Principios y métodos de análisis lógico Programación Lógica: PROLOG (Módulo 6)

Rodolfo J. Rodríguez Rodríguez
Correo-E: rodolfojrr@gmx.com
https://about.me/rodolfojrr
https://www.facebook.com/gestioncognitiva.pagina/
San José, Costa Rica

Rodolfo J. Rodríguez Rodríguez
Correo-E: rodolfojrr@gmx.com

CONTENIDOS:

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

1 Introducción

La lógica puede ser descrita como la ciencia o la teoría sobre las inferencias. Así los principios y métodos de la lógica, se orientan a demostrar la corrección o incorrección de las inferencias aplicadas. Para lograr este objetivo se acude a diversas técnicas de demostración. Por medio de demostraciones es posible decidir se se han hecho inferencias correctas o incorrectas y es posible analizar y explicar por qué de esa corrección p incorrección. Así será posible decidir si s han obtenido las conclusiones correctas a partir de la información inicial.

La Lógica busca entender **la inferencia correcta**, de tal manera que sea posible explicitar, mecanizar y formalizar los procedimientos automáticos para que cualquier agente, humano o no, pueda tener garantizada la corrección lógica de su manera de procesar la información. **G. W. F. Leibniz (siglo XVII)** explicitó que lo mejor, en vez de largas discusiones, sería poder simplemente calcular quién tiene la razón.

La **demostración automática (DA)** extiende el ideal de Leibniz de convertir a la lógica en un cálculo mecánico. Desde el siglo XIII, Raymundo Lull construyó aparatos rudimentarios para explorar automáticamente series de combinaciones entre conceptos. Pero fue a partir el siglo XX que aparecieron aparatos capaces de procesar información con suficiente velocidad y exactitud. Los primeros desarrollos lógicos de la DA trataron de usar técnicas lógicas que garantizaran el paso seguro de los **datos iniciales o premisas a las conclusiones**, es decir métodos seguros e infalibles para procesar información. Para algunos objetivos más laxos, es sensato utilizar sistemas menos exigentes: lógicas de probabilidades, factores de confianza, inferencias no-monotónicas, lógica difusa, etc.

2 Perspectiva histórica

A finales del siglo XIII**, Raymundo Lull** creó un abigarrado sistema de círculos de metal o pergamino, pintados de vivos colores. Era una máquina lógica para producir

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

combinaciones de conceptos. Hace aproximadamente doscientos años, **Charles Stanhope** reconstruyó a **los silogismos** con madera y cristal. En 1886, el lógico **Charles Dodgson** (quien escribió *Alicia en el País de las Maravillas* bajo el pseudónimo de **Lewis Carroll**) usó cartón y piezas de colores. En **1903 Annibale Pastore** usó bandas y ruedas para probar **estructuras silogísticas**. En el caso de **Lull**, trató de usar la combinatoria para la defensa de la religión católica**, Stanhope** estudió a la predicación como identidad, **Carroll** generalizó los diagramas de Euler y Venn, mientras que **Pastore** dió una de las mas dramáticas demostraciones de la presencia de estructuras "lógicas" en fenómenos de mecánica simple.

La primera máquina lógica de importancia fue hecha construir en **1869** por **William Stanley Jevons**. El "**piano lógico**" de Jevons fue específicamente creado para ilustrar en el salón de clase **la superioridad de la nueva lógica booleana sobre la anquilosada teoría aristotélica**. Después vendrían las máquinas de **Allan Marquand (1885)** y **Charles P. R. Macaulay (1913).** Fue natural reemplazar poleas y engranes con circuitos eléctricos cuando éstos aparecieron. La **primera máquina lógica eléctrica** fue usada para resolver silogismos por **Benjamin Burack en 1936**, y once años después dos jóvenes estudiantes de licenciatura, **William Burkhart y Theodore A. Kalin** construyeron **la primera máquina lógica eléctrica** para ahorrarse esfuerzo haciendo los ejercicios que les dejaba **su maestro, Quine.** Muchas otras han seguido, como aquella construida en **1951** por **Robert W. Marks** que anunciaba por un alto-parlante cuando el **enunciado era falso**, en una voz profunda: "***Mi nombre es Leibniz Boole De Morgan Quine. No soy sino un robot complicado y algo neurótico, pero me da un gran placer anunciar que su afirmación es totalmente falsa***" ["***My name is Leibniz Boole De Morgan Quine. I am nothing but a complicated and slightly neurotic robot, but it gives me great pleasure to announce that your statement is absolutely false***"].

