deben ordenar primero los conceptos involucrados. En

general el predicado se corresponderá con el verbo de la sentencia y los términos con el sujeto y objeto. Generalmente es posible expresar un mismo concepto semántico en diversas formas, así la frase "la escuela es blanca", expresa un contenido o concepto que puede expresarse por las formulas siguientes:

blanca (escuela).

es (escuela,lanca).

color (escuela,blanca).

es (color,casa,blanca).

Correspondientes a distintas elecciones del predicado y términos. Cualquier elección es correcta, siempre que sea coherente con el significado o semántica. En el cálculo de predicados cada fórmula debe poseer una interpretación. Tal interpretación se obtiene asignando una correspondencia entre los elementos del lenguaje y las relaciones, entidades y funciones del dominio del discurso. Para cada predicado se debe asignar una relación, para cada constante una entidad y para cada functor una función en el dominio. Estas asignaciones representan la semántica del lenguaje de cálculo de predicados:

En la elección de la representación deberá tenerse en cuenta que cualquier término puede ser considerado una variable. Así la formula:

pintor (Escritor, Libro).

Representa el dominio de todos los pintores y sus cuadros. Los predicados no pueden considerarse como

variables, al menos en la Lógica de Primer Orden. Podemos clasificar las lógicas en:

* **Lógica de orden cero:** *Los predicados carecen de términos, y el cálculo de predicados se reduce al cálculo de proposiciones o proposicional.*

* **Lógica de orden uno:** *Los predicados poseen términos, que pueden ser constanteso variables.*

* **Lógica de orden superior:** *los predicados lógicos pueden ser en si mismos variables.*

3.1.2 Conectores.

Son elementos *operadores* que permiten construir fórmulas más complejas a partir de los *átomos*. Clásicamente se consideran los siguientes conectores:

Conector	Sintaxis	Significado
Negación	¯p	n o p
Disyunción	p v q	p ó q
Conjunción	p ∧ q	p y q
Implicación	p → q	si p entonces q
Equivalencia	p ⇔ q	p equivalente a q

La **negación** como es conocido genera una fórmula que posee valores lógicos complementarios de la fórmula sobre la que se aplica. Estrictamente no es un conector, ya que no conecta dos fórmulas, sino que transforma una.

¯escribió (cervantes, hamlet).

La **disyunción** genera una fórmula que posee por valor el "*or*" lógico de los valores de cada componente.

está (semaforo, rojo) v está (semaforo, verde).

La **conjunción** genera una fórmula que posee el valor "*and*" de los valores de cada componente.

$$\text{color (coche, rojo)} \wedge \text{color (cielo, azul)}.$$

La **implicación** genera una fórmula compuesta de dos componentes, el primero se denomina antecedente o condición, mientras que el segundo se denomina consecuente o conclusión. El valor de la fórmula implicación se establece en base a la siguiente tabla de verdad:

Es decir, la implicación es cierta si el antecedente es falso o son ciertos ambos, consecuente y antecedente.

La **equivalencia** o doble implicación genera una fórmula que es cierta sólo si los dos componentes poseen idéntico valor.

El significado semántico de la equivalencia es idéntico al del "nor-exclusivo".

El significado de los conectores es idéntico que en el cálculo proposicional, verificándose las propiedades : **Complemento, Doble negación, Conmutativa, Distributiva, Asociativa** y las **Leyes de Morgan**.

3.1.3 Cuantificadores.

Los cuantificadores establecen el entorno de existencia de las variables. Se consideran dos tipos de cuantificadores; el **Existencial** y el **Universal**.

El cuantificador **Universal** se expresa por (∀X), acompañando a una fórmula en X, de tal modo que establece que la fórmula es siempre cierta para cada valor posible de X.

La variable X se dice que está cuantificada por ∀. Así (∀X) p (X), establece que el predicado p es cierto para todos los valores de X.

$$(\forall X)\ (\text{cuerpo}(X)\ \Lambda\ \text{posee}(\text{masa}, X)\ \Rightarrow\ \text{cae}(X)).$$

La anterior fórmula establece, en forma de cálculo de predicados el hecho de que "*todos los cuerpos que poseen masa caen*".

El otro cuantificador que consideramos es el **Existencial,** que se expresa por (∃X) p (X) de tal forma que establece que como mínimo existe un valor de la variable X que hace cierto el predicado p.

$$(\exists X)\ (\textit{descubrió}\ (X, \text{américa})).$$

La fórmula anterior establece en forma de cálculo de predicados que "*alguien descubrió américa*".

El cuantificador existencial define un dominio restringido sobre la variable X, mientras que el universal establece un dominio en todo el discurso. En el cálculo de predicados de primer orden los cuantificadores no pueden aplicarse más que a las variables, no a los predicados, así la fórmula:

$$(\forall p)\ P(\text{ordenadores}).$$

Describe a todos los predicados que son ciertos para los ordenadores, pero esta fórmula no es válida en lógica de primer orden. Denominaremos **variables cuantificadas** a aquellas que son afectadas por los cuantificadores, y **variables libres** a las no afectadas.

3.1.4 Fórmulas Bien Formadas.

Por fórmula bien formada, *fbf*, se designa a aquellas que cumplen ciertos requisitos o reglas, establecidos en función de su estructura. Estas reglas definen la sintaxis del lenguaje del cálculo de predicados. Se dice que una fórmula es una fórmula bien formada si puede derivarse de las siguientes reglas de formación:

- Cualquier fórmula atómica es una fbf.
- Si p y q son fbf, entonces también son fbf las siguientes:

  $\neg p$.
  $p \vee q$.
  $p \wedge q$.
  $p \rightarrow q$.

- Si X es una variable libre en p, entonces también son fbf las siguientes:

  $(\forall X)\ p(X)$.
  $(\exists X)\ p(X)$.

- Cualquier fórmula que no pueda formarse a partir de estas reglas no es una fbf.

Una **subfórmula**, de una fórmula r, se define siguiendo las reglas:

* Cualquier fórmula r es una subfórmula de sí misma.

* Si r es una fórmula formada a partir de p y q en los casos anteriores, entonces, cualquier subfórmula de p ó q es una subfórmula de r.

Una fórmula se dice que es una sentencia, si no posee ninguna de sus variables libres. Así la siguiente fórmula es una sentencia:

$$(\exists X)\,(\,p(X) \wedge (\exists X)\,q(X)\,).$$

En este ejemplo la fórmula resultante no posee ninguna variable libre, es por tanto una sentencia, además debe observarse que q(X) y p(X) se refieren a variables diferentes, como la variable q(X) es cuantificada primero deja de ser libre. Para evitar confusiones debe evitarse este caso, estableciéndose que:

Para cada variable X en p, o bien todas las ocurrencias de X son libres, o bien todas las ocurrencias están cuantificadas a la vez.

Cualquier fórmula que no verifique esta restricción puede ser transformada fácilmente, renombrando las variables. Así la primera fórmula quedará como:

$$(\exists X)\,(\,p(X) \wedge (\exists Y)\,q(Y)\,).$$

Para evitar interpretaciones incorrectas de las fórmulas pueden utilizarse separadores, () [] {}. Sin embargo en determinadas circunstancias, éstos pueden evitarse en base a las reglas de **asociatividad** de los operadores and y or, y en base a la siguiente jerarquía o **reglas de precedencia**. Ordenados de mayor a menor jerarquía:

$$\forall, \exists, \wedge, \vee, \rightarrow, \Leftrightarrow$$

Una fórmula se dice que está en forma normal o prenex, si todos los cuantificadores han sido desplazados al principio de la fórmula. Posteriormente se utilizará esta forma en la

conversión de fórmulas a cláusulas. Las **cláusulas** son un tipo especial de fórmula que sólo posee **operadores disyuntivos.**

3.1.5 Reglas de Inferencia.

Si la representación del conocimiento, en un contexto dado, viene representado por las fórmulas:

A1. $(\forall X)$ [tiene (alas, X) $\rightarrow$ vuela (X)].

A2. tiene (alas, ave).

Diremos que estas fórmulas, que se verifican en un contexto, constituyen **axiomas**. *Los axiomas son fórmulas que se dan por ciertas.* Si a partir de estos axiomas concluímos en la verificación de la fórmula:

T. vuela (ave).

Diremos que hemos verificado el teorema vuela (ave). El procedimiento de verificar tal teorema se denomina **demostración** y los mecanismos utilizados en la misma constituyen las **reglas de inferencia**. Las reglas de inferencia son pues mecanismos de transformación de las fbf, **axiomas,** que producen otras fbf, los **teoremas**. El proceso de demostración consistirá en aplicar las reglas de inferencia que conducen o transforman las condiciones iniciales hasta las conclusiones. Las reglas de inferencia del cálculo de predicados más importantes para nuestros propósitos son:

- Especialización Universal. Esta regla define que:

$$(\forall X)\, p(X) \Rightarrow p(a).$$

Es decir, que es siempre posible deducir la verificación para un caso concreto de un cuantificador universal.

- Sustitución. Esta regla permite sustituir cualquier proposición, en una fórmula, por otra biequivalente. Para los predicados, las proposiciones deben ser biequivalentes en todo el dominio del discurso.

- Modus Ponens. Esta regla define que de una implicación y de la *verificación de la premisa, se verifica la conclusión.*

$$[\,(p \rightarrow q) \wedge p\,] \Rightarrow q$$

Es decir, si existe un axioma de la forma **p▮ q, y otro axioma que verifica p**, entonces esta regla de inferencia establece que que se verifica el teorema **q**.

Diremos entonces que **q** es una **consecuencia lógica** de los axiomas **p▮ q** y **p**.

- Modus Tollens. Esta regla establece que:

$$[\,(p \rightarrow q) \wedge \neg q\,] \Rightarrow \neg p$$

Es decir, que si existe un axioma de la forma **p▮ q** , y otro axioma que establece **¯q**, se establece la verificación del teorema **¯p**.

Estas reglas de inferencia, **constituyen los pilares básicos sobre los que se construyen los métodos de resolución** o demostración de teoremas. Así en el ejemplo con que se inicia este apartado, se puede demostrar el teorema T

en base a aplicar estas reglas de inferencia a los axiomas **A1** y **A2**. Aplicando la regla de **especialización universal** al axioma **A1**, se obtendrá:

tiene (alas , ave) → vuela (ave).

tiene (alas , ave).

Aplicando la regla de **modus ponens** se concluirá que, vuela (ave), con lo que el teorema está demostrado. Otro procedimiento de *demostración*, denominado **resolución**, consiste en suponer que el teorema es *falso*, es decir:

tiene (alas , ave) → vuela (ave).

¯vuela (ave).

Aplicando la regla del modus tollens se podría deducir que, tiene (alas , ave), pero dado que esta conclusión entra en contradicción con el axioma **A2**, se deduce que el teorema es cierto. La demostración de teoremas lógicos es un procedimiento que involucra *una mera actividad sintáctica*, o de relación entre símbolos, *ignorándose completamente el contenido semántico que resulta de la interpretación de las fórmulas.*

3.2. Representación mediante cláusulas.

Los sistemas basados en predicados tienen un papel clave en IA por ser un método de representación de conocimiento y un mecanismo sencillo y elegante de resolución de problemas. De entre todas las fórmulas lógicas, existe un tipo denominado **cláusulas** que se presta en mayor medida a estos objetivos, debido a su **sencillez formal** y a su

modularidad conceptual. *Se considera una cláusula, a una fórmula que cumple los siguientes requisitos:*

- No posee cuantificadores existenciales.
- No posee conectores conjuntivos, es decir and.
- Pueden eliminarse, por convenio, los cuantificadores universales.

Toda fórmula que cumpla estos requisitos es una cláusula.

Por último se definirá el concepto de **Consistencia e Inconsistencia** de un conjunto de cláusulas. Los conceptos de consistencia e inconsistencia son mutuamente complementarios, por tanto un conjunto de cláusulas es inconsistente si y solo si, no es consistente. Siguiendo a **R. Kowalsky**, se considerará una **Interpretación** de un conjunto de cláusulas, como *la asignación de cada predicado a una relación, en el universo del discurso.*

Un conjunto S de cláusulas es consistente si, y solo si, todas las cláusulas son ciertas para alguna interpretación de S. (si no tiene contradicción)

Una cláusula es cierta en una interpretación de un conjunto de cláusulas S, si y solo si, es cierta para cada realización, o "instance", de las variables libres de la cláusula. Una realización se obtiene reemplazando las variables libres por términos, no variables, del universo del discurso.

Una acción de una cláusula es cierta en una interpretación I, si y solo si, o bien por un lado, todas sus condiciones son ciertas en I, o por otro lado, al menos uno de los elementos disyuntivos es cierto en I.

3.2.1 Cláusulas de Horn.

La expresión general de una cláusula es de la forma:

$$a^1 \wedge a^2 \wedge \ldots \; a^n \rightarrow b^1 \vee b^2 \vee \ldots \; b^m$$

Existiendo n antecedentes y m consecuentes. Los métodos de demostración de teoremas se reducen considerablemente, si en lugar de utilizar cláusulas generales se utiliza un caso particular, denominado cláusulas de Horn. Estas tienen la particularidad de sólo poseer un consecuente, es decir, son de la forma:

$$a^1 \wedge a^2 \wedge \ldots \; a^n \rightarrow b$$

Una representación en base a las cláusulas de Horn posee diversas ventajas e inconvenientes. Las principales ventajas son:

* La demostración de teoremas o resolución de un conjunto de cláusulas de Horn, es considerablemente más sencilla que en el caso general.

* Las cláusulas de Horn admiten una interpretación en términos de lenguaje de programación, interpretación que no poseen las cláusulas generales.

En los inconvenientes se destacarán:

* No se puede construir por métodos automáticos un conjunto de cláusulas de Horn, a partir de las especificaciones de un problema. Como se ha visto, si es posible hacerlo con cláusulas generales.

Sin embargo esta dificultad está compensada en parte por el hecho de que según **Kowalsky**:

Para cada conjunto de cláusulas generales existe un conjunto de cláusulas de Horn, tal que el conjunto original es consistente o inconsistente, si y solo si, lo es el conjunto de cláusulas de Horn.

En la práctica, este hecho significa, que si un problema se puede expresar con cláusulas generales, entonces existe un conjunto de cláusulas de Horn que expresa el mismo problema. Este conjunto debe ser construido en base a la semántica de las especificaciones, y no en base a un proceso automático.

3.2.2 Unificación y Sustitución.

El proceso de Unificación determina las condiciones y posibilidades de sustitución de un predicado por otro. Así por ejemplo en el caso trivial de los axiomas:

$$(\forall X)\ (\ p(X) \rightarrow q(X)\ )$$

$$p(a)$$

La demostración de q(a) es consecuencia de la especialización universal aplicada a: p(X) (X) , produciendo: p(a) q(a) . También puede interpretarse que se ha procedido a realizar la sustitución de p(X) por p(a) , dada la unificación de **X** por **a**. Este proceso de sustitución de términos para hacer que las expresiones sean idénticas es fundamental en la aplicación de la regla de sustitución en predicados. En general, **para realizar el proceso de unificación se utilizan métodos de comparación de literales**, en este caso de variables, constantes y functores.

Un proceso de unificación puede ser representado por un conjunto de pares de términos ordenados, es decir:

$$\theta = (\ t_1/v_1\ ,\ t_2/v_2\ ,\ \ldots\ ,\ t_n/v_n\ )$$

En donde el par t_1/v_1 indica que el término v_1 es sustituido por el t_1; siendo v_1 una variable. Para facilitar la interpretación de la unificación, se utilizará esta otra nomenclatura:

$$\theta = (v_1 \rightarrow t_1, v_2 \rightarrow t_2, \ldots, v_n \rightarrow t_n)$$

Así, dado el predicado **p** y varias de sus realizaciones:

P: p(X , f(Y) , b)

P1: p(g(Z) , f(a) , b)

P2: p(X , f(a) , b)

P3: p(Z , f(U) , b)

P4: p(c , f(a) , b)

Se pueden construir las unificaciones t_{aj} que:

$\theta_1 = (X \rightarrow g(Z), Y \rightarrow a)$

$\theta_2 = (Y \rightarrow a)$

$\theta_3 = (X \rightarrow Z, Y \rightarrow U)$

$\theta_4 = (X \rightarrow c, Y \rightarrow a)$

$$p(g(Z), f(a), b) = p(X, f(Y), b)\theta_1$$

El proceso de unificación implica la sustitución de una variable por otro término, que puede ser constante, variable o functor, en este último caso el functor no puede contener la variable sustituida. Según estas reglas de unificación, no siempre será posible unificar dos realizaciones de un predicado, en cuyo caso son no unificantes, como en este caso:

$$q(a, f(X), b) \qquad q(c, f(d), b)$$

Las unificaciones poseen la propiedad asociativa, pero en general, no poseen la propiedad conmutativa.

3.3. Resolución por Refutación.

El método de resolución, es un procedimiento de demostración de teoremas, desarrollado en el año 1965 por el investigador **J.A. Robinson**. Este procedimiento utiliza el **principio de refutación,** por lo que también es conocido como **resolución por refutación.**

Intuitivamente se puede expresar este principio para el caso de demostración de teoremas: supóngase que se quiere demostrar un teorema dado **en base a un conjunto de axiomas,** supuestos consistentes. Si el teorema es una **consecuencia lógica** de los axiomas, **entonces el conjunto constituido por los axiomas y el teorema es consistente,** o lo que es lo mismo el conjunto constituido por los axiomas y la negación del teorema es inconsistente. **El método de resolución se basa en demostrar la inconsistencia del conjunto formado por los axiomas y la negación del teorema.** Si se alcanza tal inconsistencia es porque el teorema es cierto.

Las variables libres utilizadas en la demostración del teorema provienen de variables existenciales, ya que:

$$\neg(\exists X)\, t(X) = (\forall X)\, \neg t(X) = \neg t(X)$$

3.4. Programación en Lógica.

Por programación en Lógica, se llama a la utilización de sistemas basados en reglas, en la realización de la computación y en la definición de programas. En tales tipos de sistemas, se utiliza el cálculo de predicados para definir las

relaciones entre las entradas, estructuras de datos y las salidas, consistiendo el proceso de computación en la demostración o satisfacción de un objetivo. Este esquema de computación permite ignorar, en parte, el proceso de resolución del problema, en base asuponer que las especificaciones lógicas del mismo son suficientes para que el demostrador automático de teoremas, encuentre las soluciones.

La utilización del cálculo de predicados en tareas de computación dentro de IA, es más importante en las siguientes tareas:

* **Computación General.** En la resolución de problemas de carácter general, no sólo de IA, en forma de lenguaje de programación. Este es el caso concreto del lenguaje **Prolog.**

* **Representación de Conocimiento.** Al utilizar la lógica como vehículo para representar relaciones entre objetos o hechos ciertos. Generalmente en estas aplicaciones no se suele emplear lógica clásica, como la expuesta, sino un tipo de sistemas de producciones con grados de certeza continuos, no discretos.(bases de datos)

* **Planificación de Acciones.** En aplicaciones de Robótica, y en general en todas las ramas de la técnica que requieran la manipulación y transformación de elementos. En estas áreas los problemas no son de tipo entrada-salida, sino más bien de transformación de una situación actual en una deseada, de tal forma que el trayecto o camino entre ellas está restringido por condiciones. Estas aplicaciones se encuentran en Robots Móviles y en tareas de ensamblado.

3.4.1 Resolución Declarativa de Problemas.

El concepto de resolución declarativa de problemas, se basa intuitivamente en considerar que si un problema se declara correctamente por el programador, su resolución, o lo que es lo mismo la secuencia que lo resuelve, puede ser llevada a cabo por un sistema automático. Si esto fuera posible el trabajo del programador se simplificaría. En la

resolución de un problema se deben considerar estos tres aspectos:

- Determinar las **estructuras de datos** que son adecuadas para representar los datos del problema.
- Determinar las **relaciones** existentes en esas estructuras de datos, que conducen a los resultados deseados.
- Organizar una estructura de **control** que pueda transformar los datos en los resultados, por medio de unas secuencias de cómputo.

Clásicamente se considera que los dos últimos apartados definen el **algoritmo** de resolución del problema. Así, la resolución de un problema se compone de dos fases: determinación de las estructuras de datos y determinación del algoritmo de resolución. Así, parafraseando a **N. Wirth**, un programa se compone de estos dos aspectos:

$$\text{Programa} = \text{Estructura de Datos} + \text{Algoritmo}$$

Este esquema en la resolución de problemas es plenamente válido, si no se dispone de herramientas de computación que separen los dos últimos apartados.

La utilización del cálculo de predicados como elemento de especificación de problemas, permite en parte separar el concepto de relaciones entre los datos,del concepto de secuencia de resolución. Si un problema se expresa en forma completa mediante unos hechos, unas reglas de relación, y la consecución del resultado se formula como un objetivo, entonces, en principio, la obtención del objetivo vendrá determinada por un método automático general, tal como el procedimiento de deducción o el más general de resolución. Este enfoque de la programación es lo que se denomina

Programación en Lógica, es decir, programación que utiliza la lógica como vehículo de especificación y resolución. Esta concepción es inicialmente concebida por **C. Green**, posteriormente por **P. J. Hayes**, y ha sido popularizada por el investigador **R. Kowalski**, el cual expresa este concepto como:

$$\text{Algoritmo} = \text{Lógica} + \text{Control}$$

Para que este concepto de la programación en lógica se convierta en realidad, es preciso disponer de un sistema automático de demostración de teoremas. Así, de las tres partes de la programación en lógica: Estructuras de Datos, Lógica y Control, son responsabilidad del programador las dos primeras, mientras que la tercera se supone realizada automáticamente por el sistema. Actualmente el sistema más eficiente de demostración de teoremas lo constituye el lenguaje de programación en lógica **Prolog**.

En la programación en lógica que pudiéramos denominar "pura", el programador no tiene que incluir ningún tipo de procedimentalismo, es decir, de especificación del control. En la práctica es preciso incluir algunos elementos con un carácter extra-lógico, o sea, no fundamentados en la lógica. Entre los más comunes cabría destacar:

- Predicados de efecto lateral, es decir, aquellos cuyo cumplimiento se ve acompañado de un efecto procedimental ajeno a la lógica. Entre éstos destacaremos los predicados de entrada-salida y los de tipo aritmético.

- Acciones sobre la secuencia de control. De forma que el programador pueda alterar en determinadas circunstancias la secuencia del control.

Otro punto oscuro de la programación en lógica, estriba en como representar las estructuras de datos en un formalismo basado en cálculo de predicados. La mayor parte de las soluciones dadas a este problema, consisten en considerar los términos de los predicados en forma de estructuras de datos no simples, incluyendo de esta forma

elementos tales como árboles o listas. Sin embargo, esta solución no es del todo satisfactoria en el momento actual, particularmente para estructuras indexadas como vectores y matrices.

3.4.2 Ventajas e Inconvenientes de la Lógica.

Aparentemente la utilización del lenguaje de cálculo de predicados, en la resolución de problemas y representación en IA, posee sólo ventajas. Evidentemente, posee muchas ventajas, aunque también algunos inconvenientes. Los unos o los son más destacados dependiendo de la "ideología" con que se adopte el estudio comparativo de los sistemas de producción. A pesar de todo, existen ventajas e inconvenientes que no pueden ser discutidos por su evidencia. Entre las ventajas destacaremos:

* **La lógica constituye una forma muy natural de expresar ciertos conceptos o conocimientos,** siendo un lenguaje muy próximo a una concepción del pensamiento humano.

* **La lógica es precisa,** es decir, constituye un lenguaje claramente definido, a modo de un lenguaje de programación, aportando además su sencillez.

* **La lógica es flexible,** es decir, la declaración de un hecho o regla es independiente de su posible utilización en diversas circunstancias.

* **La lógica es modular,** es decir, un hecho o regla puede ser introducido en un bloque de conocimiento, independientemente de otros hechos o reglas, esto permite estructurar los sistemas en forma modular, de forma que la adición de un conocimiento nuevo no necesita modificar el previo.

* La lógica permite separar los aspectos de la representación de los del proceso, lo que puede concentrar los esfuerzos humanos en las tareas de más alto nivel.