A mediados de los años cincuenta, el desarrollo de las computadoras permitió a las universidades el acceso a las primeras máquinas y en **1954**, en un congreso en la **Universidad de Cornell**, se reunieron un número considerable de lógicos y computólogos. **Abraham Robinson**, de los **Laboratorios Argonne**, propuso utilizar los "***Universos de***

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

Herbrand" y algunos teoremas de la lógica para atacar el problema de usar computadoras para **demostración automática de teoremas**.

En **1957** se publican los resultados de una máquina que demuestra, llamada "***The Logic Theory Machine"***, realizado por **Newel, Simon y Shaw**, llegando a demostrar 40 teoremas de ***Principia Mathematica***. En **1958 Gilmore**, usando formas normales disyuntivas y los **universos de Herbrand**, logra dar demostraciones automáticas de fórmulas moderadamente complejas de **lógica de primer orden** sin igualdad como:

x y z [{<(Pyz [Qy Rx]) Pxx> <([Pzx Qx] Rz) Pxy>} Pzz]

Esta **fórmula es una verdad lógica**(-trivialmente), es decir, la fórmula es verdadera **independientemente de la interpretación que se haga de los predicados P, Q, y R, respecto a cualquier universo no vacío.**

En **1958, Hao Wang**, en una **IBM**, desarrolla programas que prueban **220 teoremas de *Principia Mathematica*** y en **1960 Martin Davis y Hilary Putnam** proponen un mejor algoritmo para fórmulas instanciadas en forma normal conjuntiva o forma clausular. Al final de estos primeros diez años de **DAT,** en **1965 J. A. Robinson publica su regla de resolución con unificación, lo cual genera un "boom" de la DAT.**

Como conclusión de esta primera década, no sólo se dieron los primeros logros en DAT, sino que surgieron las ideas y métodos básicos de DAT que se desarrollaron en las décadas siguientes, sobre todo en los **Laboratorios Argonne**.

En **1980 Larry Wos** propone el término **"Razonamiento Automatizado"** para reemplazar el más tradicional de **"Demostración Automática de Teoremas"** Con ello quiere enfatizar que las búsquedas de solución no son ciegas y rutinarias, sino que pueden involucrar planeación, ponderación, estrategias, etc.

Curiosamente, por la división entre *hardware* y *software*, hay pocas máquinas lógicas hoy día, si no contamos los **avances en circuitos lógicos integrados** o **las máquinas japonesas especializadas para programación lógica**. Es mucho más fácil encontrar **lógica en software educativo**, como lo demuestra una somera revisión de ***The Computers and Philosophy Newsletter*** **o el** ***Computerized Logic Teaching Bulletin*****.**

La relación **entre DAT y computadoras ha sido historicamente muy estrecha**. Pero hay

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

que precaverse contra la tentación de creer que la demostración mecánica necesita de máquinas inhumanas. Cualquier persona puede calcular mecánicamente, y es **un error reducir la informática a la tecnología de las computadoras. La ciencia de la computación tiene tan poco que ver con las computadoras** como la astronomía con los telescopios. El uso de telescopios no significa que la astronomía se dedique a investigar la construcción y manejo de telescopios. Similarmente, la computadora ayuda a explorar el universo de la computación y el pensamiento mecánico, pero es sólo un instrumento para la ciencia general del procesamiento de la información. Hay que recordar que cuando el **padre de la informática actual, Alan Turing, hablaba de "calculadoras" en los años treinta, se refería a personas. Computar es una actividad tan humana como mecánica.** Al analizar sistemas computacionales, es posible sacar resultados categóricos sobre los alcances y los límites de la inteligencia en particular y sobre el pensamiento humano en general.

En el contexto de los estudios de Lógica y Demostración automática, surge la interrogante ***"¿Qué es la inteligencia?***". Típicamente (aunque no solamente), esta es posible explicarla como un sistemas de competencias para poder encontrar soluciones a nuevos problemas. Los problemas pueden ser intelectuales, emocionales, prácticos. Pueden involucrar lenguaje, movimientos corporales, relaciones y acciones interpersonales. Hay muchos tipos de inteligencia, pero en todos hay una estructura básica: partimos de cierta situación o contexto al que hay que tomar en cuenta y buscamos la solución a un problema o dificultad. La situación es como (aunque no totalmente) una serie de premisas y la solución es como lo que deseamos extraer de ellas, la conclusión o teorema de ese sistema. Pero ***llegar a la meta exige encontrar un camino***, ***un método***, que conlleve de la situación presente a la deseada. La búsqueda de ese camino es lo que se busca automatizar. Se busca **una automatización parcial de la inteligencia**, tanto para **probar que algo es verdadero, como para probar que algo es falso (refutar).** No hay que olvidar que **refutar es demostrar una negación**.