Frente a todas estas ventajas, existen también inconvenientes que deben tenerse en cuenta:

* **La lógica es costosa**, es decir, la resolución de un problema en forma declarativa mediante el cálculo de predicados, exige un coste superior al mismo en forma procedimental. Además la demostración de teoremas conduce a exploración de árboles, lo que implica que el coste puede sufrir una explosión, por poco que el número de reglas y niveles implicados aumente.

* **La lógica no es completa**, es decir, no todos los aspectos del pensamiento humano o métodos de resolución de problemas, pueden ser expresados a través de la lógica. Existen problemas o conocimiento, de marcado carácter procedimental, en los que no es posible separar el conocimiento del proceso.

* **La lógica no es muy adecuada para incorporar la heurística.** A pesar de los sistemas de producción borrosos, la incorporación de modelos específicos está restringida por la estructura formal de la lógica.

* La lógica es monótona, es decir, un teorema puede ser cierto con respecto a un conjunto de axiomas, pero puede no ser cierto si al conjunto original se le añaden nuevos axiomas. Este inconveniente puede ser debido a un cambio de interpretación del conjunto de axiomas o una falta de refinamiento en los axiomas.

El clásico ejemplo de la cualidad monótona de la lógica es el siguiente:

vuela (X) ← ave (X)

ave (colibrí) ←

ave (canario) ←

Si a este conjunto de predicados aparentemente ciertos, se le añade el de vuela (pingüino), deberíamos concluir que ave (pingüino). Este problema está relacionado con la excesiva generalidad de la regla del ejemplo.

4. Lenguaje Prolog 1ª parte.

4.1. El Fundamento Lógico del Prolog.

4.1.1 Introducción

En este capítulo se intentará explicar el fundamento lógico y el por qué de las limitaciones inherentes del **Prolog** actual.

Los lenguajes de programación convencionales pueden medir su potencia de cálculo o su mejor o peor disposición para cierto tipo de aplicaciones de diversas formas. Quizá la causa de la inevitable pregunta ¿ **Lisp** o **Prolog** ? cuando se habla de programación lógica, sea el tratar a estos lenguajes con los mismos criterios que los convencionales. Ambos lenguajes son muy potentes en aplicaciones con la programación lógica, pero tienen limitaciones en base a su construcción, el primero como evaluador de funciones y el segundo como demostrador de teoremas. La pregunta puede

replantearse de la forma ¿ cómo representar mi conocimiento del universo y ¿cómo resolver mis cuestiones cómo **Funciones** o cómo **Teoremas**?.

El objetivo inicial del **Prolog** en 1975 (**A. Colmerauer**) fue crear un lenguaje de programación basado en el **Principio de Resolución** (desarrollado por **A. Robinson**) de *demostración automática de teoremas*. Dicho principio con la ayuda de deducciones elementales permite construir de forma sistemática una demostración, con lo que el sistema **Prolog** sería capaz de seguir un razonamiento según la *lógica de primer orden*.

4.1.2 La Programación Lógica.

La programación lógica puede decirse que empezó a principio de los años 70 como una solución directa de los primitivos demostradores automáticos de teoremas. A partir del trabajo de **Herbrand** en 1930, en 1965, **Alan Robinson** publicó el llamado **Principio de Resolución de Robinson,** (A Machine-oriented Logic Based on the Resolution Principle) que es particularmente adecuado para el tratamiento mediante computador.

En 1972 **R. Kowalski** y **A. Colmerauer** sentarón la idea fundamental de que la lógica podría emplearse como un lenguaje de programación. De ellos es el nombre **Prolog** como un acrónimo de **PRO**gramming in **LOG**ic y el primer intérprete de **Prolog** se escribió en lenguaje **Algol** en la Universidad **d'Aix-Marsella** por **P. Roussel.** Esta idea de que la lógica de primer orden pudiese emplearse como lenguaje de programación fue revolucionaria, ya que hasta ese momento

sólo había sido empleada en la ciencia de la computación como un auxiliar de las descripciones y especificaciones de los programas.

Una de las ideas principales de la programación lógica debida a **Kowalski** es que un algoritmo consta de dos componentes que no tiene nada en común: la lógica y el control.

–La lógica es la declaración de *cual es el problema que debe resolverse.*

–El control es la declaración de *cómo debe resolverse.*

El ideal de la programación lógica sería que el programador sólo tuviese que especificar los componentes lógicos del programa y que el sistema se encargase por sí sólo del control. Todavía no se ha llegado a este ideal, porque hay dos problemas fundamentales sin resolver.

El primero es el control. Normalmente los programadores necesitan aportar bastante información de control ya sea para orenar el tratamiento de los datos o para otras aplicaciones propias. Pero en ambos casos las características actuales del control son insuficientes, por lo que una de las tareas de la investigación consiste en ampliar estas características. Otra fuente de investigación es la de transferir la responsabilidad del control del programador al propio sistema.

El segundo problema es el de la negación. El **Prolog** está basado en la lógica de las cláusulas de Horn que no tienen suficiente fuerza expresiva para soportar la negación lógica,

por lo que los sistemas **Prolog** actuales sólo admiten los literales (partes elementales) negativos en el cuerpo de las cláusulas (conjuntos de literales). La línea de la investigación actual va en el doble sentido de encontrar una solución para simular o aproximarse a la negación según el concepto de la lógica y liberar el **Prolog** de las Cláusulas de Horn.

Kowalski se fundaba en la idea de que la lógica tiene su propio carácter procedimental para poder considerarse como un lenguaje de programación. En pocas palabras, si una cláusula de un programa:

$$A \leftarrow B1, \ldots, B_n$$

que se leería como: si todos los B_1, B_2, . . . ,B_n son verdaderos entonces **A** es verdadero, donde **A** recibe el nombre de **cabeza** de la cláusula y $B_1 \ldots B_n$ recibe el nombre de **cuerpo**, que se considera como la definición de un procedimiento, y si:

$$\leftarrow C_1, \ldots, C_k$$

es una cláusula de objetivo (u objetivo simplemente), entonces cada C_j se considera como una llamada de procedimiento. Un programa funciona dándole un objetivo inicial. Si el objetivo en curso es:

$$\leftarrow C_1, \ldots, C_k$$

un paso del cómputo significa "unificar" un C_j con la cabeza **A** de una cláusula de programa:

$$A \leftarrow B1, \ldots, B_n$$

con lo que reduce el objetivo en curso al objetivo:

$$\leftarrow (C_1, \ldots, C_{j-1}, B_1, \ldots, C_{j+1}, \ldots, C_n)\,\&$$

donde **&** representa la sustitución por medio de la unificación. El cómputo acabará cuando se produzca el objetivo vacío (no hay más objetivos).

4.1.3 El Principio de Resolución.

Entre los muchos trabajos sobre la demostración automática de teoremas, hasta la fecha el más adecuado para ser tratado automáticamente por ordenadores es el **Principio de Resolución** (o simplemente **Resolución**) de **Alan Robinson**. Dadas dos cláusulas relacionadas la Resolución generará una nueva cláusula que es consecuencia de las otras dos. A las dos primeras cláusulas se les denomina **axiomas** o **hipótesis** y a las que van a obtenerse de ellas **teoremas**.

La Resolución está ideada para trabajar en forma clausal (con cláusulas) debido a lo cual si las hipótesis originales están expresadas en otra clase de fórmulas hay que transformarlas a la forma clausal.

La idea fundamental de la Resolución es que dadas dos cláusulas (hipótesis), si un mismo literal (elemento o parte atómica) aparece en el lado izquierdo (la cabeza) de una cláusula y en el derecho (el cuerpo) de la otra se obtiene la nueva cláusula (teorema) encajando ambas y eliminando los elementos que resulten repetidos.

Como es de suponer, la Resolución no es tan sencilla como se explica en estas líneas ya que existen dos problemas fundamentales:

* El posible valor de las variables contenidas en las cláusulas ya que estos valores varían según los pasos previos.

* El concepto de relación que existe entre las dos cláusulas ya que no es necesario que sean idénticas sino que "puedan coincidir" siendo este grado de coincidencia lo problemático.

Hay que añadir otro problema a nivel de ejecución de los programas que consiste en saber elegir (estrategia de elección) las cláusulas que son más idoneas para que al aplicarles la Resolución se obtengan los teoremas deseados, de lo contrario se malgastará mucho tiempo de cálculo en intentar relacionar todas las posibles parejas de hipótesis sin resultados apetecidos.

Una importante propiedad de la Resolución es que ante un conjunto de cláusulas inconsistentes sólo derivaría la cláusula vacía entendiendo por cláusulas inconsistentes aquellas que no representan simultáneamente expresiones verdaderas y por cláusula vacía la expresión lógica de falsedad (es decir una proposición que no es verdadera). Otra propiedad no menos importante es la de integridad por la cual un teorema deducido de unas hipótesis, con la Resolución se demostraría la inconsistencia entre la negación del teorema deducido y las hipótesis generales, dando como resultado la cláusula vacía.

Se han propuesto diversos sistemas para refinar el Principio de Resolución y otros muchos están en fase de investigación.

4.1.4 Las Cláusulas de Horn.

Uno de los muchos sistemas para refinar el Principio de Resolución es restringir las cláusulas a una determinada clase: **las cláusulas de Horn**, que definimos como una cláusula que tenga a lo más un literal no negativo. De la definición se desprende que hay dos tipos de cláusulas de Horn:

* Las que tienen un literal no negativo (o afirmativo) y que llamaremos "encabezadas".

Los literales no negativos se escriben a la izquierda del símbolo ← y consideraremos que son cláusulas condicionales, un ejemplo sería:

$$A$$

$$A \leftarrow B_1, \ldots, B_n$$

y en lenguaje más natural podría ser:

esposa (X) ← cónyuge (X) and mujer (X).

* Las que no tienen ninguno y que llamaremos "descabezadas". Siempre serán verdaderas independientemente del valor que se asigne las variables, por lo que las llamaremos cláusulas incondicionales, por ejemplo:

$$\leftarrow B_1$$

o en lenguaje natural:

← esposa (X).

Las cláusulas de Horn consideran solamente aquellos conjuntos de cláusulas donde todas menos una están encabezadas (es decir que tengan un literal no negativo) porque emplea como método de selección de objetivo la cláusula descabezada y el resto de las cláusulas como hipótesis. Por otra parte al resolver dos cláusulas de Horn encabezadas resulta otra cláusula encabezada.

4.1.5 El Prolog.

El Prolog se basa en las cláusulas de Horn y como vimos en la sección anterior una cláusula de Horn es una cláusula que a lo mucho tiene un literal no negativo y que se representaba como:

$$A \leftarrow B_1, \ldots, B_n$$

y

$$\leftarrow B_1, \ldots, B_n$$

pero esta notación es un poco incómoda y reñida con el lenguaje natural que se propugna para el **Prolog** por lo que se emplea el convenio siguiente:

El símbolo ← lo sustituiremos por la palabra **if**, la relación entre los diversos literales o elementos que se convertirán en objetivos se sustituirá por **and**. A las cláusulas en general las llamaremos **predicados** y cuando son cláusulas condicionales reciben el nombre de **reglas**.

$$A_i \leftarrow B_{1i}, \ldots, B_{ni}$$

por ejemplo:

esposa (cónyuge) if conyuge (X) and mujer (X)

y cuando son cláusulas incondicionales se denominan **hechos**:

$$A$$

por ejemplo:

mujer (dolores)

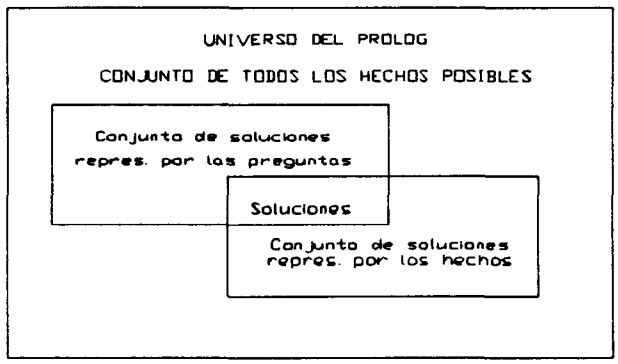
el conjunto de cláusulas se denomina **base de datos** y al conjunto de hechos **base de hechos**.

Los programas en **Prolog** serán una visión particular del Universo que se traducirán en hechos o en reglas (hipótesis) y el objetivo del **Prolog** es sacar conclusiones de ellas, es decir son cuestiones o teoremas que debe demostrar.

Acudiendo a la figura siguiente tenemos que nuestro universo del discurso por medio de la base de datos está representado en el universo del **Prolog** (cuadro exterior). Es el conjunto de todas las situaciones posibles. Al **Prolog** se le propone una cierta pregunta (cuadro de las preguntas) que

en sí misma tendría muchas soluciones, pero solamente tenemos en cuenta las que están dentro del universo del **Prolog** y de éstas solamente las que correspondan a los hechos establecidos (cuadro de hechos), siendo las soluciones finales el cuadro resultante de la intersección de los dos anteriores.

Universo Prolog



Para resolver unas cláusulas de Horn es necesario que alguna de ellas sea descabezada, lo que significa en Prolog que entre los predicados a resolver haya al menos un hecho (cláusula incondicional o descabezada). El proceso de Resolución sería aproximadamente:

- Primero seleccionar un objetivo Cj normalmente el primero de la izquierda en la cláusula programa.

- Se intenta encontrar, en cada etapa del proceso, el objetivo C_j en cada una de las hipótesis indicadas más abajo, hasta que aparezca por primera vez el objetivo C_j, con lo que se unificará A_j en C_j y se incluye en la lista de objetivos $\leftarrow B_{ij},\ldots,B_{nj}$, convirtiéndose en subobjetivos.

$$A_j \leftarrow B_{ij}, \ldots, B_{nj}$$

- Cuando un objetivo no se puede unificar con ninguna de las hipótesis, el Prolog no entrega ninguna solución y retrocede al paso anterior.

- El proceso finaliza cuando no queda ningún objetivo por seleccionar, con lo que el Prolog ofrece la lista de soluciones encontradas.

4.1.6 La Máquina Prolog.

El proceso anterior sería la aplicación de la Resolución, pero se ve claramente que como lenguaje de programación es insuficiente porque ¿que sucederá cuando haya muchos objetivos? ¿y cuando, además, estén compuestos de varios subobjetivos? ¿y como llevar un control de las reglas probadas?. Todas las respuestas las ofrece lo que se conoce como **máquina Prolog** o también *sistema Prolog*. Seguidamente se describirá la máquina **Prolog** recurriendo al siguiente esquema:

Etapa 1. Principio

Es el momento de plantear una cuestión al Prolog, y puede estar en cualquier lugar de la ejecución de un programa, tanto puede ser la primera cuestión como la última (para distinguir de los lenguajes procedimentales donde el "inicio" suele ir al principio con el objeto de borrar todos los contadores, etc).

Etapa 2. Objetivo Inicial

Se plantea el objetivo inicial que puede estar compuesto de varios subobjetivos que serán tratados como si fuesen objetivos que a su vez pueden estar compuestos de otros subobjetivos que durante el proceso de Resolución serán sustituidos por otros subobjetivos y así sucesivamente.

Etapa 3. Control de los subobjetivos.

Como se ha dicho en la Etapa 2, cada subobjetivo se tratará como un objetivo, lo cual significa que si antes era complejo el control de los objetivos, ahora este control se dispara. Afortunadamente el Prolog controla todos y cada uno de los objetivos y subobjetivos de una forma completamente transparente para el programador.

Etapa 4. Selección de un subobjetivo.

Se efectúa la selección de un subobjetivo para intentar la Resolución. Si no hubiese más subobjetivos es como si estuviera el subobjetivo vacío.

Etapa 5. Pregunta ¿queda algún subobjetivo?

Es una forma de salir del bucle y en realidad preguntaría: ¿está el objetivo vacío?. En el caso que la rspuesta sea NO, quiere decir que la lista de subobjetivos se ha acabado, y el subobjetivo en curso no tiene más subobjetivos

que demostrar, con lo cual se obtiene la visualización de todas las soluciones. Por el contrario, si la respuesta es SI, se procederá a demostrar el subobjetivo seleccionado en la Etapa 4.

Etapa 6. Control de las reglas.

Cada objetivo (y subobjetivo) hay que probarlo con las diferentes reglas del universo del Prolog, puede que sean pocas, puede que sean muchas. Si a la complejidad del control de los objetivos le añadimos la del control de las reglas se genera un problema tan complejo que sólo la máquina Prolog (o cualquier otro lenguaje al efecto) puede hacer, este trabajo de control es transparente para el programador.

Etapa 7. Preg. ¿queda alguna regla por probar?

Es la fórmula para salir del bucle. Al igual que en la pregunta de la Etapa 5, preguntaría si la lista de reglas es la lista vacía. Si queda alguna regla por probar la respuesta es SI y se salta a la Etapa 8 para comprobar si la regla es aplicable. En el caso de que no quedasen más reglas por probar, significaría que al subobjetivo en curso no le quedan más reglas y se saltaría a la Etapa 4 para seleccionar un nuevo subobjetivo.

Etapa 8. Pregunta ¿es aplicable esta regla?

En esta Etapa se produciría la Resolución, ya que el resto de las Etapas son sólo un mecanismo de control de los objetivos y de las reglas. La pregunta es lo mismo que ¿se ha

podido aplicar la Resolución a esta regla?. En caso afirmativo el Prolog anotará las soluciones y procederá a seleccionar un nuevo subobjetivo, es decir volverá a la Etapa 4. En caso negativo, volverá a la Etapa 6 a seleccionar una regla para seguir probando.

Etapa 9. Visualiza la solución.

Al salir del bucle de Resolución de un objetivo se visualizan las soluciones.

Etapa 10. Pregunta ¿quedan más objetivos?.

Es la misma pregunta que en la Etapa 5 pero referida a objetivos en una misma cláusula de programa. En el caso de que aún quedasen más objetivos, el sistema vuelve al bucle descrito entrando en la Etapa 2. Si no quedan más objetivos se dirige a la Etapa 11.

Etapa 11. Fin.

Se acaba la cuestión planteada al Prolog con una serie de soluciones. El sistema queda dispuesto para una nueva cuestión.

5. Lenguaje Prolog 2ª parte.

5.1. LOS PREDICADOS

Los elementos fundamentales del Prolog son los predicados, que es aquello que se afirma o se dice acerca de un sujeto.

Una primera clasificación de los predicados puede ser: **creados** o **instalados**.

5.1.1 Predicados Creados

Los predicados creados son aquellos que nombra y define el programador, y pueden ser:

- **-Predicados autodefinidos o unitarios**: son aquellos que contienen la información del sujeto sin depender de ninguna condición, son por lo tanto incondicionalmente ciertos. Si esta información es inmutable o estable se denominan hechos y si puede variar se le denomina dato.

Por ejemplo:

abuelo(damián,josé).

- **-Predicados definidos por reglas:** son aquellos cuya definición se basa en la de otros, y que viene expresada en forma de regla,por depender de otros predicados estan condicionados a la certeza de esos predicados, son por tanto ciertos de forma condicional.

Por ejemplo:

abuelo(x,z) if padre(x,y) and padre(y,z).

A los predicados definidos por reglas que sólo dependen de otro predicado se les denomina iterativos.

Por ejemplo:

antepasados(x,z) if abuelo(x,z).

5.1.2 Predicados Instalados

Los predicados instalados, incorporados, incluidos o predefinidos: son aquellos que están definidos en el intérprete o compilador de Prolog que utilizamos.

Dentro de los predicados instalados se pueden distinguir dos tipos:

- los condicionales que son ciertos cuando los elementos del programa cumplen una serie de especificaciones.

Por ejemplo:

free(x).

es cierto cuando la variable X es libre.

- las acciones que son ciertas cuando se realizan las acciones especificadas por el mismo.

Por ejemplo:

```
write("hola")
```

es cierto cuando Prolog escribe la palabra hola.

5.1.3 Predicados Principales y Secundarios

Los predicados también pueden clasificarse:

- **- principales** que son aquellos que no pueden definirse mediante reglas, y engloban una parte de los predicados instalados.

Por ejemplo:

```
write("hola")
```

- **- secundarios** que son aquellos que se pueden definir mediante reglas y son todos los no instalados y parte de los instalados.

Por ejemplo:

```
escribe_en_nueva línea("hola") if nl and write("hola").
```

Hay que tener en cuenta que esta clasificación es relativa, pues como ya veremos más adelante si consideramos el predicado fail principal entonces true es secundario, pero si es true el que consideramos principal es fail el que es en este caso secundario.

Existen además las uniones o conjunciones de predicados y los modificadores de predicados, que permiten la construcción de predicados más complejos y que estudiaremos con detenimiento más adelante.

5.2. Los Hechos y La Base de Hechos

Hay que comenzar, creando en el programa, un conjunto de «verdades fundamentales» que llamaremos hechos y que van a formar el universo del programa.

Al conjunto de todos los hechos que aparecen en un programa lo llamaremos base de hechos, que contrariamente a lo que a primera vista se podría pensar no es un universo cerrado.

En base a estos hechos Prolog va a inferir mediante el proceso de unificación si un objetivo es verdadero o falso.

Un hecho es una expresión como la siguiente:

```
juan es padre.
```

Que en un Prolog se escribiría como:

```
padre(juan).
```

aunque también podría escribirse correctamente como:

```
juan(padre).
```

La primera forma es preferible a la segunda, ya que permite una mayor flexibilidad del hecho.

En nuestro ejemplo parece más lógico pensar que pueden haber otras personas que puedan ser padres pero es más difícil que juan sea otra cosa que padre.

A los hechos también se les llama predicados autodefinidos,como ya hemos visto, están compuestos por dos partes: la primera que es el **functor** o cabeza, que precede al paréntesis y la segunda parte llamada **argumento** que está encerrada entre paréntesis () y lo forman constantes, variables etc. O como ya veremos más adelantes expresiones u otros hechos .

El functor puede tener varias funciones dependiendo del número de argumentos, y del uso que haga el programador, puede pues actuar como:*sujeto, acción verbal* y como *nombre de una clase*.

Unos ejemplos nos aclararán las distintas funciones del functor:

El functor como sujeto:

-la altura es de 22 metros.

En Prolog:

```
altura_en_metros(22).
```

El functor como acción verbal:

Juan es hermano Jose.

En Prolog:

```
son_hermanos(Juan,José).
```

El functor como nombre de una clase:

Maria, Alicia y Laura son nombres de mujer.

En Prolog:

```
nombres_de_mujer(maria,alicia,laura).
```

Antes de seguir haremos dos observaciones, la primera de ellas es que **las constantes comienzan siempre con minusculas**, para diferenciarlas de **las variables que comienzan siempre por mayúsculas** y la segunda es que **al final de un hecho siempre se pone un punto.**

Vamos a ver otro ejemplo:

juan es el padre de damián.

que en Prolog sería:

padre (juan,damián).

El functor es padre y los argumentos son juan y damián.

Como vemos un hecho puede estar compuesto por más de un argumento, sin más que separarlos por comas ,.

La estructura general de un hecho es la siguiente:

functor(argumento 1,argumento 2,....,argumento N).

Aunque Prolog permite que los hechos se sitúen en cualquier punto del programa y en cualquier orden, es conveniente agruparlos y ordenarlos al comienzo del programa.El orden de los mismos es importante no sólo por razones prácticas, de ahorro de recursos y estéticas, sino también por el proceso de unificación que realiza Prolog, ya que se lleva a cabo de forma secuencial desde el principio hasta el final del programa, pudiendo depender la solución del orden realizado.

Para los datos existe otra estructura más adecuada que se llama base de datos y que veremos más adelante.

Vamos a formar ahora una auténtica «mini» base de hechos que llamaremos: «deportistas famosos».

```
jugador(butragueño,fútbol).

jugador(maradona,fútbol).

jugador(severiano__ballesteros,golf).

jugador(laurent__fignon,ciclismo).

nación(butragueño,españa).
```

```
nación(maradona,argentina).

nación(ballesteros,españa).

nación(laurent_fignon,francia).
```

podiamos haber escrito:

```
nación(maradona,japón).
```

y pese a que para nosotros hubiese sido falso,para Prolog sería verdadero, y es porque nosotros y Prolog tenemos universos diferentes y en base a ellos cada uno responde si es verdadero o falso.

Existen predicados que no llevan argumentos, y que se emplean normalmente para indicar acciones a Prolog,como por ejemplo:

```
escribe.
```

Por último señalar que existe otra forma de señalar a los **hechos**, y es como reglas sin condiciones, para lo cual hay que definir previamente lo que es una regla, esta forma de verlo se apoxima más a la sintaxis de algunas versiones de Prolog que siguen el estándar Marsellés (p.e.Prolog II).

5.3. EL DIALOGO

Ahora vamos a dialogar con la base de hechos. Para ello pasamos al modo interactivo del Prolog.

El modo interactivo de Prolog, es aquel que nos permite dialogar a través de Prolog con el programa.

Vamos a hacer aquí un pequeño alto, para definir un término se empleará a lo largo de todo el texto, y es el del nombre del proceso que realiza Prolog sobre un programa para lograr los objetivos fijados.

No hablamos con propiedad si decimos que Prolog ejecuta un programa ya que ejecutar es poner en obra una cosa, hacer algo determinado, y esto es cierto en un lenguaje procedimental pero no en uno declarativo.

Tampoco es correcto llamarle unificación pues es sólo parte del proceso realizado por el Prolog, el lenguaje también hace la marcha atrás,etc.

Más correcto es el uso de verificar, probar algo, o el de comprobar, que significa confirmar una cosa por medio de cotejo o demostración, o el de demostrar, que significa probar una cosa (objetivo) partiendo de verdades universales y evidentes (los hechos).

De aquí en adelante denominaremos al proceso global que realiza Prolog; demostrar, o demostrar un objetivo.

Prolog nos informa que podemos marcarle un objetivo para demostrar con la forma:

```
Goal:
```

Vamos primeramente a demostrar un hecho.

Por ejemplo:

```
¿maradona es jugador de futbol?
```

En Prolog sería:

```
Goal: jugador(maradona,futbol).
```

A lo que Prolog nos contestará:

```
true
```

es decir verdadero, ahora vamos a preguntarle:

¿petrovic es jugador de baloncesto?

Que en Prolog sería:

goal:jugador(petrovic,baloncesto).

La respuesta en este caso sería:

False.

Vamos a realizar un nuevo tipo de pregunta, la de consulta o pregunta existencial, queremos conocer a todos los jugadores famosos del fútbol, que en Prolog sería:

Goal:jugador(nombre ,futbol).

Prolog nos responderá:

nombre = butragueño

nombre = maradona

Es decir ha buscado todos aquellos hechos cuyo segundo término coincide con la constante del objetivo(fútbol) y ha escrito como solución los distintos valores de la variable(nombre).

Podemos desear conocer el nombre de todos los deportistas famosos sin importarnos el deporte que practicas,esto lo podemos preguntar en Prolog como:

goal:jugador(nombre,____).

A lo que Prolog responderá

nombre = butragueño

nombre = maradona

nombre = severiano_ballesteros

nombre = laurent_fignon

El guión bajo "____" se emplea en Prolog para sustituir a los espacios dentro de una constante o una variable y como *variable anónima es decir aquella que no necesitamos nombrar pues no nos referimos a ella.*

Por último podemos realizar consultas complejas y/o restrictivas como por ejemplo:

"dime los nombres de los jugadores españoles de fútbol famosos".

que en Prolog sería:

goal:jugador(nombre,fútbol) and nación(nombre,españa).

Prolog contestará:

nombre = butragueño

un último ejemplo es preguntar:

"dime el nombre de otros deportes que practican los jugadores famosos aparte del fútbol".

que en Prolog sería por medio de los símbolos < > que significa distinto:

goal:jugador (__,deporte) and deporte fútbol.

A lo que Prolog nos contestaría:

deporte = golf

deporte = ciclismo

Si ahora le marcamos el siguiente objetivo:

goal:escritor(Cervantes,novela).

Prolog nos contestará con un error pues no contiene su base de hechos ningún hecho con este functor.

Hasta el momento se ha tecleado el objetivo (goal) cada vez, sin embargo puede ir en el interior del programa (objetivo interno) o bien marcarse durante el proceso de demostración de otro objetivo, con miras a una recanalización del mismo.Esto se puede conseguir mediante el uso del predicado instalado call (objetivo).

Pero antes de dejar el libro le vamos a hacer una observación muy importante,si se ha fijado en todo este diálogo al Prolog nunca le hemos informado de como debía demostrar los objetivos y sin embargo,él siempre se las ha arreglado para conseguirlo.

Como ve poco a poco vamos comprobando el lenguaje declarativo.

5.4. LAS VARIABLES

5.4.1 DEFINICION DE LAS VARIABLES

Otra clase de elementos (o objetivos) simples de Prolog son las variables, que representan objetos en los que desconocemos el valor que tendrán en un momento dado.

En la sección anterior se usaron repetidamente con un formato similar al de las constantes exceptuando las que empezaban con letra mayúscula.

Las variables pueden ser números enteros, números reales (en algunas versiones), cadenas de símbolos, etc. y dependiendo de cada versión admiten diversos tamaños de las mismas.

En ciertas versiones de Prolog los nombres de las variables tienen una forma especial, pero esto no altera en nada la funcionalidad de las variables, si a caso su legibilidad.

Pero esto último depende del gusto particular de cada programador (aunque se aconseja en la sección dedicada a la metodología que se utilicen nombres cuyas combinaciones de caracteres sean fáciles de comprender) para el Prolog lo mismo es la variable con el nombre X que con el de La__componente__transversal o con LCT129.

Todavía más, el Prolog permite utilizar la "variable anónima", representada por el guión salvo en aquellas variables que no van a ser nombradas específicamente por el programador, pudíendose utilizar cuantas se quiera en el mismo tèrmino. Así pues, tenemos como ejemplos de variables válidas:

X

XYZ

Xyz

Var2500

Entrada

Esta__es__una__variable.

5.4.2 Propiedades de las Variables

Vamos a ver algunas de las propiedades de las variables que le pueden llevar de cabeza si constantemente no las recuerda:

- La primera de ellas es que cada vez que se unifica una expresión las variables olvidan sus valores exteriores y toman los nuevos valores, es por eso que no existe

ningún predicado extandar que sea poner a cero las variables. Este primer punto la lleva a Prolog a no reconocer los valores de variables repetidas en distintas unificaciones.

- La segunda es que dentro de una expresión la primera vez que aparece la variable se dice que esta libre o no particularizada con lo que toma un valor, tras lo cual se convierte en una variable que se llama:

* -- particularizada,
* -- definida,
* -- vinculada,
* -- ligada,

(y en algunas versiones ponen "instanciadas").Con lo que en las siguientes unificaciones, si se encuentra con una variable le asigna dicho valor y si se encuentra con una constante si sus valores son iguales es verdad (éxito) y si son distintos falla.

Un pequeño ejemplo nos lo muestra:

...if...,A = 2,...,A = 3.

nos da como resultado False,mientras que:

...if...,A = 2,...,A = 2.

nos da como resultado True.

Por otra parte formas muy comunes en otros lenguajes son siempre falsas, como:

$$A = A + 1$$

Volviendo a nuestra base de hechos "deportistas famosos".

La nueva pregunta es :

deportes que practican los jugadores famosos.

Que en Prolog será:

Goal:jugador(_,deportes)

A lo que Prolog nos contestará:

Deporte = fútbol

deporte = fútbol

deporte = golf

deporte = ciclismo

Existen dos predicados instalados que hacen uso del carácter libre o particularizado de las variables, que son(según las versiones):

var (variable)

o también

free (variable)

Este predicado instalado es cierto, cuando la variable especificada en el momento de actuar el predicado es libre.

El otro predicado instalado es:

notvar (variable)

o también

bound (variable)

Este predicado instalado es cierto, cuando la variable especificada en el momento de actuar el predicado está particularizada.

Los predicados notvar y bound pueden considerarse secundarios ya que pueden definirse como:

```
notvar (X) if not (var (X)).
```

o también

```
bound (X) if not (free (X)).
```

5.5. Las Conjunciones y los Modificadores

Las conjunciones nos permiten unir los hechos con el fin de obtener estructuras más complejas.

Las conjunciones utilizadas en Prolog son: and y or, y los modificadores: not.

5.5.1 AND

La primera conjunción que vamos a ver es ***and***, que es un operador "y lógico" es decir para que una expresión formada por varias expresiones o hechos unidos por and sea verdad tienen que serlo cada uno de ellos.

Un ejemplo ayudará a comprenderlo, si le marcamos el siguiente objetivo:

```
goal:jugador(maradona,fútbol) and nación (butragueño,españa)
```

Prolog nos contestará:

```
True
```

y es verdadero pues se cumplen ambos hechos.

Si por el contrario escribimos:

```
goal:jugador(severiano_ballesteros,golf)and

jugador(laurent_fignon,españa)
```

Prolog nos contestará

```
False
```

es decir, falso y esto es debido a que el segundo hecho no es cierto.

La tabla de verdad nos muestra todas las posibles combinaciones:

* verdad AND verdad = verdad
* verdad AND falso = falso
* falso AND verdad = falso
* falso AND falso = falso

5.5.2 OR

La siguiente conjunción que vamos a utilizar es la OR que es un operador "o lógico", que significa que para que una expresión sea verdad tiene que serlo al menos uno de los términos que lo forman.

La tabla de verdad nos muestra todas las posibles combinaciones:

* verdad OR verdad = verdad
* verdad OR falso = verdad

* falso OR verdad = verdad

* falso OR falso = falso

5.5.3 NOT

Este modificador convierte un hecho en falso y viceversa.

El modificador de predicados NOT no puede aparecer como modificador de un hecho, el ejemplo:

not (jugador (Laurent_Fignon,ciclismo)).

no sería válido.

tampoco puede modificar la cabeza de una regla, por ejemplo:

not (joven(nombre)) if edad (nombre,años)and año and Años>30.

no sería correcto.

Sin embargo puede modificar tanto los predicados que forman parte de las condiciones de una regla, como los predicados empleados en la interrogación.

Con el predicado NOT abrimos el universo del programa, ya que puede reconocer como parte del programa Prolog a ciertos hechos desconocidos por la base de hechos. Este mecanismo puede emplearse para incrementar la base de hechos.

Realmente NOT puede considerarse secundario pues se puede definir como (el simbolo ! significa corte (cut) y se verá más adelante):

not (hecho) if hecho and ! and fail or true.

5.5.4 Idéntico

Otro modificador de predicados, que es secundario, es identíco (argumento, argumento), este predicado es cierto cuando los dos argumentos son identicos, y es falso cuando no lo son y puede definirse como:

idéntico (x,y) if not (not (x = y)).

5.6. LAS REGLAS

5.6.1 DEFINICION

Una regla es una expresión que contiene las condiciones para conseguir un objetivo. En Prolog estas condiciones son nuevos objetivos que deben demostrarse y que se llaman subobjetivos.

A las reglas se les ha llamado algunas veces axiomas.

La extructura de una regla es la siguiente:

If a_1 and a_2 and.... and a_n then A .

Es decir, si las condiciones a_1, a_2, etc. se demuestran que son ciertas, entonces se demuestra A.

En Prolog se escribe :

A if a_1 and a_2 and... and a_n.

A la parte izquierda del if se llama **cabeza** conclusión y a la derecha **cuerpo** o condiciones.

Por tanto if es una partícula de unión de la cabeza y el cuerpo.

Cada regla solamente puede tener una cabeza y solamente una, por ejemplo la regla:

Los perros y los gatos tienen rabo y cuatro patas.

En Prolog la siguiente regla sería incorrecta :

Numero__de__patas(nombre,cuatro) and rabo(nombre,si) if perro(Nombre) or gato(Nombre).

Y para que fuese valida habría que desdoblarla en dos:

Numero__de__patas(Nombre,cuatro) if perro (Nombre) or gato (Nombre).

rabo(Nombre,si) if perro (Nombre) or gato (Nombre).

Vamos a ver ahora un ejemplo sencillo de una regla valida:

Una persona es joven si tiene menos de 30 años.

En Prolog se escribiría:

joven (Nombre) if edad (Nombre,años) and Años < 30.

El nuevo hecho que inferimos con esta regla es el de ser joven a partir de un hecho ya condicionado que es la edad.

Si creamos ahora una minibase de hechos para probar esta regla:

```
Edad (juan, 25).

Edad(jose,35).
```

Informamos del nuevo objetivo a Prolog:

```
Goal: joven (juan).
```

A lo que nos contestará Prolog:

```
True
```

Si en lugar del anterior objetivo le indicamos el siguiente:

```
Goal: joven (nombre).
```

Prolog nos contestará:

```
Nombre=juan
```

Y si le hubiesemos pedido:

```
Goal :joven (jose)
```

Prolog nos hubiese contestado:

```
False.
```

Rápidamente habrá pensado que no hemos descubierto nada nuevo, la forma if.... then aparece en otros muchos lenguajes. Esto es sólo en parte cierto, pues por un lado en Prolog no son preposiciones lo que engloba la regla (logica de orden 0) sino que pueden ser predicados (logica de orden 1) por otro lado en Prolog se pueden escribir todas las reglas que se deseen formando una base de conocimientos y marcando un objetivo y es el Prolog quién se encarga de demostrarlo con respecto a la base de hechos.

Prolog se encargará de encadenar las reglas, evaluarlas,

---

etc. para conseguir demostar el objetivo dado, es decir contiene además un sistema de control de las reglas.

A los hechos y a las reglas se les llama sin distinción cláusulas.

Una estructura más general de las reglas en Prolog es :

A if a_1 and a_2 and...and a_n.

A if b_1 and b_2 and...and b_n.

A if m_1 and m_2 and...and m_n.

que usando la conjunción *or* queda:

A if a_1 and a_2 and...and a_n.

or

b_1 and b_2 and...and b_n

or

or

m_1 and m_2 and...and m_n.

Veámoslo con un ejemplo: Una persona puede seguir un encuentro de fútbol:

* o por la tv. y la radio simultáneamente,
* o por la radio,
* o por laT.V.
* o en directo en el propio campo.

En Prolog sería:

```
seguir_encuentro(X) if oye_radio(X) and ve_TV(X)

or ve_t.v.(X)

or oye_radio(X)

or en_el_estadio(x).
```

Observese que en este ejemplo el orden es muy importante, si la primera condición (oye__radio and ve__tv) la ponemos en último lugar nunca se activará pues antes se activaría la segunda(ve__tv).

6. Lenguaje Prolog 3ª parte.

6.1.1 Observaciones sobre las reglas.

Un programa escrito en Prolog es un conjunto de datos, hechos y reglas, con el que se puede dialogar.

En la escritura de una base de conocimiento debe tenerse en cuenta:

- 1:- Las reglas deben estar ordenadas, pues Prolog las va unificando de forma secuencial desde la primera hasta la última hasta que consiga demostrar el objetivo, o la totalidad de las soluciones si se le fuerza la marcha atrás.

- 2:- Las reglas con el mismo objetivo tienen que figurar juntas en el programa.

- 3:- Las reglas una vez ordenadas por los criterios anteriores deben ordenarse de mayor a menor, con el fin de conseguir una unificación más rápida.

- 4:- Dentro de una regla las expresiones más restrictivas deben de figurar lo más a la izquierda posible, pues esto facilita una evaluación más rápida ya que la unificación en una regla va de izquierda a derecha.

- 5:- Si entre las condiciones de una regla existen algunas acciones éstas deben de figurar lo más a la derecha posible con el fin de que no se realicen si no se verifican las condiciones de la regla.

- 6:- En la cabeza de una regla no pueden aparecer ni modificadores (not) ni uniones de predicados (and,or).

6.1.2 El Encadenamiento Hacia Atras

El encadenamiento natural de las reglas en Prolog es hacia atrás, es decir que partiendo de un objetivo se desarrolla un árbol con el fin de buscar los hechos que demuestren su veracidad o falsedad.

La estructura de las reglas es por tanto:

< objetivo> **si** <condiciones> .

Cada condición se puede convertir en un subobjetivo cuya demostración es necesaria para conseguir la del objetivo obteniéndose un encadenamiento como el siguiente:

objetivo **si** condiciones(subojetivo1
si condiciones(...))>.

6.1.3 El Encadenamiento Hacia Adelante

Sin embargo gracias a las acciones que no son otra cosa que un conjunto de predicados instalados que para que sean verdad tienen que previamente realizar una acción como

puede ser: leer un dato en el teclado o escribirlo en la patalla, etc. es posible conseguir encadenamiento hacia adelante. Siguiendo por la estructura siguiente:

< objetivo > **si** < acción > .

Bien visto el objetivo se convierte realmente en la condición necesaria para la consecución de la acción.

< condición > **para** < acción > .

6.2. Los Caracteres

Los elementos u objetos se componen de uno o más caracteres. El Prolog trata a los caracteres como un tipo más de información, concretamente como constantes enteras cuyo valor esta determinado por el código ASCII (American Standard Code for Informatión Interchange).

El Prolog reconoce dos tipos de caracteres:

* - lo imprimibles y

* - los no imprimibles.

Los primeros son los que producen una marca o señal en la impresora o en la pantalla y que se representan como:

letras minúsculas:

abcdefghijklmnñopqrstuvxyz.

letras mayúsculas:

ABCDEFGHIJKLMNOPQRSTUVXYZ.

Dígitos:

0123456789

caracteres especiales:

+-Ç*¿?:;!"$%&/()=¡+ç`.,

Estos últimos dependen de la versión particular de Prolog, algunas versiones admiten los caracteres típicos de cada idioma.

Los caracteres no imprimible no causan ninguna señal en la pantalla ni en la impresora pero realizan otras acciones tales como "imprimir" un carácter en blanco o saltar a una nueva linea. Tienen en código ASCII un valor de 32 o menor.

6.3. LAS CONSTANTES

Son elementos u objetos Prolog cuyo valor está especificado y es inalterable durante el proceso del programa. Según sus componentes y el destino a que se les dedica se les denomina *átomos* y *enteros.*

6.3.1 Los Atomos

Los átomos son las constantes simples utilizadas para representar hechos invariables que no van a utilizarse en cálculos numéricos.

Los nombres de los átomos están compuestos de la combinación de varios caracteres dependiendo de la particular versión de Prolog en cuanto a si pueden admitir mayúsculas, dígitos y cuantos caracteres como máximo. El Prolog acepta tambien como átomos ciertos caracteres especiales que tienen cierto caracter especial). En general se acepta que los átomos:

- 1:- tendrán cualquier número de caracteres,
- 2:- comenzarán con una letra minúscula o
- 3:- encerrados entre comillas:

a) cuando sea necesario que empiezen por una letra mayúscula,

b) cuando empiezen por un dígito,

c) cuando esten compuestos unicamente de dígitos y su utilización no esté destinada al cálculo numérico.

- 4:- y pueden contener el signo del guión bajo para mejorar su lectura.

Según esto los siguientes átomos son válidos:

+

gato

gATO

"GATO"

"3gatos"

"3333"

*25

trescientos__treinta__y__tres.

por el contrario los siguientes no son nombres de átomos válidos:

.geranio, 12peolot__asd

6.3.2 Los Enteros

La otra clase de las constantes, los enteros se utilizan para representar números en las expresiones de cálculo numerico. Se denominan enteros por recuerdos de las primeras versiones de Prolog, en las que solo podía haber constantes entera y positivas. En la actualidad la mayoría de las versiones, tanto para ordenadores personales como anfitriones, admiten no solamente enteros negativos sino ademas reales y algunas con notación de coma flotante. Para los ejemplos solamente utilizaremos los numeros reales. Se componen exclusivamente de dígitos y caracteres especiales con significación matemática, tales como los signos + y - y el punto decimal. Su rango depende de la capacidad del Prolog en cuestion, asi que tenemos:

"enteros positivos" comprendidos entre el

0 y el 16.383 el Prolog TM 86,

0 y el 32.767 el turbo Prolog

0 y el 16.767.215 el VAX 11 PROLOG II

0 y el 99.999.999 el LPA micro-PROLOG Professional

"Enteros negativos" comprendidos entre el

- 32.768 y el 0 el Turbo Prolog.

-16.777.216 y el 0 el VAX 11 PROLOG II

-99.999.999 y el 0 el LPA micro PROLOG Professional

6.4. SEGUIMIENTO O TRAZA DE UN PROGRAMA

Vamos a tratar de una herramienta doblemente útil que nos proporciona el Prolog: La **traza** o seguimiento de un programa. Todas las versiones de Prolog llevan incluida una ayuda al programador, que consiste en indicar los diferentes pasos secuenciales que realiza un programa durante su ejecución. Su mision es poner al alcance del programador la posibilidad de comprobar que el flujo del programa es el deseado.

Esta utilidad, que emplearemos inmediatamente, nos va a proporcionar el medio de saber exactamente como resuelve el Prolog los problemas encomendados, con lo que ganaremos conocimiento y dominio. Por eso creemos que este es el mejor momento de poner esta inestimable herramienta a su servicio, para comprender al máximo los procedimientos.

Posteriormente le será de mucha utilidad cuando los programas o los procedimientos no se resuelven según los esperado, principalmente para determinar el porqué de una respuesta False desconcertante o de la particularización de una variable con un valor extraño. Tambien nos servirá para localizar el punto donde el procedimiento se mete en un bucle sin fin.

6.4.1 trace

La llamada a este servicio se hace por medio del directivo **trace**. Esta instrucción tiene varias opciones, pero la principal es ir listando secuencialmente los nombres y los valores de los diferentes procedimientos y variables que va tratando durante la ejecución del programa, deteniéndose en algunas versiones para que el programador tome alguna decisión al efecto.

El Turbo Prolog solo tiene el mandato TRACE al comienzo del programa y no se pueden seleccionar procedimientos aislados para la traza.

El sistema utiliza 4 tipos de mensajes para darnos su información:CALL, RETURN o EXIT, REDO y FAIL, o las iniciales respectivas C, R, E, y F . La descripción de cada uno de ellos, en general, es la siguiente:

6.4.2 CALL

Aparece cada vez que se llama a un predicado, es decir, comienza la satisfacción de un objetivo. El formato suele ser el nombre del predicado y los argumentos que lo componen.

6.4.3 RETURN o EXIT

Aparece cuando un predicado ha sido satisfecho con el formato del nombre del predicado y las variables particularizadas con los valores en ese momento.

6.4.4 REDO

Aparece cuando el programa esta en la marcha atras para resatisfacer un objetivo. Suele tener el formato del nombre del predicado y las variables particularizadas con los valores en ese momento.

6.4.5 FAIL

Aparece cuando un predicado ha fallado, tiene el formato del predicado con las variables particularizadas con los valores de ese momento.

6.4.6 notrace

El predicado incorporado notrace se utiliza para indicar al Prolog que ya no se desea la funcion trace hasta una nueva orden.

6.5. AYUDA A LA DEPURACION DE UN PROGRAMA

En algunas versiones existen unos predicados incorporados que ayudan al programador en la depuración de los programas, sobre todo cuando estan en el modo interactivo. Cuando la base de hechos es extensa y sobresale el numero de lineas de una pantalla, seria muy útil conocer

todas las cláusulas donde aparezca una misma constante o todos los predicados que tienen un mismo nombre o que aparezcan en la pantalla solo aquellos que nos interesan del total.

Los predicados pp y listing nos ofrecen la solución.

6.5.1 pp

Este predicado pp (X), cuyo nombre tan estraño es comprensible si se sabe que es la abreviatura de "pretty printer" algo asi como "impresión amable" y tiene por objeto imprimir o visualizar todas las clausulas de la base de hechos que contengan un argumento particularizado a X, no importando ni la cabeza de la cláusula ,ni el número de argumentos. Apareciendo en el mismo orden que están en la base de hechos. En el caso de que X estuviese particularizado a un nombre que no aparezca en la base de hechos,el objetivo resultaría fallo.

6.5.2 listing

el objetivo listing tiene por objeto la impresión o visualización por la pantalla de todas las cláusulas cuya cabeza es X que haya en la base de hechos, independientemente del número de argumentos. Siempre que X esté particularizado al nombre de una cláusula que se encuentre en la base de hechos será éxito.

EJEMPLOS

En una pista de baile se encuentran tres parejas de amigos.Los tres chicos van vestidos de colores: rojo, verde y azul y azul respectivamente y las tres chicas van vestidas con los tres mismos colores.