La **ciencia lógica** se ha enriquecido enormemente al tratar de modelar e implementar **sistemas para la mecanización de las inferencias**, especialmente usando

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

computadoras. En el contexto de fundamentación teórica de la computabilidad, es ineludible los usos y aplicaciones de la Lógica teórica. En este sentido es posible hacer la siguiente divisisión:

- uso de estrategias,
- teoría de la complejidad,
- lógicas computacionales y
- solución de problemas abiertos.

Sobre el **uso de estrategias**, se debe considerar que dentro de la ciencia lógica, una parte fundamental es la **teoría de la prueba**. En ella se analizan las propiedades generales de los **sistemas que modelan las demostraciones**. Y la DA es el lazo que une a la **ciencia de la demostración con el arte de las estrategias más eficientes para lograr demostrar algo.** Cuando alguien propone una demostración X, la lógica puede examinarla como arte y como ciencia. Como **ciencia, la pregunta fundamental es si X es verdaderamente una demostración, en qué sentido y qué tan correcta**. Como **arte, la pregunta es si X ejemplifica una estrategia eficiente y general, una estrategia que pueda aplicarse más a menudo y mejor que otras**. Esta pregunta, de orden teórico, tiene importantes consecuencias prácticas y ha llevado a la exploración de estrategias que no se adaptan bien a las **características humanas (limitada memoria, lentitud en el procesamiento)** pero que se adaptan bien a las de las máquinas actuales (limitada capacidad estratégica, falta de sentido común).

Otra **utilidad teórica** es en el campo de la **teoría de la complejidad**. Cuando se busca implementar **sistemas computacionales** que serán usados en el mundo real, no es suficiente saber que cierto problema tiene una solución *en principio*. Es imprescindible saber qué **tantos recursos y cuánto tiempo nos tomará obtener la solución**. Esto ha dado un gran impulso a la **teoría general de la complejidad computacional**, es decir, el estudio del tipo u orden de los recursos que serán necesarios para **resolver una clase de problemas con técnicas específicas.**

También es útil el estudio de la DA para el **desarrollo de lógicas computacionales**. Se necesitan sistemas que tomen en cuenta que el valor de verdad de las proposiciones

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

cambia continuamente y que hay propiedades formales importantes en los procesos, incluyendo la modificación, **creación y destrucción de estados**. Esto ha estimulado la creación de **lógicas "dinámicas" y "lineales"** que toman en cuenta el aspecto no estático que tiene el pensamiento en el mundo real.

Estos estudios han reportado **soluciones de problemas abiertos**. Los sistemas para DA de teoremas no se han limitado a los teoremas conocidos. En matemáticas hay problemas con un alto nivel de complejidad porque antes de llegar a una conclusión hay que considerar todas las variantes posibles. Hay que generar todos los casos y chequearlos uno por uno. Por ejemplo, **el problema topológico de los cuatro colores** requirió el uso de computadoras para examinar exhaustivamente todas las combinaciones relevantes. El problema era averiguar si 4 colores bastan para colorear cualquier mapa sin que ningún par de países vecinos tengan el mismo color. Se puede ver que 3 colores no son suficientes. Por ejemplo, si coloreamos Argentina, Bolivia, Brasil y Paraguay, necesitaremos 4 colores para que ningún par de vecinos tengan el mismo color. Otra manera de formular el problema es preguntarse si puede haber **cinco países compartiendo cada uno al menos algunos metros de frontera con cada uno de los otros**. El teorema de los cuatro colores, demostrado con ayuda de computadoras, dice que tal mapa no existe pues demandaría cinco colores diferentes, y cuatro son suficientes siempre. Hay muchos otros resultados nuevos en áreas como **la geometría algebraica, teoría de redes, álgebra booleana, lógica combinatoria, teoría de grupos, álgebra de Robbins, etc.** Es importante enfatizar que el campo de la DA no sólo ha sistematizado importantes estrategias de prueba útiles para las matemáticas, sino que incluso ha creado nuevos resultados y nuevos problemas.

La utilidad **de la DA no se limita al aspecto teórico**. También tiene utilidad práctica en los campos de:

- verificación y síntesis de programas,
- diseño y verificación de circuitos y
- diseño y manejo de bases de datos.