Durante el baile cuando la chica de verde pasa delante del chico de rojo le comenta:

¿ Te has fijado que en ninguna de las tres parejas coinciden los colores del vestido del chico con los de la chica?.

La pregunta es:

¿Como van vestidas las parejas en este momento?

Veamos como lo resuelve Prolog.

```
chico (verde)

chico (rojo)

chico (azul)

chica (verde)

chica (rojo)

chica (azul)

incompatible (X,Y) if chico (X) and chica (Y) and X = "rojo"

and Y = "verde".

incompatible (X,Y) if X  Y.

incompatible (X,Y) if X  "rojo"

and not (incompatible (rojo,Y)).

incompatible (X,Y) if Y  "verde"

and not (incompatible(X,verde)).
```

pareja (chico,chica) if chico (chico) and chica (chica)

and not (incompatible(chico,chica)).

Si ahora le marcamos el objetivo a Prolog:

Goal:pareja(chico,chica)

Prolog nos dará el resultado correcto:

chico = verde,chica = rojo

chico = rojo,chica = azul

chico = azul,chica = verde

Cuatro chicas de un piso de estudiantes (Mónica,Montse,Mabel y María), el sábado por la noche realizan cuatro acciones distintas (pintarse las uñas, leer, maquillarse y peinarse).

si sabemos que:

Montse no se está maquillando ni leyendo.

Mabel no está pintándose las uñas ni maquillándose.

María no se esta pintando las uñas ni está leyendo.

Mónica no se esta maquillando ni leyendo.

Si Mónica no se pinta las uñas tampoco lo hace María.

Todas ellas estan realizando acciones distintas.

La pregunta es:

¿que hace cada una de ellos?

El problema es similar al anterior aunque un poco más complejo, el Programa en Prolog que lo resuelve de forma correcta es el siguiente:

nombre(mónica).

nombre(montse).

nombre(mabel).

```
nombre(maria).

acción(uñas).

acción(leer).

acción(maquillarse).

acción(peinarse).

incompatible(montse,maquillarse).

incompatible(montse,leer).

incompatible(mabel,maquillarse).

incompatible(mabel,uñas).

incompatible(maria,leer).

incompatible(maria,uñas).

incompatible(mónica,leer).

incompatible(mónica,maquillarse).

incompatible(mónica,uñas) if incompatible (maria,uñas).

incompatible(X,Y) if X  "mónica"

and not (incompatible(mónica,Y)).

hace (X,Y) if nombre (X) and acción (Y)

and not(incompatible(X,Y)).
```

Ahora marcamos el objetivo a Prolog:

```
Goal:hace(Nombre,Acción)
```

a lo que Prolog nos responde:

```
Nombre = Mónica,Acción = Peinarse

Nombre = Montse,Acción = uñas
```

Nombre = Mabel, Acción = leer

Nombre = Maria, Acción = maquillarse

De nuevo le invitamos a que pruebe con el programa, variando el nombre de los hechos y de las reglas.¿que sucede si quitamos la última regla del tipo incompatible?.

6.5.3 EJEMPLO DE TRAZA DE UN PROGRAMA

Para ver el comportamiento del Prolog durante el seguimiento vamos a retroceder al punto donde se tecleó el objetivo después de hacer la selección de TRACE:

Goal:hace(Nombre,Acción)

Y el Prolog comienza la resolución y al tiempo va enviando los siguientes mensajes que le serán fáciles de interpretar. Están todos sólo hasta llegar a la primera solución. Además para ahorrar páginas hemos agrupado los mensajes idénticos poniendo por nuestra cuenta "aquí tantos XXXX iguales". siendo XXXX el tipo de mensajes.El espaciado entre mensajes distintos tambien es nuestro.

CALL:nombre(_)

RETURN:nombre(mónica)

CALL:acción(_)

RETURN:acción (uñas)

CALL:incompatible("mónica","uñas")

REDO:incompatible ("mónica","uñas") (repetido 8 veces)

CALL:incompatible ("maria","uñas")

REDO:incompatible("maria","uñas") (repetido 6 veces)

RETURN:incompatible("maria","uñas")

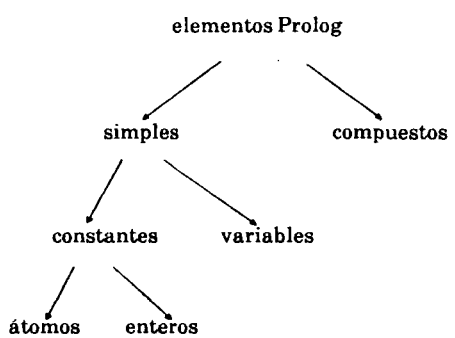
RETURN:incompatible("mónica","uñas")

REDO:acción (__)

6.6. LOS ELEMENTOS COMPUESTOS

Vamos a tratar las combinaciones de los diferentes elementos u objetos que formas los términos de un programa Prolog. Ahora veremos que los elemento a su vez pueden ser simples o compuestos:

los simples se dividen en constantes y variables y a su vez los constantes se dividen en átomos y en enteros.



Una vez conocidos los elementos simples del Prolog vamos a combinarlos para formar otros elementos compuestos o estructurados. La utilidad de estos elementos compuestos se pone de manifiesto al poder referirnos a un elemento compuesto como si se tratara de un elemento simple. Veamos un ejemplo práctico,aunque poco original. Se trata de cualquier directorio y en particular el de una biblioteca.

Cada ficha tiene el nombre del libro, el nombre del autor, la editorial y el número de referencia del libro dentro de la biblioteca.De momento podemos referirnos al elemento libro como uno de los muchos que existen en la biblioteca con el término:

corresponde(ficha1,libro).

que viene a decir que la ficha 1 corresponde al libro y

corresponde (ficha 2,libro)

que la ficha 2 corresponde al libro, pero no nos dice mucho en el sentido de que seguimos sin saber qué libros son pues suponemos que tienen un título y que hay más de uno, aunque este es un formato que conocemos bien por haberlo tratado en los ejemplos anteriores. Podríamos, por supuesto, sustituir libro por su valor concreto, con lo que tendríamos:

corresponde (ficha 1, Prolog).

Y

corresponde (ficha 2,Prolog 2).

que es un método muy válido pero que en este caso no nos sería muy útil porque ademas del título queremos guardar y recuperar el nombre del autor y de la editorial y donde se puede encontrar en las estanterías de la biblioteca. Toda esta información se puede colocar una detrás de otra mientras el total de caracteres lo admita como un sólo átomo.Pero el caso es que también deseamos acceder en otro momento al nombre del autor o al de la editorial. Para no alargar demasiado vamos a estructurar el elemento libro con cuatro componentes: el título, el autor, la editorial y el número de referencia. La forma de estructurar es colocar entre paréntesis los componentes separados por comas y fuera del paréntesis el functor. En nuestro caso tendremos que el elemento compuesto libro es abreviadamente:

libro(títilo,autor,editorial,referencia).

con lo que tendriamos en el

```
corresponde(ficha1,libro(prolog,bratko,wesley,12345)).
```

Al igual que hemos estructurado libro puede interesarnos descomponer el autor (para separar los de igual apellido y diferente nombre) ya que los componentes de una estructura pueden a su vez ser elementos compuestos, con autor (apellido,nombre):

```
corresponde(ficha1,libro(prolog,autor(bratko,ivan),wesley,12345)).
```

¿como utilizar esta información?. Los elementos compuestos obedecen las mismas reglas que los simples, por tanto todas las características del Prolog vistas hasta ahora son aplicables a los elementos compuestos y en el caso de tener que consultar en el directorio por las ficha correspondiente a algún libro que se llamase Prolog preguntaraímos como:

```
Goal:corresponde(ficha,libro(prolog,autor(ape,nom),editor,Nserie).
```

que nos permite comprobar que se pueden utilizar los datos estructurados, pero aún nos quedan muchas cosas por ver, por ejemplo simplificar un poco más la pregunta anterior utilizando la variable anónima, una simplificación podría ser:

```
Goal: corresponde (ficha,libro(prolog,autor(_,_)_,_)).
```

o más sencillamente:

```
Goal:corresponde (ficha,libro(prolog,_,_,_)).
```

o todavía más simple:

```
Goal:corresponde (ficha,libro(prolog,_)).
```

pues el Prolog admite cualquier número de variables anónimas dentro de la misma cláusula. Otra pregunta podría ser la de averiguar que libros hay de un cierto autor:

```
Goal: corresponde (ficha,libro(_,autor(bratko,_),_,_)).
```

6.7. ARBOLES

Una representación de los elementos compuestos la forman los árboles que definiremos como:

"Conjunto de elementos, llamados vertices, unidos por segmentos orientados, llamados arcos, que cumplen las tres condiciones siguientes:

* A.- Existe un único vértice, llamado inicial o raiz, en el no termina ningun arco.

* B.- Se puede crear para todos los vértices del árbol una secuéncia ordenada de ramas que empiezan en el vértice inicial y acaban en el citado vértice.

* C.- En todo vértice, excepto en el inicial, termina un solo arco.

los vértices se suelen identificar por un número. El elemento que representa el vértice recibe el nombre de vértice descendiente: Se llama vértice descendiente de un vértice V a cualquiera de los vértices que llega un arco que parte de V. Por convenio se dice que un vértice es descendente de sí mismo.

Por analogía con un árbol de la naturaleza o genealógico, muchos de los conceptos del árbol gráfico tienen nombres vegetales o de relaciones familiares:

* RAIZ es el vértice inicial.

* RAMAS son los arcos.

* HOJAS son los vértices o nodos terminales.

* HIJOS son los nodos descendentes

* PADRES son los nodos de los que llegan los arcos.

* DESCENDIENTE son los vértices descendentes.

* ANTECESOR son los vértices de los que parten los arcos.

- Se llama árbol ordenado aquel en que el orden de las ramas es significativo. Es decir que dos árboles ordenados con el mismo número de elementos pero en distinto orden resultan distintos.

- Se llama vértice interior al que tiene un nodo ascendente y otro descendente, o de otro modo el que no es ni el inicial ni un terminal.

- Se llama longitud del camino V al número de arcos que hay que recorrer desde la raiz hasta el vértice V.

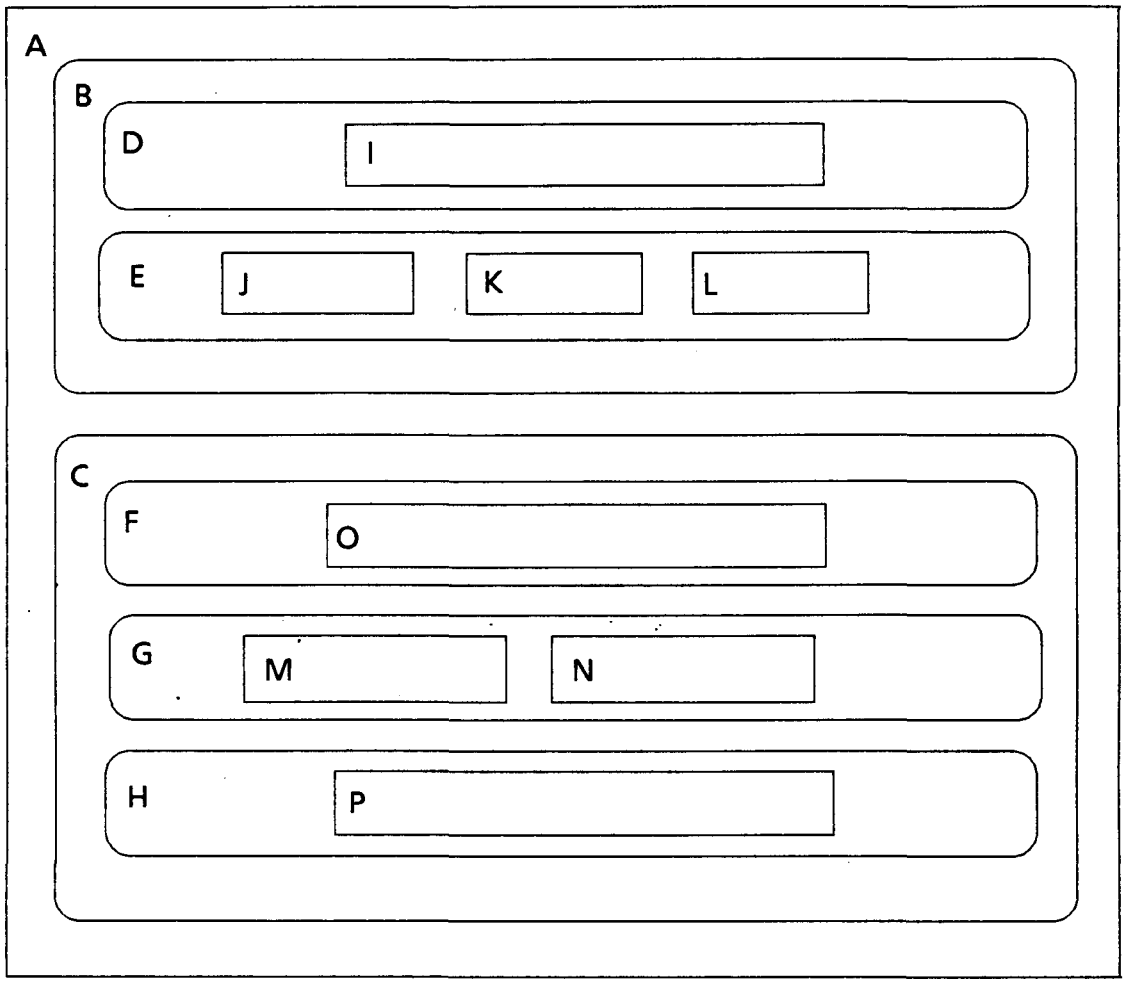
- Se dice que un nodo N esta en nivel i, su descendente D estará en el nivel i + 1. De igual modo, si un nodo N está en el nivel i,su antecedente estará en el nivel i-1. Por convenio se establece que el vértice inicial o raiz está en nivel 1.

- Profundidad o altura de un árbol es el número máximo de los niveles de todos los elementos de un árbol.

- Se llama subárbol al árbol obtenido a partir de un vértice.

- Se llama grado de un vértice interior al número de descendentes directos que parten de él.

Un árbol binario es aquel cuyo grado es 2.

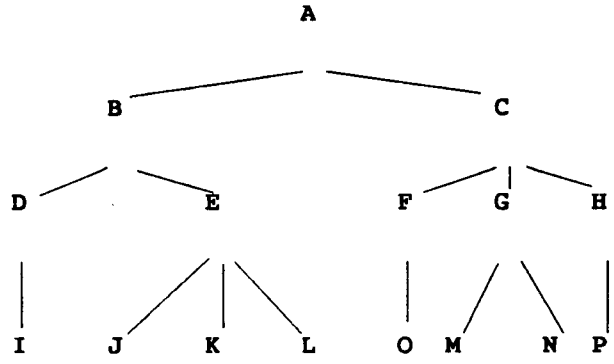


Un árbol.

Dados los elementos del árbol de la figura anterior, cuya representación es:

(A(B(C(I),E(J,K,L)),C(F(O),G(M,N),H(P))))

Puede representarse «por niveles» como la siguiente figura:



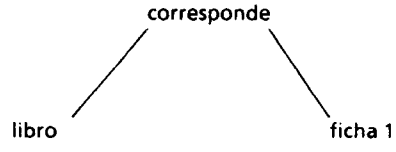
Representación gráfica del árbol de la figura anterior

Aunque internamente Prolog represente el conocimiento como un árbol, en la programación con Prolog (excepto el Lenguaje Natural) apenas se manipula con árboles, empleándose las listas que son una clase especial de árboles, como se verá extensamente con posterioridad.

El conocimiento (o hechos) se representa con árboles, donde la cabeza del predicado es el functor del árbol y los argumentos son los componentes del árbol. Por ejemplo:

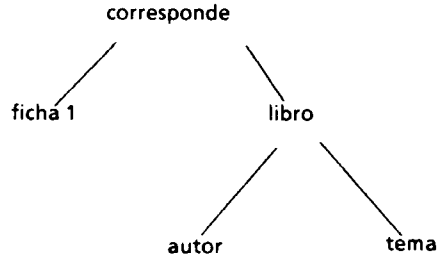
corresponde (ficha 1,libro).

vendría representado por:



Si a su vez el elemento libro fuese un elemento compuesto como:

corresponde (ficha 1, libro (autor, tema)).



su representación gráfica sería:

6.8. Las Listas.

Las listas son una representación especial de los árboles (a las listas se les denomina árboles degenerados porque cada nodo tiene como máximo un subárbol), muy utilizadas en la programación simbolica pues llegan a poder representar a casi todo: datos, funciones, programas, etc. Hay un lenguaje de programación, el LISP, en el que la estructura básica son las listas. *Una lista es una secuencia ordenada de cualquier número de elementos.* Los elementos pueden ser simples o a su vez, ser otras listas cuyos elementos contengan a su vez otras listas y así sucesivamente.

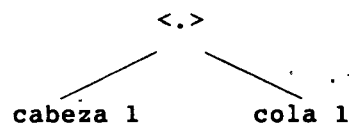
La longitud de una lista es el número de elementos que contiene. Conviene recalcar que la longitud de una lista hace referencia a los elementos y no a la totalidad de elementos que pueden obtenerse de sumar los elementos de una lista con los elementos de los elementos cuando éstos a su vez son otras listas.Los elementos de una lista también pueden ser variables que serán particularizadas en las mismas condiciones que las demás variables.

El Prolog, al igual que otros lenguajes de programación, al emplear las listas recurren a una doble representación: la interna y la externa.

La notación interna o notación de punto es la representación de un árbol cuyo functor es un punto >.< y los argumentos a los que se denominan cabeza y cola de la lista. La cabeza de la lista es el primer elemento y la cola son todos los restantes. Se denomina lista vacía la que tiene de longitud 0 elementos y se representa por un corchete abierto y otro cerrado []. En esta notación de puntos cada lista se va descomponiendo en otras dos: la cabeza y la cola, con lo que la representación de la lista sería:

(cabeza 1, cola 1).

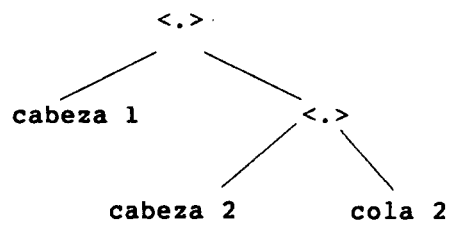
cuya representación gráfica es:



que a su vez sería:

(cabeza 1, . (cabeza 2, cola 2))

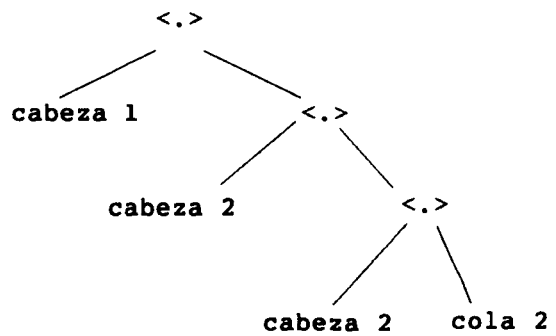
y gráficamente



y a su vez

(cabeza 1, . (cabeza 2,. (cabeza 3, (cola 3))))

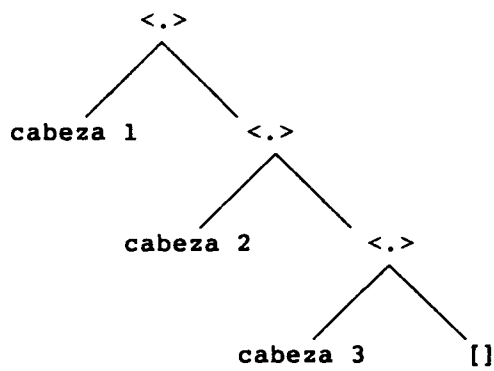
y gráficamente



y representando el final de una lista como la lista vacía

(cabeza 1, . (cabeza 2, . (cabeza 3, (....., [])))

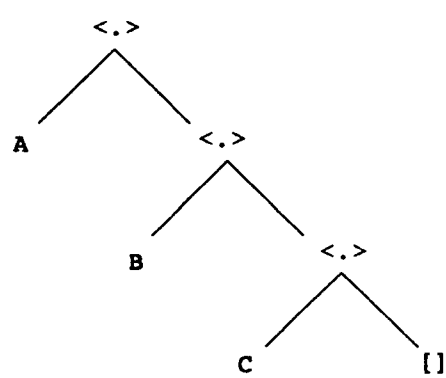
la representación gráfica en forma de árbol sería:



Si tuviéramos una lista con los tres átomos (es decir elementos simples) su representación interna sería:

.(A, . (B, . (C, [])))

y su representación gráfica:



El functor <.> punto de define internamente como un operador, por lo que la lista anterior puede representarse como

A . B . C []

Como la manipulación de las listas se hace a través de la descomposición en cabeza y cola, hay una notación especial para representar una lista por medio de la cabeza y la cola. Cada versión particular de Prolog tiene algún signo especial para representar la lista. En TurboProlog emplearemos la notación:

[X.Y]

que quiere decir: «la lista cuya cabeza es X y cuya cola es Y»

La ***notación externa o notación de listas***, es una forma más facil de representar las listas. En la notación de listas, una lista se representa como una serie ordenada de elementos encerrados entre paréntesis y separados los elementos por espacios en blanco, por comas o por algún otro signo que determine la versión particular de Prolog. Cuando se desee que los elementos de una lista se representen por sus caracteres en codigo ASCII, dentro del paréntesis encerraremos a estos elementos entre dobles comillas.Por tanto la lista:

(A..(B..(C,[])))

en notación de listas se representa como:

(A B C)

o tambien como:

(A,B,C)

Si algunos de sus elementos son a su vez listas, se representará:

(A(B C)(F(G H I(J K)))L)

donde el segundo elemento es una lista con dos elementos, el tercer elemento es una lista cuyo cuarto elemento es a su vez otra lista.Para mayor claridad siempre utilizaremos las comas para separar los elementos y recordaremos:

* a.- Los elementos de la lista están encerrados entre paréntesis.

* b.- Los elementos de la lista están separados por comas.

* c.- El orden de los elementos altera el resultado.

* d.- Los elementos de una lista pueden a su vez ser otras listas.

* e.- La longitud de una lista es el número de elementos que tiene.

* f.- La lista de longitud 0,es decir sin ningun elemento,se llama lista vacia y se representa como: [].

* g.- Al primer elemento de una lista se le llama cabeza de lista.

* h.- Se llama cola de una lista a la lista formada con todos los elementos menos el primero

* i.- La lista vacía no tiene cabeza ni cola.

* j.- La representación de una lista en cabeza y cola es :[X,Y].

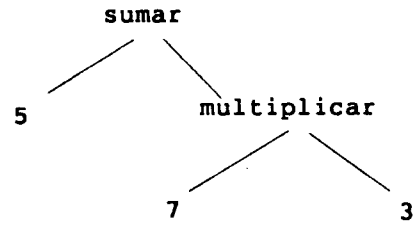
6.9. Los operadores

Algunos caracteres o signos especiales tienen para el Prolog una significación especial, por ejemplo el signo « + » se asocia a la operación de sumar. Estos caracteres especiales son los operadores y suelen emplearse en las operaciones aritméticas.En éstas se emplea en lugar del formato compuesto del functor y los argumentos entre paréntesis como p.e. +(5,3) el formato donde el operador se encuentra entre los argumentos, como p.e. 5+3, por similitud con la forma usual en la vida normal. La expresión +(5,*(7,3)) aparentemente parece más compleja en esta forma que en la de 5+(7*3). pero no debemos dejarnos atraer por lo aparentemente fácil. En Prolog, y en el mundo de los árboles en general, es mucho más sencilla la representación de la

primera expesión, y si sustituimos los signos + y * por los nombres de sumar y multiplicar resulta una expresión con un formato muy familiar:

sumar(5,multiplicar(7,3))

que es una estructura o elemento compuesto y cuya representación en árbol sería:



Es muy importante recordar que las expresiones arítmeticas no llevan a cabo ninguna operación arítmetica sino que son sólo una forma de estructurar los datos donde 7 + 3 tiene la misma estructura que +(7,3) que no tendria el valor de 10 sino que en el caso de equipararse con 10 sería éxito. Hay muchos operadores en los lenguajes de programación, pero veremos principalmente los operadores arítmeticos.

Las propiedades de los operadores son la posición, la prioridad y la asociatividad.

6.10. La posicion

La posición de un operador respecto de sus argumentos puede ser:

- **Infija:** cuando el operador esta entre sus argumentos:

3+5 8/4 7*6

- **Prefija:** cuando el operador esta delante de los argumentos:

-5 -4 +2

- **Sufija y Postfija:** Cuando el operador va detras del argumento,por ejemplo el factorial de un número: 5!. Lo que da origen a que los operadores se llamen respectivamente infijos, prefijos y sufijos.

6.11. LA PRIORIDAD

La prioridad sirve para indicar en que orden deben realizarse las operaciones. Si nos encontramos en el caso de x+y*z no es lo mismo primero resolver x+y y el resultado multiplicarlo por z que encontrar primero el producto entre y por z y luego sumarle el resultado a x. Un método sencillo sería poner la expresión primitiva en el formato +(x,*(y,z)) donde primero habría que resolver el segundo argumento (Y*Z) y el resultado aplicarlo al primer argumento para hallar el resultado final. Otra forma de evitar la duda hubiera sido el uso de paréntesis en la expresión inicial que hubiera quedado como x+(y*z). El Prolog por sí mismo suministra una clave de prioridad para cada operador, que depende de cada versión particular de Prolog, en general podemos afirmar que es un número entero que en algunas versiones está comprendido entre 1 y 255, siendo el 1 el de mayor prioridad y el 255 el de menor. En otras versiones como en el TurboProlog, el número esta comprendido entre el 1 y el 3, siendo por el contrario el tres el de mayor prioridad y el 1 el de menor.

6.12. LA ASOCIATIVIDAD

La asociatividad sirve para deshacer la ambiguedad cuando hay dos operadores contiguos con la misma prioridad, como ocurre en el ejemplo 8/2/2, que da como resultado de (8/2)/2 el valor de 2 y de 8/(2/2) el valor de 1.Con la expresión equivalente de la original (8,/(2,2)) tampoco habría duda. Normalmente se emplean paréntesis, por comodidad, para quitar las dudas, pero debemos saber que los operadores llevan asociados además de la prioridad una clave de asociatividad. Esta puede estar representando al operador por la letra f y sus argumentos por x e y respectivamente:

- **Asociatividad Izquierda:** Dice que el argumento puede contener operadores con la misma o menor clase de prioridad que el operador f y se representa por yfx o fy.

- **Asociatividad derecha:** Dice que el argumento puede contener operadores con clase de prioridad menor que el operador f y se representa por xfy o fx.

6.13. Las Comparaciones y las Operaciones Aritmeticas

Los predicados instalados o incorporados son aquellos cuya definición viene incluida en el propio Prolog. Ahora simplemente vamos a ver algunos que son necesarios para poder continuar. Son los predicados referentes a las comparaciones de igualdad y de mayor y menor.

6.13.1 IGUALDAD

El predicado de igualdad se representa por el carácter = (recordemos que algunos caracteres tenían significado por sí mismos). Pese a que su formato debería ser =(a,b) se representa como un operador infijo, es decir como: a = b, donde a y b son dos términos Prolog. Cuando se presenta el objetivo:

Goal: a = b

El Prolog intenta verificar si a y b son iguales, acabando con éxito solamente si lo son. No debemos ni pensar siquiera que el Prolog intente hacer igual a y b , sólo comprueba que lo sean. Como los terminos a y b pueden ser cualquiera, incluidas las variables, vamos a establecer alguna de las reglas por las que se rige la comparacion de igualdad en el Prolog.

* a.- Los enteros y los átomos son siempre iguales a sí mismos, p.e. los objetivos

Goal: giralda = giralda

Goal: esto_se_hunde = esto_se_hunde

Goal: 123 = 123

Acabarán como éxito, en cambio los objetivos.

Goal: giralda = sevilla

Goal: esto_se_hunde = socorro

Goal: 123 = 12300

* b.- Si a es una variable no particularizada y b es un termino con una constante o una variable ya particularizada, entonces a y b son iguales y ocasionará que a sea particularizada con el valor de b. Los objetivos.

Goal: A = a

Acaba como éxito y particulariza A como a

Goal: Futbolista (A) = Futbolista (Maradona)

Acaba como éxito y particulariza A como Maradona.

* c.- Si a y b son dos estructuras serán iguales si tienen el mismo functor, el mismo número de argumento y estos son iguales, independientemente del número de functores y de argumentos.

Los siguientes objetivos:

Goal: jugador (A, futbol) = jugador (Butrageño, futbol)

Acabará como éxito y además particularizará a A como Butrageño.

* d.- Cuando los 2 terminos a y b son dos variables no particularizadas, el objetivo acaba como éxito y las variables se quedan «compartidas» de tal forma que en cuanto una quede particularizada la otra lo será automaticamente.

6.13.2 No Igualdad

En Prolog existe tambien un predicado incorporado de no igualdad, que se representa <> o por \=. El objetivo a\=b o a<>b es éxito cuando falla el objetivo a=b y es éxito cuando falla el objetivo a=b.

6.13.3 Comparaciones de mayor y menor

Las comparaciones de mayor y menor se suelen hacer con los numeros enteros, pero esto tan solo es una costumbre pues veremos en algunos ejemplos de lenguaje natural como comparamos las palabras para ordenarlas alfabeticamente. De

momento nos referiremos a los numeros enteros. Vamos a ver otros predicados incorporados empleados en las comparaciones, tambien en forma infija entre los terminos a y b:

- a< b a es MENOR que b
- a>b a es MAYOR que b.
- a = < b a es MENOR o IGUAL que b
- a> = b a es MAYOR o IGUAL que b.
- a = b a es IGUAL que b
- a\ = b a NOes IGUAL que b.

Observe que se ha evitado el simbolo de = > que tiene una aplicación específica distinta de la comparación por mayor o igual.

Tan sólo como un ejemplo de la utilización de los predicados, pensemos en que una determinada aplicación necesita que se introduzca una fecha. Para evitar errores posteriores, hacemos una pequeña comprobación de la fecha, como puede ser que el día no sea superior al tope de los días en cada mes. Tendríamos en la base de hechos:

```
Tope ( Enero, 31 ).

Tope ( Febrero, 28 ).

Tope ( Marzo, 31 ).

Tope ( Abril, 30 ).

Tope ( Mayo, 31 ).

Tope ( Junio, 30 ).

Tope ( Julio, 31 ).

Tope ( Agosto, 31 ).
```

Tope (Septiembre, 30).

Tope (Octubre, 31).

Tope (Noviembre, 30).

Tope (Diciembre, 31).

Si hemos introducido la fecha como (dia, mes) podemos hacer una comprobación por medio del:

Goal: fecha_correcta(dia, mes) if tope (mes, tope_dia) and dia = tope_dia.

El objetivo acabará con éxito si el mes coincide con alguno de los de la tabla y ademas el día tecleado es menor o igual que el del tope al del mes correspondiente. Si hubiesemos tecleado (30, Enero), el objetivo sería éxito, pues particulariza día con 30 y mes con Enero. Si hubiesemos puesto (30, Enero) o (32, Enero), el objetivo acabaría como fallo. Por supuesto que podrían hacerse cosas mucho mejores pero esto es solo un ejemplo de como emplear los predicados de comparación.

6.13.4 Las Operaciones Aritmeticas

Para hacer operaciones aritméticas se utiliza el predicado incorporado <is>. Este realiza las operaciones arítmeticas contenidas en una expresión arítmetica. Recordemosque estas últimas se vieron cuando se estudiaron los operadores. Una expresión aritmética tiene la misma estructura que cualquier otro tipo de datos, lo único que necesita el predicado <is> es que su argumento de la derecha sea una estructura adecuada al cálculo númerico. El predicado is calcula el valor de la expresión aritmética donde todas las variables han de estar particularizadas y en caso de éxito particulariza las variables del argumento de la izquierda.

También recordar las restricciones que el propio Prolog impone, p.e. los enteros hasta un número muy alto. Las versiones actuales suelen traer suficientes facilidades para satisfacer las aplicaciones más usuales tanto en predicados incorporados como en la librería de programas . Por otra parte tener en cuenta que el Prolog es un lenguaje declarativo, donde el cálculo númerico no es tan esencial. Los operadores arítmeticos son:

a + b suma los números a y b

a-b resta los números a y b

a*b multiplica los números a por b

a/b divide el número a por b y da un cociente entero.

Dijimos que los operadores tenían unas características que eran la posición (prefija, infija y sufija), una prioridad (mod, / y * tenían más prioridad que + y -) y una asociatividad (derecha e izquierda).Para evitar la ambigüedad decidimos que en caso de duda pondríamos paréntesis.

6.14. La Recursion

La recursión es una técnica ampliamente utilizada en la programación y que poseen algunos lenguajes tales como PASCAL, LISP, LOGO y APL. Para introducirnos en la materia nos trasladamos con la imaginación a los tiempos de colegio en un día de clase.Y precisamente cuando nos están haciendo la pregunta de definir la recursión. Como no lo sabíamos responderíamos un poco al tuntún:

«la recursión es lo que sirve para hacer recursiones.»

En el colegio no pueden admitir esta respuesta, pero nosotros ahora sí. Y ya que hemos retrocedido al pasado recuerden aquel anuncio de un :

bote de leche condensada que se veía en la etiqueta a un niño en cuya mano sostenía un

bote de leche condensada que se veía en la etiqueta a un niño en cuya mano sostenía un

bote de leche condensada que se veía en la etiqueta a un niño en cuya mano sostenía un

bote de leche condensada que se veía en la etiqueta a un niño en cuya mano sostenía un

y así seguiríamos indefinidamente hasta la extenuación.

Decimos que un procedimiento es recursivo cuando se llama a sí mismo.

En la literatura de la programación simbólica hay verdaderas maravillas narrativas para describir la recursión.

Nosotros vamos a presentar la recursión inspirándonos en ***Chadwick (1986)*** y así la mostraremos con la ayuda de una familia de gnomos.

Encargamos a los gnomos resolver cuál es el factorial del número 3. Estos, que además de sabios son muy estrictos, consultan el libro de los gnomos que les dice:

«Para resolver el factorial de número N, cada hermano se guardará el número N y encargará a su hermano menor que resuelva el factorial de N-1. Y esperará hasta que su hermano menor le entrege el factorial de N-1. Cuando tenga el valor del factorial de N-1,calculará el producto del número

N por el factorial de N-1 y se lo entregará a su hermano mayor que le hizo el encargo. Cuando un hermano reciba el factorial de número 0 devolverá el valor de 1.»

Con estas normas tan precisas los gnomos (que les llamaremos A, B, C y D) se disponen a calcular el factorial de 3.

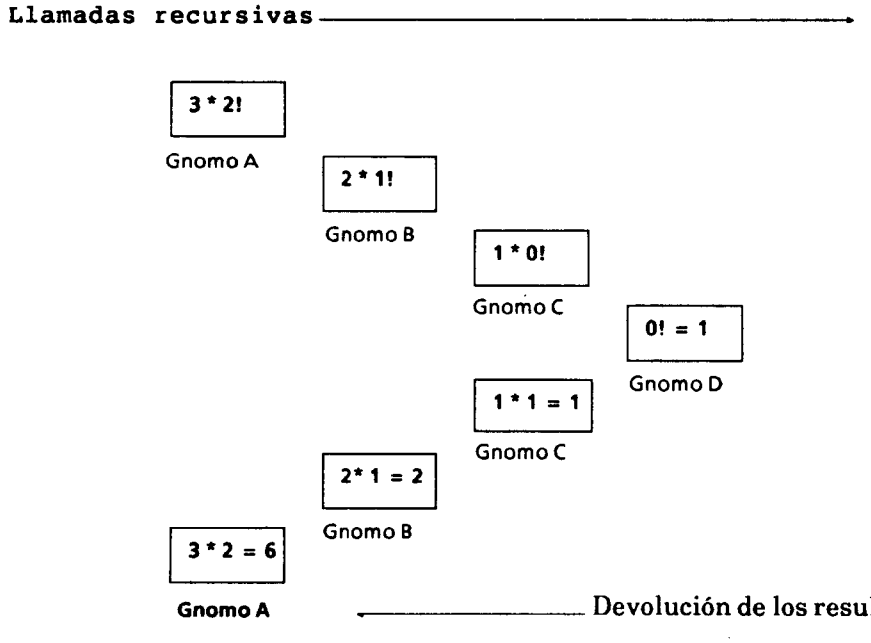
* El gnomo mayor, A, recibe el encargo de calcular 3!. Según sus normas, guarda el número 3 y le traspasa a su hermano menor, B, el encargo de calcular 2!.

* Este gnomo B se guarda el número 2 y le traspasa a su hermano C que calcule el valor de 1!. El gnomo C se guarda el número 1 y le entrega al gnomo D que calcule el valor de factorial de 0.

* El gnomo D, el más pequeño y más afortunado, sólo tiene que tender una mano y recibir el encargo y con la otra dar el resultado de 1. Con él empieza el camino de vuelta.

* El gnomo C recibe el valor del factorial de 0, que es 1, y lo multiplica por el número que se guardó en el viaje de ida. Una vez realizado este producto resulta el valor de 1! que devuelve a su hermano B.

* El gnomo B multiplica el número que tenía retenido y calcula el 2! que devuelve al gnomo A, que a su vez multiplica este valor de 2! por el número 3 con lo que, finalmente, calcula el valor de factorial de 3, que es 6.



Nosotros en lugar de tener gnomos tenemos el Prolog que hace la labor de todos los gnomos él solo. Pero debemos pedirle los procedimientos recursivos de un modo específico porque de lo contrario no responde de los resultados.

Las normas por las que se rige la formulación de un predicado recursivo son:

- 1.- Encontrar cómo se realiza la operación recursiva para el elemento general.
- 2.- Reducir la operación a un elemento general menor.
- 3.- Determinar la operación que determina la recursión.

En el ejemplo del cálculo del factorial del número 3, que generalizando al número N, tendremos:

- 1.- La operación de calcular el factorial del número N, para todo N<0, es el producto de todos los números desde el hasta el 1, es decir:

N*(N-1)*(N-2)*(N-3).....3*2*1

- 2.- Reducir la expresión del la operación a un termino general menor,en este caso si en el paso 1 era el término N ahora ponemos la operación en función del término N-1.

N! es lo mismo que N*(N-1)!

- 3.- Encontrar la forma de detener la recursividad, en este caso cuando se llega al factorial de 0 que por convenio es 1.

Con estas directrices vamos a formular en dos sentencias el predicado factorial (N,Facto) que es éxito cuando el factorial del número N es Facto:

```
factorial (0,1).
```

```
factorial (N,Facto) if N1 is N-1 and

factorial (N1,Facto N1) and

Facto is N* Facto N1.
```

La primera sentencia es la condición de final de la recursividad, cuando se llega al factorial de 0, que devuelve directamente el valor de 1. Igual podría haberse definido factorial (1,1) pero en ese caso deberíamos haber puesto la condición de que N sea mayor que 0, cosa que puede ser necesaria en algunas versiones.

La segunda sentencia es el cálculo del caso general, por medio del predicado incorporado "is", se calcula recursivamente cuanto vale el factorial del término N-1 con el propio predicado factorial. Despues se multiplica el factorial de N-1 (Facto N1) por el número N con lo que resulta el factorial de N.

6.15. La Manipulacion de listas

En esta parte vamos a ver como tratar las listas y sus elementos con los conocimientos que tenemos hasta el momento.

6.15.1 La relacion de pertenencia

Llamamos relación de pertenencia de un elemento a la cirscunstancia de pertenecer a una lista. El predicado para comprobar la relación de pertenencia se llama member".

Lo que deseamos saber es si un elemento , por ejemplo enero está entre los elementos de una lista p.e. (enero, febrero, marzo).

Y vamos a verlo haciendo la pregunta ¿está el elemento en la cabeza de la lista?, ¿ está el elemento en la cabeza de la cola de la lista?, ¿ está el elemento en la cabeza de la cola de la cola de la lista? y así seguiríamos; ¿hasta cuando?. Hasta llegar a la respuesta satisfactoria o a la lista vacía, es decir, al último elemento. Es decir el elemento debe estar en la cabeza o en la cola de la lista, de lo contrario no está en la lista.

El predicado pertenece_a(x,y) será éxito si el elemento x pertenece a la lista y. Por lo que hemos dicho hasta ahora podemos suponer acertadamente que se nos presenta una ocasión de hacer efectivas las propiedades de la recursión.Dijimos que:

* a) El elemento x será miembro de la lista y si x es lo mismo que el elemento de la cabeza de la lista y, que lo escribimos en Prolog como:

```
pertenece_a(X,[X:-]).
```

Utilizamos la variable anónima porque no vamos a volver a nombrar la cola de la lista.En la recursión nos servirá esta cláusula como condición de final o de salida, cuando se encuentre que el elemento es el mismo que el de la cabeza de la lista.

* b) Si no está en la cabeza de la lista, estará en la cola de la lista. Es decir, estará en la cabeza de la cola. Y utilizaremos nuevamente el predicado pertenece_a para comprobar si está en la cabeza de la cola. Por tanto tendremos:

```
pertenece_a (X,[_:Y]) if pertenece_a (X,Y).
```

La segunda condición de final ocurre cuando se encuentra la lista vacía, es decir no está el elemento en la lista. Ambos predicados van juntos. Ahora si de nuevo

preguntamos si el elemento marzo pertenece a la lista (enero, febrero, marzo, abril) tendremos que definir los predicados de pertenencia que son:

```
pertenece_a(X,[X:_]).

pertenece_a(X,[_:Y]) if pertenece_a(X,Y).
```

Y el objetivo que será:

```
Goal: pertenece_a(marzo,(enero,febrero,marzo,abril)).
```

a lo que Prolog contestará:

```
True
```

Con el objetivo:

```
Goal: pertenece_a(mayo,(enero ,febrero,marzo,abril)).
```

en Prolog contestará

```
False
```

Ya que habrá preguntado recursivamente por el primer elemento de la lista, y luego por el segundo, y por el tercero, y por el cuarto, y despues habrá llegado en la segunda regla a la lista vacía, es decir al final, con lo que acaba la recursión originando que objetivo acabe en fallo.

6.15.2 CONCATENAR

Otro predicado muy importante, tanto o más que el de pertenencia, es el de concatenar "append".

El predicado concatenar (X,Y,Z) es éxito cuando dadas las listas X e Y, la lista Z está formada con la lista X(coinciden las cabezas de las listas X y Z) y la lista Y. O de otra forma, dadas dos listas X e Y encontrar otra lista Z que sea la unión (concatenación) de las dos.

El objetivo:

Goal: concatenar([un,dos,tres],[cuatro],

[un,dos,tres,cuatro]).

será éxito y responderá.

True

también

Goal: concatenar([un,dos,tres],[cuatro],[Z]).

será éxito y ademas nos dará como respuesta la lista Z:

Z= [un,dos,tres,cuatro]

Tambien el objetivo

Goal: concatenar([un,dos,tres],[Y],[un,dos,tres,cuatro]).

será éxito y ademas nos dará como respuesta la lista Y:

Y=[cuatro]

El programa concatenar tambien es recursivo y se compone de dos reglas o predicados con la misma cabeza,cuya definición es la siguiente:

concatenar([],L,L).

concatenar([X:L1],L2,[X:L3]).

(los nombres de las variables puede elegirlas ud. mismo)

La primera regla sirve:

- A.- El concatenar una lista con la lista vacía da como resultado la misma lista
- B.- Como condición de final, pues al llegar a la lista vacía se acaba la recursión.

en cuanto a la segunda regla tendremos en cuenta:

- A.- El primer elemento de la primera lista será siempre el primer elemento de la tercera lista.
- B.- A la cola de la primera lista se le añadirá la segunda lista para dar como resultado la tercera lista.

- C.- Para hacer esto último volvemos a usar el concatenar de una forma recursiva.

- D.- Cuando la primera lista llegue a ser la lista vacía, se llegará a la condición de final, con lo que acaba la recursión.

Insistimos en que este predicado es muy importante en el tratamiento de listas, pues tanto nos sirve para unir dos listas en una tercera, como para separar una lista en otras dos. Por ejemplo para unir dos listas:

```
Goal: concatenar([de,hoy,no],[pasa],[L3]).
```

nos responde con la lista concatenada

```
L3= =[de,hoy,no,pasa]
```

Para descomponer una lista en otras dos:

```
Goal:concatenar([si,lo,se],[L2],[si,lo,se,no,vengo]).
```

nos responderá con la segunda lista L2:

```
L2=[no,vengo]
```

¿Que pasará cuando no estén particularizadas las lista primera y segunda?.En general el Prolog actuará según el criterio del constructor de la versión. P.e. en PROLOG-86 TM ante el objetivo:

```
Goal: concatenar([A],[B],[a,b,c]).
```

responde con:

```
False
```

6.15.3 El ultimo elemento de una lista

Vamos a definir el objetivo (X,L) que tiene éxito cuando el elemento X es el último de la lista L. Lo definimos con dos cláusulas, siendo la primera la condición de final cuando sólo queda un elemento en la lista L. Observará que en esta cláusula primera en su parte derecha hay un signo extraño,el

!, que se llama "corte"(cut), cuya función es controlar la marcha atrás y que sólo se generen las soluciones que nos interesan. Ambos términos,corte y marcha atrás, se explicarán ampliamente más adelante.

La segunda cláusula es una llamada recursiva.

último([X],X) if !.

último([_:Y),X) if último(X,Y).

Con el objetivo

Goal: último([delfín],[el,último,es,el,delfín]).

la respuesta es:

True

tenemos el objetivo

Goal: último(X,[allá,en,el,rancho,grande]).

larespuesta es

X = grande

que nos dice que la variable X puede estar o no particularizada.

6.15.4 Dos Elementos Consecutivos

Vamos a definir el objetivo consecutivo (X,Y,Z) que es éxito si los elementos XeY de la lista Z son consecutivos.Por lo menos uno de los elementos X e Y debe estar particularizado.Se define con dos términos, el primero que hace de condición final y el segundo hace una llamada recursiva:

consecutivos (X,Y,[X,Y:_]) if !.

consecutivos (X,Y,[_:Z]) if consecutivos(X,Y,Z).

Tenemos el objetivo

Goal: consecutivos([romeo],[julieta],

[calixto,melibea,romeo,julieta]).

el objetivo es éxito y la respuesta es:

True

El objetivo

Goal: consecutivos(X,[gaspar],[melchor,gaspar,baltasar])

el objetivo es éxito y la respuesta es:

X = melchor

6.15.5 Invertir una lista

El objetivo invertir (X,Y) tiene éxito si la lista Y resulta de la inversión de los elementos de la lista X, es decir que el primer elemento de la lista X es el último de la lista Y, el segundo de la lista X es el penúltimo de la lista Y,etc. El programa invertir se define por medio de dos cláusulas, la primera es la condición de final cuando la primera lista queda sin elementos, es decir la lista vacía y cuya inversión es tambien la lista vacía. La segunda cláusula hace una llamada recursiva y hace una concatenación de la cabeza de la primera lista a la cola de la segunda. Sabemos que por definición la cola de una lista es siempre una lista, pero la cabeza de una lista no es siempre otra lista.Por tanto debemos encerrar entre corchetes la cabeza de la primera lista en el predicado concatenado.

invertir([],[]).

invertir([X:Y],L) if invertir(Y,Z),concatenar(Z,[X],L).