Verificación y síntesis de programas. **Encontrar una solución es como encontrar una**

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

demostración. La **situación y la meta son como las premisas y la conclusión**. Mientras mejor formulado esté el problema, más claro es el camino hacia su solución. Por ello, **una aplicación natural de los métodos de DA es en el contexto de la generación de programas de computadora**. Se puede expresar con relativa claridad los objetivos del programa, **el comportamiento que se espera, y las condiciones de aplicación**. Es decir, se puede tratar de escribir rigurosamente **la entrada (*input,* situación inicial o premisas)**, **la *corrida* (comportamiento del programa o derivación),** y la **salida (*output,* meta o conclusión).** Nuestro objetivo puede ser producir el programa que se necesite o, si ya se tiene construido, verificar que hace lo que se desea que haga.

Supóngase que se desea un programa o rutina para pasar todos los objetos de A hasta B mediante la repetición de ciclos en cada uno de los cuales sustraemos un objeto de A y se lo agregamos a B hasta agotar A. ¿Cómo se sabe si un programa funciona? Por ejemplo, se pueden **detectar invariancias**. Sabemos que después de cada ciclo, si el método se sigue rigurosamente, el número de objetos en A decrece pero la suma de los objetos en A y B se mantiene constante. **Esta invariancia permite descubrir casos en que el programa falla.**

Otro uso práctico de la DA es en el **diseño y verificación de los llamados circuitos lógicos.** Se sabe desde hace décadas que **las constantes lógicas como la conjunción, la disyunción y la negación tienen correlatos en los circuitos lógicos en serie, paralelo, e invertidores**. Para ser verdadera, **una conjunción necesita que ambos conyuntos sean verdaderos; análogamente, un circuito con dos tramos en serie necesita que pase la electricidad en ambos tramos para que pase en el circuito entero**. Una **disyunción sólo necesita la verdad de uno de los disyuntos, igual que un circuito en paralelo sólo precisa que pase electricidad en alguno de los tramos.** Y la **negación produce una proposición con el valor opuesto**, de manera **análoga a un inversor eléctrico.** El comportamiento de **los circuitos eléctricos** es similar a las **relaciones lógicas** de **las conectivas proposicionales**. Se puede especificar el comportamiento en términos lógicos y **automáticamente generar diseños de circuitos eléctricos** que tengan **las propiedades que deseamos**. O, **si ya se tiene el diseño,**

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

traducirlo al lenguaje lógico y verificar con técnicas lógicas si del diseño se derivan las propiedades que buscábamos. Todo esto sin tener que construir todavía un solo circuito. Y además de **los circuitos lógicos, se pueden verificar otros circuitos**. Por ejemplo, **se ha producido una prueba completamente automatizada de la corrección de un circuito que implementa el estándar internacional (de la IEEE) para cálculos con punto flotante. Se han verificado también circuitos de tamaño comercial para adición y multiplicación y para sistemas de seguridad.**

La DA es útil en el diseño y manejo de bases de datos. Una base de datos útil no es simplemente un montón inerte de informaciones. Si en principio sólo se sabe que en el salón de clases hay 32 alumnas y 28 alumnos, estos datos pueden manipularse para obtener otros datos derivados: Hay **más** alumnas en ese salón, hay cuatro alumnos **menos**, el **total** son 60, etc. Una **base de datos normalmente incluye una maquinaria inferencial que procesa la información y extrae otra información que sólo estaba implícita y dispersa en los datos iniciales**. **Estas bases de datos son llamadas "deductivas",** y la maquinaria inferencial puede utilizar sofisticadas técnicas de DA para extraer el conocimiento que necesitamos. **La lógica es importante en dos aspectos de las bases de datos.** Por un lado, puede **ser una guía para la representación concreta de datos.** Por otro, puede ofrecer **los mecanismos para especificar la información en abstracto que queremos manipular. La representación concreta dentro de la máquina:** La **información que se añade a la base de datos puede estar escrita en una forma similar a la que normalmente se emplea en lógica,** es decir, con una **estructura oracional formada básicamente por nombres y variables de objetos, funciones, predicados, conectivas y cuantificadores.** Para guardar la información de que Juan es hombre y María es mujer, representamos a Juan y a María **como estructuras en la computadora, las enlazamos a sus predicados, y finalmente las conjuntamos mediante una función que produce oraciones complejas.** En la computadora esto puede estar presente **en registros (*registers*), arreglos *(arrays)*, apuntadores *(pointers)*, etc., en una sintaxis que sigue a la de la lógica matemática clásica.**