Si tenemos el objetivo:

Goal: invertir([un,dos,tres],[tres,dos,uno]).

acabará como éxito. Y el objetivo:

Goal: invertir([cinco__perros,mataron,cinco__lobos],X).

acabará con éxito y responderá:

X=[cinco__lobos,mataron,cinco__perros]

6.15.6 Borrar un elemento de una lista

Para borrar o eliminar todas las veces que aparece un elemento dado en una lista se emplea el predicado borrar (X,Y,Z) que quita todas las veces que aparece el elemento X de la lista Y creando la nueva lista Z.Este predicado se forma con tres cláusulas.La primera es lacondición de final,cundo la lista llega a ser la lista vacía.En la segunda y tercera, si no se encuentra el elemento en cabeza de la lista, se investiga si está en la cola por medio de la recursión.Tambien aparece el signo del "corte".

```
borrar(_,[],[]).

borrar(X,[X:L],M) if!,borrar(X,L,M).

borrar(X,[Y:L1],[Y:L2]) if borrar (X,L1,L2).
```

el objetivo:

```
Goal: borrar([uno],[uno,uno,dos,uno,tres,uno,cuatro],X).
```

será éxito y responderá:

X=[uno,dos,tres,cuatro]

6.15.7 Sustituir un elemento de una lista

El programa de sustituir un elemento por otro es muy parecido al de borrar un elemento.Dado el predicado sustituir (X,L,Y,M) creará la lista M con todos los elemento de la lista L excepto los elementos X que serán sustituidos por los elementos Y.Tiene tres cláusulas, la primera es la condición de final, cuando encuentra la lista vací a.La segunda cuando

encuentra al elemento X en el segundo argumento.La tercera cláusula cuando no encuentra al elemento X en el segundo argumento. Estas dos últimas son recursivas.

```
sustituir(_,[],_,[]).

sustituir(X,[X/L],Y,[Y/M]) if!,sustituir(X,L,Y,M).

sustituir(X,[Z/L],Y,[Z/M]) if sustituir (X,L,Y,M).
```

el objetivo:

```
Goal: sustituir([pobre],[yo,soy,muy,pobre],[rico],X).
```

acabará en éxito y respondrá:

```
X=[yo,soy,muy,rico]
```

6.15.8 Sublistas

Se dice que la lista X es una sublista de la lista Y si todos y cada uno de los elementos de X pertenecen en el mismo orden a los elementos de la lista Y. En este programa hacen falta dos predicados. El primero encuentra el primer elemento de la lista X que coincide con algún elemento de la lista Y. El segundo predicado comprueba que los restantes elementos coinciden uno por uno con los de la lista Y:

```
sublista[X:L],[X/M]) if prefix(L,M),!.

sublista(L,[_:M]) if sublista(L,M).

adelantado ([],_).

adelantado ([X:L],[X:M]) if adelantado (L,M).
```

Tendremos pues que el objetivo:

```
Goal: sublista([es,una,sublista],[es,una,sublista,de,la,lista]).
```

acaba con éxito. En este predicado ambos argumentos deben estar particularizados, de lo contrario Prolog dará varias respuestas.

6.15.9 Correspondencia entre listas

La operación de correspondencia entre listas es aquella que permite convertir una lista en otra aplicando a cada elemento de la lista original una función específica, dando como resultado la lista final.

```
correspondencia (_,[],[]) if!.

correspondencia(F,[H:T],[Y:M]) if F(H,Y),!,correspondencia(F,T,M).
```

6.16. CONJUNTOS

El concepto de conjunto se usa ampliamente en las matemáticas de todo tipo y el Prolog también está capacitado para trabajar con los conjuntos.Porque ¿qué son los conjuntos sino unas listas especiales?. En efecto recordemos someramente algunas características de los conjuntos que confirman su semejanza con las listas:

6.16.1 Conceptos

-DEFINICIÓN-

«un conjunto es una colección bien determinada de objetos que por definición son sus elementos.»

Los conjuntos se representan con letra mayúscula y los elementos con minúsculas.

El Simbolo ∈

Representa una relación entre un elemento y un conjunto; m ∈ X significa «m es un elemento del conjunto X» . Dado un elemento m y un conjunto X , siempre es posible determinar si se verifica que m ∈ X

(∈ significa en conjuntos " pertenece ").

IGUALDAD DE CONJUNTOS

Se dice que dos subconjuntos son iguales únicamente si los dos conjuntos son identicos, es decir, contienen exactamente los mismos elementos. Si dos conjuntos X e Y son iguales se escribe X = Y .

SUBCONJUNTOS

Si un conjunto X está compuesto íntegramente por elementos de otro conjunto Y , diremos que X es un subconjunto de Y.

CONJUNTO VACÍO

Es el conjunto que no tienen ningún elemento. Por definición el conjunto vacío es un subconjunto de todos los conjuntos.

Todas las definiciones dadas para las listas y que hemos visto anteriormente pueden aplicarse a los conjuntos. Sólo hay una salvedad:

La lista es una colección ORDENADA de elementos, donde el orden es significativo ya que una lista se diferencia de otra con los mismos elementos pero con distinto orden.

El conjunto es una colección de elementos en cualquier orden, no habiendo diferencias entre dos conjuntos con los mismos elementos aunque en distinto orden.

Para tratar en Prolog con los conjuntos vamos a proponer una serie de predicados, que al igual que hicimos con las listas, los ofreceremos más adelante en un apéndice. Estos predicados suelen venir en las librerías de programas de algunas versiones. Al no ser predicados instalados hay que introducirlos en los programas al igual que se hace con los datos o reglas.

6.16.2 Relacion de pertenencia

Antes de estudiar las operaciones con conjuntos vamos a ver la propiedad de pertenencia es decir, comprobar si un elemento pertenece a un conjunto, por medio del predicado pertenece_a:

```
Pertenece_a ( X,[X:_]).

pertenece _a(X,[_:Y]) if pertenece_a(X,Y).
```

SUBCONJUNTO DE UN CONJUNTO

Queremos encontrar un predicado que asegure que el subconjunto X pertenece al conjunto Y. Partimos básicamente del apartado pertenece_a pero con una modificación.

El conjunto vacío por definición es subconjunto de todos los conjuntos. Esto hace que la condición de final, en el predicado recursivo, esté en la segunda cláusula, en lugar de en la primera como en pertenece_a para detectar la lista vacía como final de la recursión.

Tendremos pues el predicado subconjunto

```
subconjunto ([A:X],Y) if pertenece_a (A,Y),subconjunto (X,Y).

subconjunto([],Y).
```

6.16.3 Operaciones con conjuntos

Las dos operaciones principales con conjuntos son la unión y la intersección, por las cuales se pueden obtener nuevos conjuntos a partir de otros.

INTERSECCIÓN

La intersección de dos conjuntos arbitrarios X e Y, es el conjunto formado por aquellos elementos que pertenece a X y a Y, y se escribirá XY o X.Y (en Prolog X*Y).

La definición de un predicado que satisfaga la operación de intersección no es facil por una razón: de antemano hay que presumir que las listas que se van a utilizar no tienen elementos duplicados.

UNIÓN

Dados dos conjuntos arbitrarios X e Y, se define la unión de X e Y como el conjunto formado por los elementos de X o de Y o de ambos, y se representa como X + Y.

El predicado unión es un predicado concatenar modificado por la condición de pertenecer al conjunto.

```
unión ([],X,X).

unión ([X:R],Y,Z) if
```

```
pertenece__a(X,Y),!,

unión (R,Y,Z).

unión([X/R],Y,[X/Z]) if

unión (R,Y,Z).
```

DISYUNCIÓN

Dados dos conjuntos arbitrarios X e Y diremos que son disyuntos cuando ningún elemento del conjunto X es subconjunto del Y.

Según esta definición el predicado disyunción es éxito cuando ningún elemento de X pertenece a Y.

```
disyunción (X,Y) if not ((pertenece__a (Z,Y))).
```

CUADRO DE LOS PREDICADOS QUE MANIPULAN CONJUNTOS

Determinar si un conjunto X es subconjunto del Y (en inglés subset)

```
subconjunto([A:X],Y) if pertenece__a (A,Y),subconjunto (X,Y).

subconjunto ([],Y).
```

Disyunción de dos conjuntos (en inglés disjoint)

```
disyunción (X,Y) if not ((pertenece__a (Z,X),pertenece__a(Z,Y))).
```

Intersección de dos conjuntos (en inglés intersection)

```
intersección([],X,[]).

intersección ([X:R],Y,[X:Z]) if

pertenece__a(X,Y),!,

intersección (R,Y,Z).
```

```
intersección ([X:R],Y,Z) if intersección (R,Y,Z).
```

Unión de dos conjuntos (en inglés union)

```
unión([],X,X). unión([X:R],Y,Z) if

pertenece__a(X,Y),

!,

unión([X/R],Y,[X/Z]) if

unión (R,Y,Z).
```

6.17. Las bases de datos

Una base de datos es una suma de información interrelacionada que forma un único ente.

Tradicionalmente se ordena la información por registros variables de un único elemento y por campos que son los valores que toma una variable en los distintos registros.

En una base de datos de "escritores" un campo sería nombres (Miguel,Juan Ramón...) y un registro sería (Miguel,de Cervantes, Saavedra).

Los datos pueden estar estructurados bien de forma estática, en la que el número de componentes queda fijado en una declaración, o dinámica en la que el número de los mismos puede variar durante la ejecución del programa.

Una base de datos se dice que es relacional cuando se puede acceder a dos o más ficheros de forma simultánea, es decir que se puede manipular información de diferentes bases de datos unidas por un dato común.

El Prolog permite de una forma sencilla crear y gestionar base de datos dinámicas y relacionales, gracias al caracter declarativo de Prolog.

En Prolog no existen diferencias entre la memoria de trabajo, los hechos y las reglas, todos están en el mismo nivel por lo que también se le llama base de datos uniforme.

6.17.1 Predicados que manipulan la base de datos

Existen cuatro predicados instalados en Prolog, que son: ASSERTA,ASSERTZ,RETRACT y SAVE.Estos predicados permiten la creación y mantenimiento de la base de datos, la gestión de las mismas se realiza directamente en Prolog como vimos en la base de hechos, encargándose el proceso de unificación de seleccionar los valores y la marcha atrás de obtener todos los posibles

ASSERTA

Este predicado introduce un hecho al comienzo de la base de datos .

asserta

Por ejemplo:

Goal: asserta(libro(cuento,michael_ende,momo))

hace que tengamos en la base de datos:

libro(cuento,michael_ende,momo)

si repetimos con otro hecho:

goal: asserta(libro(cuento,voltaire,micromegas))

nos queda:

libro(cuento,voltaire,micromegas)

libro(cuento,michael_ende,momo)

ASSERTZ

Este predicado introduce al final de la base de datos el hecho encerrado entre paréntesis.Es fácil recordar que asserta agrega al principio de la base de datos, como la letra A, y assertz agrega al final, como la Z.

siguiendo con el ejemplo:

Goal: assertz(libro(poesia,becquer,rimas_y_leyendas))

nos queda la base de datos:

libro(cuento,voltaire,micromegas)

libro(cuento,michael_ende_momo)

libro(poesia.becquer,rimas_y_leyendas)

La base de datos es única y en la misma pueden coexistir hechos con distintas estructuras.

Existe una variable del predicado assertz del tipo:

assertz(hecho,número de orden).

que nos permite introducir el hecho en la posición indicada por el número de orden.

RETRACT

EL predicado retract borra de la base de datos el primer hecho que encuentra y que se puede unificar con el hecho encerrado entre paréntesis.

Una ventaja de que la base de datos en Prolog sea uniforme es que los predicados de la base de datos pueden aparecer en las reglas sin que exista ninguna diferencia formal con los hechos.

Por ejemplo:

```
me__gusta(Título) if libro (Tipo,__,Título)

and Tipo = cuento.
```

Cuando la base de datos es consultada en una unificación se procede de forma secuencial desde el primer dato que aparece en la misma hasta el último.

Un ejemplo de la uniformidad podría ser la siguiente regla, variante de la anterior:

```
me__gusta(Título) if libro (cuento,__,Título)

or libro(poesia,__,Título).
```

Y un ejemplo del uso de estos predicados descritos pueden ser las siguientes reglas: Si un libro es de cuentos incluirlo en la base de datos.

En Prolog sería:

```
libro(Tipo,Autor,Titulo) if Tipo = cuento

and asserta (libro(tipo,autor,título)).
```

Borrar todos los libros de cuentos de la base de datos.

```
borrar if libro (Tipo,Autor,Título)

and Tipo = cuento

and retract(libro(Tipo,Autor,Título))

and fail.
```

Para modificar un registro de la base de datos es necesario primeramente borrar el registro que se desee modificar y seguidamente introducir el dato nuevo

SAVE

Este predicado permite copiar una base de datos de la memoria principal , que es donde trabajamos, a una memoria secundaria, que normalmente es un disco, dándole el nombre de la constante entrecomillada.

save("nombre base de datos")

Nosotros con el ejemplo queremos guardar nuestra base de datos, y le damos el nombre de biblioteca y el atributo de dba para que reconozcamos el tipo de fichero que es, cuando procedamos a la lectura del directorio del disco:

Goal:save("biblioteca.dba")

CONSULT

Este predicado instalado nos copia en la memoria principal el contenido de la base de datos almacenada en la memoria secundaria y que tiene por nombre el de la constante entrecomillada, y es de la forma:

consult ("nombre base datos").

Esta operación es previa a cualquier trabajo con la base de datos, de tal forma que si en una regla se hace mención a un dato de la base y esta no se ha "abierto" con el predicado consult la unificación fallará en este punto dando como resultado falso.

Ahora deseamos en nuestro ejemplo, volver a trabajar en la base de datos:

Goal: consult("biblioteca.dba").

Podemos llamar a tantas bases de datos como queramos, pero debido a la uniformidad, todas ellas se nos unirán y a la hora de guardarlas en la memoria secundaria aparecerá la unión de todas ellas en cada base de datos. Por ello es necesario guardar cada base antes de consultar una nueva, salvo que nuestra intención sea el unirlas.

6.17.2 Otros predicados

Un punto que debe cuidarse es que debido a que cada vez que se unifica una regla se crean nuevas variables internas no suele existir en Prolog a diferencia de otros lenguajes ningún predicado de puesta a cero de las variables y dado que la base de datos es uniforme, tampoco suelen existir predicados de borrado de la misma ya que tambien se borraría el contenido del programa.

NEW

En algunas versiones aparece el predicado new que borra toda la memoria de trabajo.

DELETE

El predicado delete que borra las zonas de memoria que se le especifiquen. Otros predicados de borrado que son muy practicos son:

ABOLISH

Este predicado instalado borra de la base datos todos los datos que tengan el functor especificado.

RECONSULT

Este predicado borra la base de datos especificada en la memoria principal antes de copiarla de nuevo en la misma, desde la memoria secundaria.Con lo que se evita la duplicación de los términos de la misma.

reconsult("nombre base de datos")

En las versiones de Prolog que carecen de estos predicados instalados es de utilidad saber cómo borrar en su totalidad la base de datos de la memoria principal una vez empleado save.

Empleamos el siguiente predicado secundario:

```
borra_memoria if retract (_) and fail.

borra_memoria.
```

El segundo predicado es realmente un hecho que evita que Prolog nos responda false, la explicación a esta contestación del Prolog se verá más adelante.

Como vemos en la base de datos se trabaja directamente en Prolog y no es necesario describir los procedimientos, basta con declarar los objetivos.

Existen versiones en las que la base datos admite tambien reglas, lo cual aumenta las posibilidades del Prolog al poder tener una base de conocimientos dinámica, es decir que puede aprender.

Por último conviene tener presente que Prolog no recuerda nunca los valores de las variables que se han obtenido al unificar una regla, sólo recordará si el resultado ha sido verdadero o falso, por lo que si los valores los

queremos utilizar en otro punto del programa, o resultan ser las soluciones que buscamos puede sernos de mucha utilidad el empleo de la base de datos.

A CONTINUACION UN EJEMPLO ACLARATIVO

****** *EL problema de los flamencos y los cocodrilos* ******

En un zoológico, el cuidador de los animales, joven algo bromista, le comunica al director la llegada de un envío de animales de la siguiente forma:

"Señor director, han llegado los cocodrilos y los flamencos, en total son 32 ojos y 44 patas".

A lo que inmutable contestó el director:

"Gracias por informarme del número exacto de ambas especies". El cuidador salió sorprendido por la respuesta del director.

¿ Cuántos animales de cada especie llegaron ?

El problema no es difícil de resolver y en Prolog el programa es:

```
inicio if cuenta (15)

cuenta (X) if not (resultado(X))

and Z is X-1

and cuenta (Z).

resultado (X) if Y is 16-X
```

```
and 2*X + 4*Y is 44

and write ("flamencos = ",X)

and nl

and write ("cocodrilo = ",Y)

and nl.
```

La resolución es generalizable a otros problemas de este tipo, la recursión permite implantar un contador decreciente mientras no se cumplan las ecuaciones, en cuanto se cumplan se corta la recursión. Para Prolog este corte por incumplimiento provoca no poder demostrar el objetivo, dando como resultado false, pero como hacemos que previamente nos escriba la solución no nos importa que Prolog aparentemente halla fallado.

La solución que da el programa es:

```
flamencos = 10

cocodrilos = 6

False
```

6.18. EL CONTROL

El proceso que sigue Prolog en las demostraciones debe conocerse, a fin de que el programador esté seguro de que la respuesta obtenida es correcta tanto cuantitativamente como cualitativamente.

Para esto es necesario conocer bien el funcionamiento de Prolog, y si aun así no fueramos capaces de estar seguros, podríamos utilizar algunas herramientas de seguimiento o trazado del proceso de demostración.

Prolog además admite dos tipos de control sobre programa en su proceso de demostración, que son:

- Búsqueda de todas las soluciones posibles, que se consigue con la marcha atrás, que provoca el predicado instalado fail.

- Restricción el el número o tipo de soluciones y en las reglas utilizadas que se consiguen con el corte de la marca atrás y las meta reglas.

El control de Prolog no sólo nos permite obtener exactamente las soluciones que deseamos sino que además nos permite ahorrar tiempo y memoria de trabajo, aspectos que van tomando importancia a medida que la base de hechos y de conocimiento crecen.

6.19. LA MARCHA ATRAS

Como ya hemos visto, cuando marcamos un objetivo, Prolog intenta demostrarlo basándose en las reglas y en los hechos.

El proceso de demostración se realiza mediante la unificación, repasemos su funcionamiento:

- Una variable libre puede ser unificada por un término, pasando a ser una variable particularizada para los siguientes términos de la regla.

- Una constante puede ser unificada con ella misma o con una variable libre.

- Una variable particularizada puede unificarse con una variable libre o con ella misma.

- Un hecho puede unificarse con otro hecho compuesto, si tienen el mismo functor y el mismo número de argumentos, unificándose cada argumento según los tres primeros puntos vistos.

Prolog unificará de forma secuencial los hechos y las reglas del primero al último, y en una regla procederá de izquierda a derecha, hasta que demuestre el objetivo, si no puede demostrarlo dará como resultado el valor falso y si puede demostrarlo separará el proceso de unificación dando como resultado el valor:

verdadero.

Cuando Prolog encuentra un hecho que resulta ser falso, busca otro que sea cierto, si esto no es posible , vuelve marcha atrás hasta el nudo o bifurcación del punto donde había tomado esa solución (que ha marcado internamente), toma otra nueva solución y vuelve a intentarlo y así cuantas veces haga falta, hasta agotar todas las posibilidades.

FAIL

Prolog también inicia una marcha atrás si se encuentra el predicado instalado FAIL aunque ya haya demostrado todos los subobjetivos(Fail = fallo).

la utilidad de este predicado está en llamar esplícitamente a la marcha atrás porque se desean todas las soluciones posibles que puedan generarse.

Otra utilidad está precisamente en detener el proceso para lo cual se combina con el !(corte) de tal forma que el predicado:

cualquier_predicado:-!,fail.

causará que al llegar el curso normal de la demostración al objetivo cualquier_predicado en el fail se abandone definitivamente dando fallo.

Vamos a ver un proceso completo, y para ello nada mejor que un sencillo ejemplo, en el que Prolog clasifica una serie de vehículos:

> Tenemos tres vehículos que son 2 bicicletas de marcas BH y Orbea, que naturalmente no tienen depósito para combustible ni tubo de escape y un automovil de la marca Renault que por supuesto tiene deposito para combustible y tubo de escape. Definimos que un vehículo tiene motor cuando tiene depósito y escape y cuando carece de ellos decimos que es un vehículo sin motor.

con esto vamos a ver como lo espresamos en Prolog toda esta información

```
/*1*/ datos(marca(BH),caracteristicas(no_depósito,no_escape)).

/*2*/  datos(marca(Orbea),características(no_depósito,no_escape)).

/*3*/      datos(marca(Renault),características(depósito,escape)).

/*4*/        motor(X) if datos(marca(X),características(Y,Z)) and
Y = depósito and Z = escape.

/*5*/    vehiculo_con_motor(X) if datos(marca(X),_) and motor(X).

/*6*/    vehículo_sin_motor(X) if datos(marca(X),_) and not(motor(X)).
```

/*7*/ vehículo(X)if vehículo__con__motor(X).

/*8*/ vehículo(X)if vehículo__sin__motor(X).

Los números contenidos dentro de las barras y los asteriscos /*...*/ no influyen en el programa pues son comentarios y no son tenidos en cuenta por Prolog.

Ahora le vamos a marcar un objetivo a Prolog:

Goal: vehículo (marca)

vamos a seguir el proceso que realiza Prolog:

* Busca un hecho o una regla con la que pueda unificar el objetivo,tomando la /*7*/.

* La regla /*7*/ es verdad si lo son sus condiciones, en este caso vehículo___con___motor.Busca entonces una regla o un hecho con el que unificar la condición de la regla,y lo encuentra en la regla /*5*/ y para que esta lo sea deben ser verdad dos subobjetivos que son datos y motor.

* Pasa a comprobar el primer subobjetivo y a verificar si lo puede unificar con el hecho /*1*/ X = bh.

* Prolog pasa ahora a un segundo subobjetivo e intenta unificarlo con la regla /*4*/.

* Para que sea verdad la regla /*4*/ deben cumplirse unos nuevos subobjetivos que son esta vez expresiones de tipo lógico.

* Los subobjetivos de la regla /*4*/ no se cumplen por lo que Prolog vuelve marcha atrás hasta la última bifurcación del árbol, es decir hasta dato.

* Prolog descarta el hecho que primeramente tomó y pasa al siguiente /*2*/, ahora vuelven a tomar valores todas las variables al unificarse de nuevo.

* Con el hecho /*2*/ falla de nuevo Prolog, es un proceso similar.

* Con el hecho /*3*/ Prolog consigue su objetivo.En principio Prolog pararía dando como resultado:

X = Renault

Sin embargo, como el goal lleva implícito un fallo(fail) que fuerza la marcha atrás, no termina el proceso y pasa a la siguiente regla, y pasa en este caso a la /*8*/.

* Prolog ahora tiene un nuevo subobjetivo que es vehículo_sin_motor, y que intenta demostrarlo para lo cual vuelve a repasar una vez más todos los hechos y las reglas encontrando que la regla /*6*/ puede unificar el objetivo(parte izquierda de la regla.

* Para que la parte izquierda de la regla sea cierta lo tiene que ser la derecha. Por tanto debe comprobar que lo son todas las expresiones de la derecha, de nuevo estamos como antes.

* Para esta regla. La /*6*/ , Prolog encuentra que se cumple para los hechos /*1*/ y /*2*/, aportando dos nuevas soluciones:

X = bh

X = Orbea

que con la anterior ya son tres soluciones, ahora de nuevo Prolog intenta seguir buscando soluciones pero ya no hay más reglas por lo que da por finalizado el proceso.

El árbol que ha desarrollado Prolog para verificar el objetivo marcado con este programa se puede representar

con el gráficosiguiente.

7. LENGUAJE PROLOG 4ª PARTE

7.1. EL CORTE

Hasta ahora solamente nos hemos preocupado de obtener todas las soluciones posibles, pero existen otras muchas cirscunstancias en las que no basta unicamente con partes de las mismas, en estos casos nos será de mucha utilidad el predicado instalado corte, representado por <!>.

En otras palabras, cuando Prolog ha iniciado una marcha atrás por alguna de estas razones:

* porque haya fracasado
* forzado por un fail
* forzado de forma implicita por un goal

nunca podrá superar un !.

Cuando Prolog avanza en el proceso de demostración el predicado! no tiene ninguna influencia en el proceso.

El predicado ! puede ponerse en cualquier punto de la parte de acciones y condiciones de una regla, como cualquier otro predicado.

La forma general de escribir una regla con ! es:

A fi a1 and a2 and ! and...and an.

En nuestro ejemplo anterior, si la regla /*6*/ fuera:

vehículo__sin__motor(X)ifdatos(marca(X),__) and ! and not(motor(X)).

Al objetivo:

Goal: vehículo(Marca)

Prolog nos contesta:

Marca=Renault

Marca=bh

«!» ha impedido que una vez verificada esta regla con el dato /*1*/, Prolog hiciera marcha atrás para tomar el siguiente dato /*2*/.

En este caso hemos tomado solamente como solución final la primera obtenida en la regla. Si hubiese fracasado Prolog con el primer dato, tampoco podría haber obtenido otro pues el predicado ! lo impide.

Otro ejemplo del uso del corte es limitar a una sóla, las reglas que se pueden activar:

A1 if ! and a11 and a12 and....and a 1n.

A2 if ! and a21 and a22 and....and a 2n.

An if an1 and an2 and.....and ann.

Solamente una y solo una de las reglas puede activarse.

En nuestro ejemplo si la regla /*7*/ fuera:

vehículo (X) if ! and vehículo__con__motor(X).

la solución a:

Goal: vehículo(marca)

sería:

marca = renault

CUTL

En algunas versiones aparece el predicado principal cutl (functor) que es un corte selectivo, es decir solo actúa en la marcha atrás del functor especificado.

Vamos a verlo con un ejemplo,consistente en la elaboración de aperitivos:

comida(almendras).

comida(aceitunas).

bebida(cerveza).

bebida(vino).

aperitivo(X,Y) if comida(X) and bebida (Y) and cutl (bebida).

al objetivo:

Goal: aperitivo(X,Y).

Prolog nos responderá:

X = almendra Y = cerveza

X = aceitunas Y = cerveza

ya que se ha impedido la vuelta atrás en los hechos con el functor bebida.

7.2. LAS METAREGLAS O MAXIREGLAS

Otro elemento de control muy importante, sobre todo en el desarrollo de sistemas expretos, con el que cuenta Prolog y que solo se encuentra implantado en algunas versiones son las metareglas o maxireglas.

Las metareglas pueden definirse como las reglas de las reglas y en términos de Sistemas Expertos contienen el conocimiento sobre el conocimiento, en otras palabras son un tipo especial de reglas que contienen la estrategia de la demostración y por tanto pueden modificar y controlar un proceso de demostración. Las metareglas permiten variar el orden de unificación de las reglas según unas variables y así forzar a Prolog que unifique de principio a fin o viceversa ,etc.

Tambien permiten la inhibición o exclusión mutua de reglas o bloques de reglas.

Permiten seleccionar el orden en que deben ser demostrados los subobjetivos no teniendo que ser obligatoriamente de izquierda a derecha.

Las metareglas en resumen, encaminan la unificación podando las ramas inútiles o poco útiles y seleccionando el orden de las necesarias, según el deseo del programador que impone las condiciones de actuación, lo que ahorra tiempo y memoria de trabajo.

Es en las metareglas donde reside toda la potencialidad no determinística del Prolog.

7.3. LAS PARADAS

En este punto existe una gran diversidad de predicados, pero los más usuales son:

7.3.1 *WAIT*

Este predicado instalado detiene momentáneamente la demostración, cuando Prolog lo encuentra el programa.

7.3.2 *STOP*

Este predicado instalado detiene la demostración del programa de una forma indefinida, es secundario, pues se puede definir como:

```
stop if ! and wait and fail.
```

Si se nos hubiera ocurrido el hacerlo mediante una recursión, hubieramos descubierto que un proceso recursivo tiene un límite muy cercano.

7.3.3 *EXIT*

Este predicado instalado no sólo detiene la demostración sino que ademas sale del modo interactivo.

7.4. Aclaraciones sobre la marcha atras

Hemos intentado exponer todo cuanto necesitamos saber acerca de la marcha atrás para emprender una larga andadura, pues el conocimiento completo de la marcha atrás llegará despues de la experiencia.

Esperamos que hayan quedado un par de puntos muy claros y no cometeremos los herrores muy extendidos que se oyen por ahí de quienes no han tenido un manual sobre Prolog.

"*Un desatino es decir que la marcha atrás cuando no puede hacer una unificación, vuelve por otro camino para poder hacerla de algún modo.*"

No hay duda que podemos no sólo encontrar la respuesta adecuada sino también comprender el porqué del error.Esto se debe simplemente a la falta de conocimiento exacto del mecanismo de la marcha atrás. Los errores más usuales son:

- El Prolog quiere obtener la demostración a toda costa.
- La marcha atrás elige su camino para conseguir la unificación.

El Prolog no tiene ningún interés especial por el número de demostraciones conseguidas, él simplemente, si consigue la demostración del objetivo con las cláusulas que existen en la base de datos, tiene éxito, de lo contrario falla y se queda tan tranquilo en cualquiera de los dos casos.

Por otra parte el mecanismo de la marcha atrás es totalmente monótono y transparente hasta la desesperación.

Otra cosa, es cuando por distribuir las cláusulas en la base de datos de un modo no apropiado, la marcha atrás en su camino nos devuelve una solución no esperada. El empleo correcto del corte, que en sí mismo es muy sencillo, nos evitará estas sorpresas.

7.5. PREDICADOS INSTALADOS

Vamos a tratar de los predicados instalados o tambien incorporados o predefinidos, que son aquellos cuya definición está incluida en el propio sistema Prolog ,es decir, el

programador no tiene que preocuparse en escribirlos, sólo tiene que llamarlos en su programa. Debemos recordar que hay otros predicados muy extendidos que se ofrecen en las librerias de programas, pero no son predicados incorporados, la diferéncia está en que aquellos siempre hay que definirlos en el programa como los demas predicados o reglas.

Sabemos que el Prolog es un lenguaje de programación en lógica, y que dado un objetivo tiene que demostrarlo por el principio de resolución.Pero hay ciertos objetivos que se apartan de la lógica general del Prolog, uno de ellos puede ser la orden de imprimir una linea o hacer sonar un timbre, donde todo el problema se reduce a crear una secuencia de instrucciones en código máquina sin conexión aparente con la programación lógica. Por otra parte, todos los constructores de sistemas Prolog intentan superar los servicios ofrecidos por las demas versiones, con la inclusión de predicados nuevos o incrementando las prestaciones de los ya existentes, aunque no tengan que ver con la programación lógica.

Vimos algunos predicados incorporados anteriormente según los ibamos necesitando para ampliar las posibilidades de experimentar el lenguaje Prolog.

Podemos asegurar que en general todos los predicados incorporados son semejantes. Los predicados instalados ademas de realizar su función específica puede que lleven a cabo otras acciones que de no tenerlas en cuenta podrían suponer algún retraso en la consecución de los objetivos finales.Los resultados de estas acciones, efectos colaterales de un predicado, son fácilmente controlables siguiendo las especificaciones del Manual de Usuario de cada versión de Prolog.

Otra razón para consultar el Manual es saber exactamente el formato del predicado y las consecuencias que puede acarrear el que una variable no esté particularizada cuando el predicado exige que lo esté.

7.5.1 *Predicados que manipulan las bases de datos*

LOAD

El predicado load (A) carga en memoria el contenido del fichero cuyo nombre se ha especificado en A. Se emplea para cargar el programa y los ficheros que mantienen bases de datos. El nombre del fichero debe ser un átomo. Debe cuidarse el no repetir el predicado load con el mismo fichero, pues borra lo anterior para dejar sitio al actual.

ASSERTA (hecho)

Este predicado introduce al comienzo de la base de datos un hecho.

ASSERTZ (hecho)

Introduce al final de la base de datos el hecho encerrado entre paréntesis. Resulta fácil recordar assert A

La diferencia entre los predicados load y assert está en que:

- load (A) carga el contenido de A en la memoria,borrando lo que hubiera de A en una carga anterior.

- assert(A) añade el contenido de A a la base de datos por lo que puede darse el caso de haber cláusulas duplicadas si no se repite el predicado assert.

RETRACT (hecho)

Borra de la base de datos el primer hecho que encuentra y que se puede unificar, con el hecho encerrado entre paréntesis.

Una ventaja de que la base de datos en Prolog sea uniforme es que los predicados de la base de datos pueden aparecer en las reglas sin que exista ninguna diferencia formal de los hechos. Recordemos que cuando la base de datos es consultada en una unificación se procede de forma secuéncial desde el primer dato que aparece en la misma hasta el último.

SAVE ("constante")

Este predicado permite copiar una base de datos de la memoria principal que es donde trabajamos a una memoria secundaria, normalmente un disco, dándole el nombre de la constante entrecomillada.

CONSULT ("constante")

Este predicado instalado nos copia en la memoria principal el contenido de la base de datos almacenada en la memoria secundaria y que tiene por nombre el de la constante entrecomilladas. Esta operación es previa a cualquier trabajo con la base de datos, de tal forma que si en una regla se hace

mención a un dato de la base y ésta no se ha "abierto" con el predicado consult, la unificación falla en este punto dando como resultado falso.

Podemos utilizar tantas bases de datos como queramos, pero debido a la uniformidad de todas ellas, se nos unirán y a la hora de guardarlas en la memoria secundaria, aparecerá la unión de todas ellas en cada base de datos. Por ello es necesario guardar cada base antes de consultar una nueva, salvo que nuestra intención sea el unirlas.

RECONSULT ("constante")

Este predicado es igual que el anterior consult, con la sola diferencia que las cláusulas leidas reemplazan a las ya existentes con el mismo predicado. El formato es idéntico a consult.

Algunas versiones admiten que el argumento de consult y reconsult sea una lista formada con los nombres de varios archivos con objeto de facilitar la labor de lectura en una sola cláusula.

LISTING

Este predicado listing (X) siempre tiene éxito y hace que todas las cláusulas en la base de datos que tengan como predicado el átomo representado por X, sean impresas en la terminal. Es de mucha utilidad en los rastreos o en la

comprobación de la disposición de las cláusulas de un predicado en particular. Este predicado no está en todas las versiones Prolog.

Los predicados que vamos a ver a continuación almacenan en la propia base de datos unas estructuras generadas por el propio programa, con objeto de poder utilizar tales estructuras en diferentes etapas de la demostración y que se mantengan a pesar de la marcha atrás.

RANDOM

El predicado random (A,B) particulariza B a un número entero elegido aleatoriamente entre el 1 y A. Este objetivo no se resatisface con la marcha atrás.

GENSYM

El predicado gensym proporciona un método para generar nuevos nombres de átomos a partir de una raíz dada. El Prolog recuerda el último número dado a una raíz para no crear nombres duplicados. Así el objetivo gensym particulariza a X con el valor de ejemplo 1 la primera vez, a la segunda lo particulariza con el ejemplo 2 y a la décima vez lo particularizará con ejemplo 10 y así sucesivamente. Este objetivo no se satisface con la marcha atrás.

FINDALL

Este predicado findall (X) determina todos los términos que satisfacen el predicado X. La definición completa de objetivo findall (X,O,L) es construir una lista L con todos los objetos del universo Prolog (X) que satisfacen el objetivo O. Para ello es necesario que O este particularizado a un término que Prolog puede entender que es un objetivo sin preocuparse de lo complejo que pueda ser.

7.6. PREDICADOS QUE INFLUYEN EN EL CONTROL

El curso normal del Prolog es, dado un objetivo, poder demostrarlo o satisfacerlo, es decir, que tiene éxito o por el contrario falla. Hay dos predicados que especifican que un objetivo acabe con éxito o fallo antes de finalizar el proceso, son respectivamente true y false.

7.6.1 TRUE

Este objetivo siempre acaba con éxito y sustituye la a veces penosa clasificación de los hechos para forzar a que un determinado objetivo acabe con éxito.

7.6.2 FAIL

El predicado fail hace que el Prolog nunca pueda unificar el objetivo, con lo que este falla y se inicia la marcha atrás.

7.6.3 OTROS PREDICADOS

Otros predicados que influyen en el control son los que lo detienen, tanto temporalmente como definitivamente. En estos predicados existe una gran diversidad según las diferentes versiones pero los más usuales son :

WAIT

Este predicado instalado detiene momentáneamente la demostración, cuando Prolog lo encuentra en el programa.

STOP

Este predicado instalado detiene la demostración del programa de una forma indefinida.Es secundario, pues no se puede definir como:

```
stop if ! and wait and fail.
```

EXIT

Este predicado instalado no sólo detiene la demostración , sino que además sale del modo interactivo.

7.6.4 Predicados que influyen en la marcha atrás

Hay dos predicados incorporados que alternan la secuencia normal de los sucesos durante la marcha atrás, éstos son el cut (corte) y repeat.

CUT

Este predicado incorporado, pues como tal ha de considerarse, quita las posibilidades de resatisfacción de los objetivos, principalmente para ahorrar tiempo y memoria en la búsqueda de soluciones que a priori sabemosque no son necesarias. se representa con el símbolo !.

REPEAT

El predicado repeat proporciona un método para generar múltiples soluciones a través de la marcha atrás.Con el objeto de estudiar el comportamiento de este predicado incorporado vamos a definirlo como sí no lo estuviera:

repeat

repeat if repeat

El objetivo será éxito por la primera cláusula repeat.

7.6.5 PREDICADOS DE TIPIFICACION

En los programas hay ocasiones en las que la lógica del mismo puede presentar el caso de que según sea la estructura de un término se emplee un procedimiento u otro. En otras palabras, se necesitan unos recursos para determinar el tipo de estructura que puede presentar en un momento dado un término.También puede preguntar si una variable esta libre o definida.

ATOM

El objetivo atom (X) acaba con éxito si el término X en ese momento representa un atomo Prolog.

INTEGER

El objetivo integer(X) es éxito si en ese momento X representa un entero Prolog.

ATOMIC

El objetivo atomic(X) acaba con éxito si en ese momento X representa tanto un entero como un átomo Prolog. En el caso de que alguna versión no tuviera este predicado, se puede definir como:

atomic(X) if atom (X)

atomic(X) if integer(X)

VAR o FREE

El predicado var(X) o free(X) tiene éxito si la variable X en este momento esta libre (no particularizada).

NONVAR o BOUND

El objetivo nonvar(X) o bound(X) acaba con éxito si la variable X en ese momento es una variable definida o particularizada. Es el predicado opuesto a los var o free.

7.6.6 Impresion y lectura de terminos

Hasta el momento presente la introducción y la impresión de datos en el programa se hacía respectivamente por medio de las cuestiones y modificaciones en la base de datos y las respuestas que ofrecía el sistema. Desde ahora

llamaremos entrada o lectura a la operación de introducir datos al sistema y salida, grabación o impresión a la operación de obtener la información ofrecida por el sistema.

GRABACION O IMPRESION DE TERMINOS

WRITE

El objetivo write(X) imprime (graba) el término X en el terminal para lo cual X debe estar particularizado. Una característica de los predicados de entrada/salida es que en la marcha atrás no son resatisfechos.

DISPLAY

El predicado display(X) escribe el término X en la unidad corriente de salida. Se diferencia del predicado write en que ignora cualquier operador. Además el formato de salida depende de cada versión en particular.

NL

El predicado incorporado nl se emplea para forzar que la salida impresa sea en una nueva línea, de hecho "nl" es la abreviatura de "new line" (linea nueva). Sólo se satisface una vez con éxito.

TAB

El predicado incorporado tab (X) se emplea para imprimir una serie de caracteres en blanco, tantos como marca X. Por tanto X debe estar particularizado a un entero Prolog.Tiene éxito una sola vez y si se intenta resatisfacer falla.

LECTURA DE TERMINOS

READ

El predicado read(X) leerá el término que se teclee inmediatamente en la terminal. Este predicado exige que despues del término se teclee un punto(que no formará parte de la información leida) y después se pulse la tecla de fin de mensaje. En el caso de que X sea libre el objetivo read (X) causa que X sea particularizado con el término recien tecleado. En el caso de que X este particularizado, el objetivo read sólo podrá tener éxito una vez.

7.6.7 Impresion y lectura de caracteres

Los caracteres tambien pueden imprimirse y leerse en Prolog por medio de los predicados incorporados adecuados. Los argumentos de éstos siempre hacen referencia a la representación de los caracteres en código ASCII, que es un número entero.

IMPRESION O GRABACION DE CARACTERES

PUT

El objetivo put(X) imprimirá el caracter representado por X, que deberá estar particularizado con un entero que represente un caracter ASCII. El predicado put siempre será éxito y fallará cuando se intente resatisfacer.

LECTURA DE CARACTERES

Hay dos predicados incorporados para leer los caracteres, son muy parecidos. Sabemos que hay dos clases de caracteres: los imprimibles y los no imprimibles.Los primeros son los que causan alguna señal en la impresora o en la pantalla de su terminal. Los no imprimibles no causan ninguna marca o señal, pero realizan otras acciones como saltar a una nueva linea, etc.

GET 0

El objetivo get0(X) será siempre éxito y particularizará X con el primer carácter que se teclee, tanto si es un carácter imprimible como si no.

GET

El objetivo get(X) siempre es éxito y particularizará X con el primer carácter imprimible que se teclee, ignorando todos los caracteres no imprimibles anteriores.En algunas versiones, si X está particularizado con un caracter, el

predicado get comparará el caracter imprimible tecleado con el caracter X, dependiendo el éxito o fallo del objetivo get(X) del resultado de la comparación.

SKIP

El predicado skip(X) ignora todos los caracteres de la corriente de entradas hasta encontrar un carácter que pueda equipararse con X. El objetivo skip (X) solo tiene éxito una sola vez.

7.6.8 Operaciones de E/S con los archivos

El Prolog tiene interiormente definidos unos "caminos" por donde circula la información.Unos de estos caminos son los de entrada y de salida. Algunos predicados hacen que estos caminos tengan un destino u otros.El camino de salida por lo general esta dirigido hacia la pantalla del usuario , por medio de un predicado desviamos esta corriente hacia otro dispositivo para grabar la información en un archivo.

En cuanto a los archivos o ficheros diremos que son un conjunto coherente de datos almacenados en un soporte magnético, generalmente en un disco. Los datos se almacenan agrupados en registros (conjunto de datos con la misma estructura). Al principio y al final del archivo hay una serie de marcas para informar al ordenador que es el principio y final del archivo respectivamente. El sistema(el Prolog) al detectar estas marcas obra en consecuencia.

Los predicados de grabación y lectura son los mismos que los estudiados hasta el momento.La diferencia está encaminar la información hacia un dispositivo o hacia otro.

Las operaciones de entrada/salida en un archivo,pueden considerarse divididas en tres bloques:

* abrir el archivo.
* grabar o leer en el archivo.
* cerrar el archivo.

GRABACION EN LOS ARCHIVOS

TELL

El objetivo tell(X) modifica el camino de salida hacia un archivo cuyo nombre es X. El objetivo tell(X) solo puede satisfacerse una vez, cuando la marcha atrás no cambie el camino de salida al estado primitivo. La primera vez que aparece el predicado tell(X) para grabar en el archivo X, el Prolog supone que es un archivo nuevo y puede ocurrir:

- a) Si X es el nombre de un archivo inexistente, Prolog crea un nuevo archivo con ese nombre.

- b) Si X es el nombre de un archivo ya existente, Prolog destruye la información que hubiera en el y empieza a grabar desde el principio.

En cualquiera de los dos casos, Prolog considera el archivo X abierto y pueden grabarse los datos en el por medio de los predicados write y put.

TELLING

El objetivo telling(X) tiene éxito si X coincide con el nombre de la corriente de salida, de lo contrario falla.

TOLD

El objetivo told(X) hace que el archivo cuyo nombre es X se cierre y la corriente de salida vuelva al archivo de usuario.Si se intenta alguna operación de escritura en el archivo X despues del objetivo told (X) causará un error.

LECTURA DE ARCHIVOS

SEE

El objetivo see(X) cambia el camino de entrada hacia el archivoX y abre éste, para su lectura con los predicados read y get. Permanecerá abierto hasta que se encuentre el objetivo seen(X).

SEEING

El objetivo seeing(X) tiene éxito si X coincide con el nombre de la corriente de entrada, de lo contrario falla.

EOF

El objetivo (end of file) eof(X) es éxito si se detecta el final del archivo X,es decir,no hay más datos para leer.

SEEN

El objetivo seen (X) cierra el archivo cuyo nombre es X, es decir, devuelve la corriente de entrada al teclado. Si se pretende leer este archivo despues del objetivo seen(X) causará un error.

CONSULTA DE ARCHIVOS

Hay muchas veces que los archivos sólo son un reflejo de la base de datos. En tal caso podemos beneficiarnos de los predicados de manejo de la base de datos, tales como consult y save. Se entiende que estos archivos son de una extensión reducida, pues el predicado consult transfiere los datos desde el almacenamiento auxiliar del archivo a la memoria de trabajo, mientras que en una operación de lectura con read sólo transfiere a la memoria de trabajo un registro.

7.6.9 Predicados para utilizar la pantalla

* CURSOR

* MAKEWINDOW

CURSOR

El objetivo cursor siempre tiene éxito y coloca el cursor en la posición marcada por los valores de Fila y Columna particularizados en ese momento.

MAKEWINDOW

Este objetivo define una ventana, donde X es un conjunto de variables particularizadas que especifican el nombre de la ventana, la posición dentro de la pantalla, el color, etc.

7.6.10 Predicados que manipulan clausulas

CLAUSE

Vamos a ver un predicado que examina las claúsulas que hay en la base de datos, que es muy importante cuando se trabaja con programas que tienen como datos otros programas escritos en Prolog o en otros lenguajes.

El objetivo clause(X,Y) provoca que X e Y sean equiparados con la cabeza y el cuerpo respectivamente de una cláusula que se encuentre en la base de datos.Debe, pues,

estar X particularizado al nombre de una cláusula, que de no encontrarlo en la base de datos el objetivo fallará. En el caso de haber varias cláusulas con el mismo predicado se equipara con la primera. El predicado clause solo tiene un argumento para equiparar el cuerpo de la regla, encargándose el Prolog de que coincidan el número de argumentos de ambos, tanto si en la cláusula no hay argumentos como si hay más de uno.

8. INTRO. A LOS SISTEMAS EXPERTOS

8.1. COMPONENTES DE UN SISTEMA EXPERTO

Actualmente no existe algo que pueda ser llamado sistema experto "estándar". Debido a la variedad de técnicas que se han seguido para crear sistemas expertos, las diferencias entre ellos son tan varias como las que existen entre los programadores que los diseñan y los problemas que pretenden resolver. Sin embargo, hay unos componentes básicos que existen en la mayoría de los sistemas expertos y que son:

- Una base de conocimiento.
- Un motor de inferencia.
- Un interfaz de usuario.

8.2. BASE DE CONOCIMIENTO

En la primera etapa de la inteligencia artificial muchos científicos creyeron que los ordenadores, al emular el proceso de razonamiento humano, podían resolver los problemas sin necesidad de tener acceso a grandes cantidades de conocimiento específico. Aunque los primeros intentos de resolver los problemas por puro razonamiento parecían prometedores, al final no tuvieron éxito.

El planteamiento actual para desarrollar sistemas expertos es el opuesto al que se tenía inicialmente. Se considera vital el que el sistema experto tenga acceso a un conocimiento del dominio tan amplio como sea posible, ya que debe ser capaz de dar consejos inteligentes sobre este dominio particular. El componente del sistema experto que contiene el conocimiento del sistema recibe el nombre de ***base de conocimiento***, y es tan consustancial a la forma en que se construyen los sistemas expertos, que se les conoce como ***sistemas basados en conocimiento***.

Una base de conocimiento contiene tanto conocimiento declarativo como conocimiento procedimental. Los dos tipos de conocimiento se pueden presentar aislados o de forma conjunta.

El conocimiento procedimental en un sistema basado en reglas está en la forma de reglas de producción heurística "si...entonces" y está completamente integrado con el conocimiento declarativo.

8.3. MOTOR DE INFERENCIA

Un experto no es aquel que únicamente tiene acceso a una gran cantidad de conocimiento, sino el que sabe cuándo y cómo debe aplicar el conocimiento apropiado. En la misma forma, el tener una base de conocimiento no hace inteligente a un sistema experto, sino que debe tener otro componente que dirija y controle la implementación del conocimiento. A ese elemento se le conoce con diferentes nombres, como pueden ser la estructura de control, el intérprete de reglas o el motor de inferencia.