Pero este primer aspecto tiene **dos cualificaciones**. La **primera cualificación es que**

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

este método de representar la realidad, aunque es el más común y el que normalmente se encuentran en **los lenguajes humanos**, no es el único. Se pueoden **tener lógicas combinatorias** donde no hay objetos y propiedades sino únicamente "**combinadores**". Hay muchos **tipos de lenguajes lógicos** y rara vez se siguen completamente las **estructuras del lenguaje natural**. La **segunda cualificación** es que **la representación en la computadora comúnmente se divorcia de las estructuras lógicas y gramaticales**. En **sistemas conexionistas** los datos están "**dispersos**" en toda una **red de asociaciones**. Son los **lazos entre los nodos**, **su fuerza, propiedades y direcciones, lo que contiene la información.** Igual que parece ocurre dentro **del cerebro humano,** la representación dentro de una computadora puede carecer de signos lingüísticos a la usanza lógica.

El otro aspecto que se mencionó sobre la utilidad de la lógica para el estudio e **implementación de bancos de datos era la representación abstracta** de lo que se quiere manipular. La información puede ser especificada, en **un lenguaje de alto nivel**, **usando lógica clásica**. En vez de usar el español o el inglés para definir lo que se quiere que contenga la base de datos, se puede usar el lenguaje usual de la lógica. Queda entonces claro que el objetivo **al preguntar por algo a la base de datos es averiguar si se sigue lógicamente de los datos que se tienen a través de la maquinaria inferencial disponible.** Las **consultas a bases de datos pueden así verse como peticiones de demostración automática de teoremas (conclusiones del sistema).**

3 Sistemas Basados en Conocimiento

Uno de los desarrollos más exitosos de la Inteligencia Artificial como lo son los Sistemas Basados en Conocimiento. En tanto que se trata de sistemas de representación del conocimiento (experto) y sistemas de razonamiento automatizado (cibernético), es posible acudir al la Teoría General de Sistemas para realizar un enmarque teórico del mismo.

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

Un sistema experto o sistema basado en conocimiento se puede definir como:

Sistema que resuelve problemas utilizando una representación simbólica del conocimiento humano.

Los sistemas expertos has sido particularmente exitosos por sus muchas aplicaciones que van. Desde espectaculares instrumentos de ayuda al pensamiento, como la neurociencia computacional hasta la medicina y la industria automotriz. Como una rama de la cibernética, los sistemas expertos se apoyan en este dictum: razonar es calcular, sostenido por Leibnitz en el siglo XVII.

3.1 Antecedentes

En 1969, el genetista, Joshua Lederberg y los expertos en **Inteligencia Artificial:** E. Feigenbaum y E. Buchanan se asociaron para resolver en forma automática, por medio de computadora, el problema de inferir la estructura molecular de un compuesto químico, a partir de la información facilitada por un espectrofotómetro de masas, problema que hasta entonces requería del trabajo de químicos analíticos expertos. El problema lo resolvieron mediante un programa llamado DENDRAL, inspirada en la simplificación de soluciones posibles realizada por los expertos —en este caso lo que los químicos analíticos sabían de los patrones picos en el espectro. Así surgió el campo de los sistemas expertos, sistemas intensivos en conocimiento cuyo funcionamiento obedece a un gran número de reglas que simplifican en algoritmos la experiencia derivada de conocer con la intuición, agudeza y precisión.

Más tarde, los científicos referidos desarrollaron un programa para encontrar infecciones de la sangre, más complejo que el anterior, ya que en él no había un modelo teórico general del cual deducir las reglas, sino que debían formularse por medio de un diálogo con los expertos reales, quienes, a su vez, las habían adquirido de sus experiencias con casos particulares.

El poder de predicción de los sistemas expertos quedó demostrado cuando, en 1980, **PROSPECTOR** aconsejó horadaciones exploratorias en un sitio descartado por varios geólogos, ya que estaba rodeado de antiguas perforaciones y minas. El programa

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

contenía, en forma de reglas, el conocimiento de nueve expertos, y dio con éxito con un depósito importante de molibdeno, elemento básico para fabricar anticorrosivos. La importancia de concretar el conocimiento y, en cierta forma, la intuición se hicieron evidentes al surgir un novedoso método para efectuar deducciones lógicas, llamado de resolución. Este método fue eficaz en la prueba automática de teoremas y se basa en escribir todas las afirmaciones que definen un problema y las conclusiones intermedias que se deriven, en una fórmula especial llamada "cláusula".