El motor de inferencia decide qué técnicas de búsquedas heurística se usarán, para determinar cómo se deben aplicar al problema las reglas que se encuentran en la base de conocimiento, ejecutando las reglas y determinando si se ha encontrado una solución aceptable.

Como el conocimiento no está "entremezclado"con la estructura de control, un motor de inferencia que trabaje bien en un sistema experto puede trabajar igualmente bien con una base de conocimiento diferente, reduciendo en esta forma el tiempo de desarrollo de un sistema experto. Un ejemplo de lo anterior lo puede constituir el motor de inferencia de MYCIN, del que se puede disponer por separado con el nombre de EMYCIN (MYCIN Esencial). Se puede crear un nuevo sistema experto con una nueva base de conocimiento y con EMYCIN, eliminando la necesidad de desarrollar un nuevo motor de inferencia.

8.4. INTERFAZ DEL USUARIO

Un sistema experto,por muy sofisticado que sea, será perfectamente inútil si el usuario no puede comunicarse con él. La parte del sistema experto que permite esta comunicación es conocida como el interfaz del usuario.

La comunicación que realiza un interfaz de usuario es bidireccional. Al nivel más sencillo, el usuario debe ser capaz de describir su problema al sistema experto y éste debe ser capaz de responder con sus recomendaciones. Se pretende en la práctica que un interfaz de usuario realice otras funciones adicionales. Unas veces se pide al sistema que explique su "razonamiento" y otras el sistema pide información complementaria del problema.

Aunque los diseñadores de sistemas expertos normalmente tienen bastante experiencia con los ordenadores, frecuentemente los posibles usuarios suelen ser principiantes en su uso del ordenador.Por tanto,es importante asegurar que un sistema experto sea especialmente fácil de usar.

La mayor parte de los interfaces hacen amplio uso de técnicas desarrolladas en otra disciplina de IA como es el procesamiento del lenguaje natural. Estas técnicas permiten la comunicación con un sistema experto en lenguaje ordinario y permiten que el sistema experto responda en el mismo lenguaje. Este tipo de interfaz de usuario se conoce como preproceso de entrada de lenguaje natural.

8.5. CARACTERISTICAS DE UN SISTEMA EXPERTO

¿Cuales son las características de un buen sistema experto? Aunque cada sistema experto tiene sus propias características particulares, hay varias que son comunes a muchos sistemas. La siguiente lista, del libro RULE-BASED EXPERT SYSTEMS, sugiere algunos de los criterios que los autores piensan que son prerrequisitos importantes para que los sistemas expertos sean aceptados por sus posibles usuarios.

- "El programa debe ser útil." Se debe desarrollar un sistema experto que resuelva una carencia específica, para la que se sabe que se necesita ayuda.

- "El programa debe poder usarse. "Se debe diseñar unsistema que pueda ser usado por cualquier persona que no tenga excesiva experiencia con los ordenadores.

- "El programa debe ser educacional cuando se necesite. "El sistema podrá ser usado por los no expertos, que de esta forma podrán incrementar su propia experiencia.

- "El programa debe ser capaz de explicar sus propios consejos. "El sistema deberá explicar su propio sistema de "razonamiento" que le ha llevado a sus conclusiones, para que así podamos decidir si aceptamos las recomendaciones del sistema.

- "El programa debe ser capaz de responder a preguntas sencillas. "Como diferentes personas con diferentes niveles de conocimiento pueden usar el sistema, éste deberá responder preguntas sobre puntos que no están claros para todos los usuarios.

- "El programa debe ser capaz de incorporar nuevo conocimiento. "El sistema experto no sólo será capaz de responder a las preguntas del usuario, sino que también debe ser capaz de hacer preguntas para adquirir información adicional.

- "El conocimiento del programa pueda ser modificado fácilmente." Es importante que podamos revisar fácilmente la base de conocimiento de un sistema experto para corregir errores o añadir nueva información.

8.6. DESARROLLO DE UN SISTEMA EXPERTO

Aunque se ha avanzado notablemente en el proceso de desarrollo de un sistema experto, todavía es un proceso que lleva bastante tiempo de realización. Desarrollar un pequeño sistema experto por una o dos personas puede llevar unos cuantos meses; sin embargo, la implementación de un sistema experto complejo puede ocupar a un equipo de varias personas durante más de un año.

En el desarrollo de sistemas expertos intervienen dos tipos de personas: los ***ingenieros del conocimiento*** y los ***expertos del dominio***.

Un ingeniero de conocimiento es un especialista de IA (bien sea un científico de ordenadores o un programador), que posee el "arte" de desarrollar sistemas expertos.

Hasta ahora no hay criterios que sean universalmente aceptados para determinar exactamente quién es un ingeniero del conocimiento. No se necesita un título en "ingeniería de conocimiento" para llamarse ingeniero de conocimiento; de hecho, casi cualquier persona que haya contribuido a la parte técnica del proceso de desarrollo de un sistema experto puede ser considerado un ingeniero de conocimiento.

Un experto en el dominio es un individuo que tiene una experiencia significativa en el dominio del sistema experto que está siendo desarrollado. No es estrictamente necesario que el experto en el dominio entienda de IA o de sistemas expertos; eso será función del ingeniero del conocimiento.

El ingeniero del conocimiento y el experto en el dominio trabajarán juntos durante largos periodos de tiempo, a través de las distintas etapas del proceso de desarrollo. Normalmente un sistema experto se desarrolla y se depura durante un periodo de varios años.

8.7. IDENTIFICACION

Antes de comenzar con el desarrollo de un sistema experto, es importante que se describa el problema que se requiere resolver con el mayor detalle posible. No es suficiente pensar que un sistema experto puede ser útil en una situación dada, sino que se debe determinar la naturaleza exacta del

problema y establecer los objetivos precisos que indiquen exactamente cómo se espera que el sistema experto contribuya a la solución.

En principio, el ingeniero del conocimiento, que posiblemente no está familiarizado con el dominio en particular, debe consultar manuales y libros para tomar contacto con el tema en cuestión. A continuación, el experto en el dominio describe varias situaciones típicas del problema. El ingeniero del conocimiento intenta extraer los conceptos fundamentales de los casos presentados para desarrollar una idea más general del propósito del sistema experto.

Despues de que el experto en el dominio describe distintos casos, el ingeniero del conocimiento desarrolla una "primera" descripción del problema. Normalmente el experto en el dominio opina que esta descripción no representa el problema en su totalidad, por lo que sugiere cambios en esta descripción, dando unos ejemplos adicionales que ilustran los puntos más significativos en el problema. El ingeniero del conocimiento reforma la descripción, y el experto en el dominio sugiere nuevos cambios. Este proceso se repite hasta que el experto en el dominio está seguro de que el ingeniero del conocimiento ha comprendido el problema, y hasta que ambos están de acuerdo en que la descripción de éste es la adecuada.

Este procedimiento"iterativo" es típico de un proceso de desarrollo de un sistema experto. Los resultados se evalúan en cada etapa del proceso y se comparan con las especificaciones; si no concuerdan con ellas, se hacen ajustes y se evalúan los nuevos resultados. El proceso continúa hasta que se alcanzan resultados satisfactorios.

En esta etapa preliminar no sólo se identifica el problema, sino que también es importante identificar los recursos.¿Quién debe participar en el proceso de desarrollo? ¿Existe un único experto en la materia o la experiencia está distribuida entre varias personas? ¿Es suficiente un ingeniero de conocimiento para desarrollar el sistema, o es necesario proporcionar una asistencia técnica adicional?

Las personas que intervienen en el desarrollo no son los únicos recursos que deben ser identificados, ya que el conocimiento del dominio no sólo está en los expertos humanos, sino que también hay fuentes de información en libros de referencia y manuales de procedimiento.

8.8. CONCEPTUALIZACION

Una vez que se ha identificado el problema que el sistema experto debe resolver, la siguiente etapa consiste en analizar el problema más profundamente, para que realmente se entiendan tanto sus puntos específicos como sus generalidades.

En la etapa de conceptualización, el ingeniero del conocimiento crea frecuentemente un diagrama del problema, en el que representa gráficamente las relaciones existentes entre los objetos y los procesos. A menudo es útil dividir el problema en una serie de subproblemas y hacer un diagrama entre las relaciones existentes dentro de cada subproblema y las relaciones de los subproblemas entre si.

Como sucede en la etapa de identificación, la conceptualización usa un proceso circular iterativo entre el ingeniero del conocimiento y el experto en el dominio. Cuando ambos están de acuerdo en que los conceptos claves y las relaciones existentes entre ellos están adecuadamente conceptualizadas, se podrá decir que esta etapa está completa.

En el proceso de desarrollo de un sistema experto no sólo debe ser iterativa cada etapa, sino que también deben serlo las relaciones entre etapas. Como cada etapa es más detallada que la etapa anterior, cualquiera de ellas puede descubrir un fallo en la etapa precedente.

Por ejemplo, durante la etapa de conceptualización podemos detectar alguna incongruencia en la descripción hecha en la etapa de identificación. Esto puede ser consecuencia de la omisión de un elemento clave en la descripción o quizás a que se ha establecido un objetivo de manera incorrecta. En este caso se necesita una vuelta a la etapa de identificación para incrementar la precisión de la descripción. Un proceso similar puede ocurrir en cualquier etapa del desarrollo.

8.9. FORMALIZACION

En las etapas precedentes no se ha hecho ninguna tentativa de relacionar el problema del dominio con la tecnología IA que pueda resolverlo. En ellas nos hemos

centrado completamente en el entendimiento del problema. En la etapa de formalización el problema se conecta con la solución propuesta, analizando las relaciones descritas en la etapa de conceptualización. El ingeniero del conocimiento comienza a seleccionar las técnicas de desarrollo que son apropiadas para este sistema experto particular.

Durante la formalización es importante que el ingeniero del conocimiento esté familiarizado con:

- Las distintas técnicas de representación del conocimiento y la búsqueda heurística usadas en sistemas expertos
- Las "herramientas" del sistema experto que puedan facilitar el proceso de desarrollo.
- Otros sistemas expertos que puedan resolver problemas similares y que por tanto puedan ser adaptados al problema en cuestión.

A menudo sería deseable seleccionar una herramienta o técnica de desarrollo única que pudiera ser usada a través de todas las etapas del sistema experto. Sin embargo, el ingeniero del conocimiento comprueba a veces que no hay una técnica particular apropiada para el sistema completo, teniendo que utilizar diferentes técnicas para los diferentes subproblemas. Una vez determinadas las técnicas que serán usadas, el ingeniero del conocimiento comienza a desarrollar la especificación formal que puede ser usada para desarrollar un sistema experto prototipo.

En el caso de un sistema basado en reglas, el ingeniero del conocimiento desarrolla un conjunto de reglas diseñadas para representar el conocimiento del experto en el dominio. Esta es una parte muy crítica del desarrollo y requiere gran destreza por parte del ingeniero del conocimiento. Muchos expertos en el dominio pueden explicar lo que hacen, pero no porqué lo hacen; por tanto, una de las responsabilidades del ingeniero del conocimiento es analizar situaciones tipo y a partir de ellas extraer las reglas que describan el conocimiento del experto en el dominio.

El proceso de formalización es a menudo la etapa más iteractiva del desarrollo del sistema experto y la que más tiempo consume. El ingeniero del conocimiento debe desarrollar un conjunto de reglas y consultar con el experto en el dominio si representan su conocimiento adecuadamente. Este revisa las reglas y sugiere cambios que son incorporados a la base de conocimiento por el ingeniero del conocimiento.

Como en otras etapas del desarrollo, este proceso también es iteractivo: la revisión de reglas se repite y éstas se van mejorando hasta que los resultados sean satisfactorios. No es estraño que el proceso de formalización de un sistema experto complejo se prolongue durante varios años.

8.10. AUTOMAT. DE LA CREACION DE LA BASE DE CONOCIMIENTO

Debido al considerable gasto de tiempo que supone la transferencia del conocimiento del experto a la base de conocimiento de un sistema experto, los investigadores de IA

están experimentando varios métodos de reducir este proceso. La mayor parte de la investigación se realiza en las siguientes líneas:

- Herramientas más sencillas. Existe en la actualidad una gran variedad de herramientas de desarrollo de sistemas expertos,que son programas especiales diseñados para simplificar el proceso de construcción de una base de conocimiento.A pesar de ello,el uso de estas herramientas todavía es relativamente complejo.

Sin embargo, aunque es mucho más fácil construir un sistema experto usando las herramientas apropiadas, se requiere una gran experiencia en IA para ser capaz de usar las herramientas de una manera efectiva.

Actualmente el uso de las herramientas de desarrollo de sistemas expertos requiere la experiencia de un ingeniero de conocimiento. Se sigue una linea de investigación dedicada a simplificar el uso de las herramientas de desarrollo, de tal manera que un experto en el dominio sin experiencia anterior en ordenadores sea capaz de construir una base de conocimiento sin necesidad de un ingeniero de conocimiento.

- Adquisición automática del conocimiento. Aunque los conocimientos contenidos en un libro no pueden reemplazar al conocimiento del experto en el dominio, sin enbargo pueden proporcionar un suplemento valioso a su experiencia o constituir la base sobre la que se puede construir una base de conocimiento.

Los científicos de IA están investigando las herramientas de desarrollo de sistemas expertos que puedan leer libros sin intervención humana y que automáticamente incorporen el conocimiento de esos libros a la base de conocimiento.

La tecnología actual de visión por ordenador permite a un ordenador "leer" material impreso, en el sentido de que puede reconocer letras y palabras. A pesar de ello, la tecnología de lenguaje natural no ha progresado suficientemente como para permitir a un ordenador comprender todo lo que lee y ni si quiera es capaz de extractar la información esencial. Aunque todavía hay mucho trabajo por hacer, se han realizado experimentos muy prometedores en la universidad de Yale por el grupo que dirige Roger Schank, en el que los sistemas de lenguaje natural desarrollados muestran un pequeño grado de entendimiento en dominios muy limitados.

- Descubrimiento de nuevos conceptos. A pesar del concepto erróneo ampliamente extendido de que un ordenador no puede nunca saber nada para lo que no haya sido programado, la investigación en inteligencia artificial ha demostrado claramente que los programas pueden "aprender". Esto se demostró en la primera etapa de la IA, cuando los programas de juego, como el Jugador de Damas de Samuel, mostraron lo bien que habían aprendido al convertirse en mejores jugadores que sus creadores.

La mayor parte de los sistemas expertos llegan a sus conclusiones después de un proceso de razonamiento desde conceptos conocidos, pero se están investigando métodos para que puedan aprender nuevos conceptos. El programa AM de Doug Lenat descubre nuevos conceptos, los evalúa y los añade a su base de conocimiento EURISKO, que es la ampliación de AM, descubre nuevas heurísticas en dominios especificos.

8.11. IMPLEMENTACION

Durante la etapa de implementación se introduce, en el ordenador que se ha elegido para el desarrollo del sistema, el programa de los conceptos formalizados, usando las herramientas y técnicas predeterminadas para implementar un primer prototipo del sistema experto.

En teoría, si las etapas anteriores se han seguido con gran cuidado, la implementación del prototipo será una tarea relativamente fácil. En la práctica, el desarrollo de un sistema externo es tanto un arte como una ciencia, ya que el seguimiento de todas las reglas no garantiza que el sistema funcione cuando se implementa por vez primera y generalmente suele suceder lo contrario. Muchos científicos consideran el primer prototipo como un sistema para tirar, que sólo es útil para evaluar los progresos que se van haciendo, pero que no servirá como sistema experto.

Si el prototipo funciona, el ingeniero del conocimiento podrá decir si las técnicas elegidas para el desarrollo del sistema experto son las adecuadas. A veces puede descubrir que las técnicas elegidas no pueden ser implementadas. Por ejemplo, puede suceder que las técnicas seleccionadas de representación del conocimiento no puedan integrarse para los diferentes subproblemas. En este caso, los conceptos deben ser reformalizados o quizá sea necesario crear nuevas herramientas de desarrollo para implementar el sistema eficientemente.

Una vez que el sistema prototipo se ha perfeccionado suficientemente para ser ejecutado, el sistema experto está listo para ser probado, y así asegurar que se ejecuta correctamente.

8.12. PRUEBA

La posibilidad de que un sistema experto se comporte desde el primer momento sin fallos es muy remota. El ingeniero del conocimiento no espera que en la etapa de prueba se verifique que el sistema está construido correctamente en su totalidade, esta etapa sirve para identificar los puntos débiles de la estructura y de la implementación del sistema para hacer las oportunas correcciones.

Dependiendo de los tipos de problemas que se encuentran, el procedimiento de prueba servirá para indicar que el sistema está complementado incorrectamente o que las reglas sí eran adecuadas pero su formalización fue incompleta. Los resultados de las pruebas se usan como "realimentación" a la etapa anterior, y de esta forma se va ajustando el comportamiento del sistema.

Cuando el sistema resuelve correctamente los problemas sencillos, el experto en el dominio propone problemas complejos que requieren una gran cantidad de

experiencia humana.Este tipo de pruebas descubren en el sistema fallos más serios y proporcionan la oportunidad de depurarlo aún más.

El sistema experto se considera que está listo cuando opera al nivel de un experto humano.El proceso de prueba no está completo hasta que las soluciones propuestas por el sistema experto sean tan validas como las propuestas por el experto humano.

8.13. COMENTARIO SOBRE EL PROGRAMA EJEMPLO

El objetivo del autor no ha sido crear un sistema experto práctico (o sea, una aplicación real) sino un programa prototipo que utiliza el método de programación declarativa y no la programación clásica procedimental.

No obstante, este programa está diseñado de tal modo que aceptará cualquier entrada de datos (de un modo coherente, ¡claro !) e implementará una base de conocimiento real con los hechos y reglas que se le vayan introduciendo. Este programa utiliza el encadenamiento hacia atrás y encuentra múltiples soluciones.

/* Este programa ha sido realizado utilizando la sintaxis de la versión 1.0 del compilador de Prolog de la casa Borland «Turbo Prolog 1.0». Este ejemplo es un sistema experto con encadenamiento hacia atrás que encuentra múltiples soluciones */

```
domains

  lista=symbol*

  database

  info(symbol,lista)

   /* objeto, lista de atributos */

  si(symbol)      /* recordar las */

  no(symbol)      /* respuestas si y no */


predicates

  juntar(lista,lista,lista)

  escribirlista(lista,integer)

  introducir

  purgar

  añadir(symbol,lista,lista)

  preguntar(lista)

  atributos(symbol,lista)

  procesar(symbol, symbol, symbol)

  start

  anterioressi(lista)

  anterioresno(lista)

  xanterioressi(symbol,lista,lista)

  xanterioresno(symbol,lista,lista)

  intentar(symbol,lista)
```

```
  miembro(symbol,lista)

  xwrite(symbol)

goal

  start.

clauses

  start if

  consult("x.dat") and  /* leer un info */

  fail.  /* ir a la siguiente cláusula */

  start if

  assert(si(end)) and

  assert(no(end)) and

  write("¿Introducirá información? (s/n): ") and

  readln(A) and

  A=s and

  not(introducir) and

  save("x.dat") and ! and

  preguntar([]).

  start if

  preguntar([]) and purgar.

  start if

  purgar.


  /* introducir objetos y atributos */

  introducir if

  write("¿que es?: ") and

  readln(Object) and

  Object"fin" and

  atributos(Object,[]) and

  write("¿más? ") and ! and
```

```
readln(Q) and Q=s and
introducir.
/* obtener los atributos */
atributos(O,List) if
write(O," es/tiene/hace: ") and
readln(attribute) and
Attribute"fin" and
añadir(Attribute,List,List2) and
atributos(O,List2).
atributos(O,List) if
assert(info(O,List)) and
escribirlista(List,1) and ! and
nl.

/* añadir un symbol a una lista */
añadir(X,L,[X|L]).

/* pedir información al usuario para
 encontrar un objeto  */
preguntar(L) if
info(O,A) and
not(miembro(O,L)) and    /* no encontrar el mismo*/
añadir(O,L,L2) and       /* dos veces */
anterioressi(A) and      /* comprobar las respuestas si anteriores */
anterioresno(A) and      /* respuesta no */
intentar (O,A) and
! and  write(O,"  satisface los") and nl and
```

```
write("atributos actuales") and nl and
write("¿continuar? ") and
readln(Q) and  Q=s and
preguntar(L2).

/* seleccionar todos los objetos que no tienen
los atributos apropiados como ya están
determinados  */
anterioressi(A) if
si(T) and ! and
xanterioressi(T,A,[]) and !.

xanterioressi(end,_,_) if !.
xanterioressi(T,A,L) if
miembro(T,A) and ! and
añadir(T,L,L2) and
si(X) and not(miembro(X,L2)) and ! and
xanterioressi(X,A,L2).

/* seleccionar todos los objetos que tienen
atributos ya determinados como no
pertenecientes al objeto */
anterioresno(A) if
no(T) and ! and
xanterioresno(T,A,[]) and !.

xanterioresno(end,_,_) if !.
xanterioresno(T,A,L) if
not(miembro(T,A)) and ! and
```

```
añadir(T,L,L2) and
no(X) and not(miembro(X,L2)) and ! and
xanterioresno(X,A,L2).

/* intentar una hipótesis */
intentar(_,[]).
intentar(O,[X|T]) if
si(X) and ! and
intentar(O,T).
intentar(O,[X|T]) if
write("es/tiene/hace ",X," ") and
readln(Q) and
procesar(O,X,Q) and ! and
intentar(O,T).

/* procesar varias respuestas */
procesar (_,X,s) if
asserta(no(X)) and ! and fail.
procesar(O,X,why) if
write("Creo que puede ser") and  nl and
write(O,"porque tiene: ") and nl and
si(Z) and xwrite(Z) and nl and
Z=end and ! and
write("es/tiene/hace ",X,"? ") and
readln(Q) and
procesar(O,X,Q) and !.

xwrite(end).
xwrite(Z) if
```

```
write(Z).

purgar if
retract(si(X)) and
X=end and fail.
purgar if
retract(no(X)) and
X=end.

juntar([],List,List).
juntar([X|L1], List2, [X|L3]) if
juntar (L1,List2,L3).

escribirlista([],_).
escribirlista([Head|Tail],3) if
write(Head) and nl and escribirlista(Tail,1).
escribirlista([Head|Tail],I) if
N=I+1 and
write(Head," ") and escribirlista(Tail,N).

miembro(N,[N|_]).
miembro(N,[_|T]) if  miembro(N,T).
```

BIBLIOGRAFIA

* **IA, Introducción y situación en España**
Edición de Ricardo Valle, José Barberá y Francisco Ros.
Colección Informes FUNDESCO 1984.

* **IA, Conceptos, técnicas y aplicaciones**
Serie MUNDO ELECTRONICO
MARCOMBO. BOIXAREU EDITORES, 1987

* **A fondo: Inteligencia Artificial**
Mishkoff
ANAYA Multimedia, 1988

* **Programming for artificial intelligence**
BRATKO, Ivan
Addison-Wesley, Wokingham, 1986

* **Lógica, programación e inteligencia artificial**
KOWALSKY, Robert
Díaz de Santos, Madrid 1986

* **Lógica Informática**
CUENA, José
Alianza Informática, Madrid 1986

* **The Handbook of Artificial Intelligence**
Barr, Avron y Edward A. Feigenbaum
vol . 3, Stanford, California, Heuris Tech Press, 1982

* **Programming in Prolog**
CLOCKSIN, W.F. y MELLISH, C.S.
Springer-Verlag, Berlín Heidelberg 1984

* **PROLOG, lenguaje de la inteligencia artificial**
COLMERAUER, Alan
Mundo Científico, Noviembre 1984

* **The art of Prolog. Advanced programming Techniques**
STERLING, Leon y Shapiro, Ehud
MIT Press, Cambridge, Massachusetts, 1986

* **User's Guide Turboprolog**
Borland International, Inc.

* **Aplique TURBOPROLOG**
ROBINSON, Philip R.
Osborne-Mc Graw-Hill, 1987

* **TURBO PROLOG, programación avanzada**
SCHILDT, Herbert
Osborne-Mc Graw-Hill