Así, para probar el resultado de una lista de cláusulas, se supone la negación de lo que se desea demostrar y se intenta obtener una contradicción, al añadir a esa negación alguna cláusula de la lista que permita "resolverla" por cancelación de las contradicciones. La deducción se convierte en algo puramente mecánico. Sin embargo, debido a la explosión combinatoria, como sucede en la realidad, en cada paso sólo podían resolverse pocas cláusulas de listas larguísimas de afirmaciones iniciales, por lo que el problema resultaba intratable. Entonces, en **1971**, el investigador **Robert Kowalski** probó que algunas reglas bastante sencillas permiten eliminar la mayoría de las "resoluciones" que no interesan para la prueba en curso, con lo que se reduce en forma notable el número de casos que es preciso examinar. Al mismo tiempo, el francés Alain Colmerauer describió una forma automática, mediante la cual no sólo era posible representar las afirmaciones a través de estructuras arborescentes sino manipularlas por medio de reglas simples, tanto para el análisis como para la generación de nuevas afirmaciones.

Después, junto con **Phillipe Roussel**, **Kowalski** y **Comerauer** escribieron **PROLOG**, un lenguaje de programación basado en técnicas de resolución a manera de estructuras arborescentes. Éste fue el primer programa, hipercibernético, aunque no se impuso de inmediato, pues en ese entonces se prefería usar LISP y FORTRAN.

4 Programación Lógica

La máquina inferencial puede tener reglas especiales de acuerdo a la base de datos a la que se va a aplicar. Los **principios y reglas más generales, sin embargo, son los de la lógica clásica.** Esto no es una limitación ya que los **demostradores automáticos** pueden usar **otras lógicas subyacentes**. Además de **la lógica matemática clásica**, hay **lógicas**

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

probabilísticas, difusas, no-monotónicas, etc., sin contar los **casos de sistemas expertos que usan "grados de confiabilidad" que implementan formas de inferencia difíciles de sistematizar.** Cada sistema distinto ofrece ventajas y desventajas comparado con la lógica clásica. Por ejemplo, **hay demostradores que siguen sistemas de lógicas relevantes**. La **ventaja es que el árbol de búsqueda se poda severamente si nos limitamos a considerar enunciados temáticamente pertinentes. La desventaja es que la lógica relevante es difícil de implementar eficientemente y tiene problemas de decidibilidad.**

La programación lógica consiste en utilizar las reglas y principios lógicos que normalmente se interpretan como afirmaciones (una "semántica declarativa"), e interpretarlos como órdenes o comandos dados a la computadora (una "semántica procedimental").

Es común confundir la programación lógica con los lenguajes en que se ha implementado. El lenguaje de programación en lógica más famoso es **Prolog,** desarrollado a principios de los setentas por **Colmerauer en Marsella**. PROLOG es la abreviatura de PROgramación LOGica, con lo que se hace mención a la procedencia del lenguaje: Es una realización de lógica de predicados, como lenguaje de programación. Aunque **limitado a las cláusulas Horn (básicamente, condicionales simples o anidados), es un lenguaje poderoso y práctico en el que podemos decirle a la computadora lo que se desea, en vez de tener que especificar cómo se desea.** Desgraciadamente, tiene algunas propiedades que dificultan seriamente darle **una semántica puramente declarativa**, como **el supuesto de mundo cerrado (*closed world assumption*, el supuesto de que ya tenemos toda la información pertinente),** el **comportamiento de la negación no como falsedad sino como fracaso del intento de demostración (*negation as failure*), y la orden de abandonar ciertas búsquedas, recortando (*cut*) el árbol de búsqueda.**

En la actualidad, el PROLOG se aplica como lenguaje de desarrollo en aplicaciones de Inteligencia Artificial en diferentes proyectos de Europa. En los Estados Unidos, el **LISP** está más extendido que el PROLOG. Pero para la mayoría de los terminales de trabajo de

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

Inteligencia Artificial se ofrece también el PROLOG. Como una especie de **semiestándar** se han establecido el **DECsystem-10 PROLOG de Edimburgo** y el PROLOG descrito en el libro "Programming in Prolog" de W.F.Clocksin y C.S.Melish. La mayoría de los dialectos PROLOG se basan en este y contienen el DECsystem-10 PROLOG en su ámbito lingüístico. Asimismo y para un estudio más detallado del Prolog, se recomienda la obra de Sterling, L. y E. Shapiro(1986) *The Art of Prolog. Advanced Programming Techniques.*

Al contrario que el LISP (y otros lenguajes), en el PROLOG los programas son confeccionados de forma distinta. L**a lógica se representa en forma de predicados**. Estos predicados aparecen en tres formas distintas: como **hechos, como reglas y como preguntas**. La lógica formulada como hechos y reglas se define como **base de conocimientos**. A esta base de conocimientos se le pueden formular preguntas.

Los mecanismos importantes del PROLOG son**: recursividad, instanciación, verificación, unificación, backtracking e inversión o negación.**

La Recursividad representa la estructura más importante en el desarrollo del programa. En la sintaxis del PROLOG no existen los bucles FOR ni los saltos; los bucles WHILE son de difícil incorporación, ya que las variables sólo pueden unificarse una sola vez. La recursión es más apropiada que otras estructuras de desarrollo para procesar estructuras de datos recursivas como son las listas y destacan en estos casos por una representación más sencilla y de mayor claridad.

La **Instanciación** es la unión de una variable a una constante o estructura. La variable ligada se comporta luego como una constante.

La **Verificación** es el intento de derivar la estructura a comprobar de una pregunta desde la base de conocimientos, es decir, desde los hechos y reglas. Si es posible, la estructura es verdadera, en caso contrario es falsa.

La **Unificación** es el componente principal de la verificación de estructuras. Una estructura estará comprobada cuando puede ser unificada con un hecho, o cuando puede unificarse con la cabecera de una regla y las estructuras del cuerpo de dicha regla pueden ser verificadas.

5 Fundamentos de Prolog

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

Prolog es un lenguaje de programación orientado a la resolución de problemas tal que involucren objetos y relaciones entre estos objetos. Los elementos básicos de Prolog son los siguientes:

1)Hechos
2)Preguntas
3)Variables
4)Conjunciones
5)Reglas.
6)Listas
7)Recursión

5.1 HECHOS

Prolog consiste entonces en la declaración de algunos Hechos acerca de objetos y de sus relaciones, definiendo algunas reglas acerca de los objetos y de sus relaciones. Asimismo se formulan preguntas acerca de los objetos y de sus relaciones. Los sistema Prolog implementa hechos(datos) y de las reglas que se aplican sobre los objetos dando la posibilidad de realizar inferencias de un hecho a otro. Los Hechos acerca de objetos y sus respectivas relaciones lógicas. Para tal caso los nombres de las relaciones y de los objetos deben empezar con minúscula. Las relaciones deben ser escritas primero y los objetos deben ser escritos separados por comas, y lo objetos deben estar encerrado entre paréntesis redondos. El punto . debe venir al final de cada hecho.

Ejemplos de Hechos:

valioso(oro). El oro es valioso.
padre(juan,maría). Juan es padre de María
da(juan,libro,maría). Juan le da un libro a María

Los hechos expresan relaciones arbitrarias entre objetos. Una colección de hechos es una BASE DE CONOCIMIENTOS de un programa PROLOG.

5.2 ARGUMENTOS Y PREDICADOS

Los ARGUMENTOS son los nombres de objetos que se encuentran encerrados por

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

paréntesis redondos en cada Hecho. Los Predicados son los nombres de las relaciones que vienen justo antes del paréntesis redondo.

5.3 PREGUNTAS.

Cuando se tienen algunos Hechos, se pueden hacer preguntas sobre estos, con el símbolo ?, así:

?.- tiene(maría,libro).

La pregunta anterior, llama a la persona María y un objeto particular: libro. La pregunta es ¿Tiene María un libro?. Cuando se hace esta pregunta, Prolog buscará los objetos a través de la base de datos de Hechos. Prolog busca encontrar el objeto equivalente en el hecho de la pregunta. Dos hecho se emparejan si sus respectivos predicados son los mismos y sus respectivos argumentos también son los mismos. En Prolog la respuesta NO es implementada, pero esta nos significa Falsedad sobre la pregunta sino que no se ha encontrado ningún emparejamiento con la pregunta.

5.4 VARIABLES.

Las variables son letras mayúsculas que serán instanciadas por nombres de objetos particulares. Es decir son nombres de un segundo tipo. Las variables pueden instanciarse en los objetos concretos que representan. Asimismo estarán las variables no-instanciadas que no están representado ningún objeto en particular aún.

Si se tienen lo siguientes hechos:

gusta(juan,flores).

gusta(juan,maría).

gusta(pablo,maría).

y se pregunta lo siguiente:

? gusta(juan,X). (es decir, existe alguna cosa que le guste a Juan).

PROLOG responderá:

X = flores colocando ahí el puntero

si luego se presiona RETURN, entonces detendrá la búsqueda,

mientras que si se presiona ; reiniciará la búsqueda desde

donde estaba colocado el puntero. PROLOG intentará resatisfacer la pregunta y X es

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

instanciado de nuevo, así:

X = maría

5.5 CONJUNCIONES.

Las conjunciones sirven para unir dos metas separadas que el sistema Prolog necesita satisfacer. La conjunción de dos metas se representa por una coma,

?.- gusta(juan,maría),gusta (maría,juan).

La pregunta respectiva es ¿Le gusta Juan a María y a María Juan,leyéndose la , como una y, sirviendo para separar cualquier número de metas distintas a ser satisfechas.

El uso de conjunciones y de variables combinadas pueden generar interesantes preguntas. Puede generarse una pregunta como por ejemplo:

Existe algo que le guste a Juan y María. Primero se buscará algún X que le guste a María, entonces buscará cualquier X que a Juan le guste.

En PROLOG las dos metas con conjunción puede ser puesto de la siguiente manera:

?- gusta(maría,X),gusta(Juan,X).

Las respuestas de PROLOG a la preguntan intentarán satisfacer primero la primera meta. Si la primera meta está la base de datos, PROLOG marcará en ese lugar el puntero e intentará satisfacer la segunda meta. Si la segunda meta es satisfecha entonces PROLOG colocará en ese lugar de la base de datos el puntero, y entonces se habrá encontrado la solución que satisface ambas metas. La Conjunción de metas será organizada de izquierda a derecha, separada por coma. Así PROLOG intentará satisfacer una meta a la vez,trabajando de izquierda a derecha. Si una meta es satisfecha, entonces PROLOG lleva el puntero al lugar asociado con la meta en la base de datos. La satisfacción e metas empieza con el lado izquierdo del vecindario. Las variables previamente desinstanciadas pueden ahora ser instanciadas. PROLOG intentará satisfacer las metas del lado izquierdo del vecindario, empezando de la parte superior de la base de datos. En cualquier momento en que una meta falle(no encuentra ningún hecho con que se acople), PROLOG se devuelve e intenta resatisfacer su vecindario izquierdo, empezando desde donde se encuentra el puntero.

PROLOG debe primero "desinstanciar" cada una de las variables que han sido instanciadas en las metas anteriores. PROLOG desinstancia todas las variables cuando resatisface una

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

meta. Esto es conocido como "retrorastreo"(backtraking), es decir PROLOG intenta repetidas veces satisfacer y resatisfacer las metas en una conjunción.

5.6 REGLAS

Las reglas en PROLOG son usadas cuando se quiere señalar que un hecho depende de u grupo de otros hechos. Así el condicional

"Si" expresa una regla.

"Yo uso una sombrilla si está lloviendo"

"Juan compra vino si este es menos caro que la cerveza"

Así se pueden dar definiciones:

X es un pájaro, si:

X es un animal, y

X tiene alas

X es una hermana de Y, si:

X es mujer, y

X y Y tienen los mismos padres.

Es importante recordar que X denota los mismos objetos cuando aparece cada vez en una regla. Así, una regla es una oración general acerca de los objetos y de sus relaciones.

Ejemplo:

"Juan le gusta a cualquiera que le guste vino",

en otras palabras,

"Juan le gusta algo si este le gusta el vino"

en variables:

"Juan gusta X si x gusta vino"

En PROLOG una regla consiste en una cabeza y un cuerpo, así:

CABEZA :- CUERPO

:- es el condicional "Si"

gusta(juan,X):- gusta(X,vino).

Las reglas deben terminar siempre con un punto.

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

CABEZA: gusta(juan,X).

La cabeza de una regla describe que hecho de la regla se intenta definir.

CUERPO: gusta(X,vino).

Describe la conjunción de las metas que deben ser satisfechas, una después de la otra, para hacer la cabeza verdadera.

gusta(juan,X):- gusta(X,vino),gusta(X,comida).

En palabras, Juan gusta de cualquiera que guste vino y comida.

"Juan gusta de cualquier mujer que le guste el vino":

gusta(juan,X):- mujer(X),gusta(X,vino).

En cualquier momento que se mire una regla de PROLOG, debe ser explícito donde está la variable. Cuando cualquier X viene a ser instanciada a algún objeto, todas las X's son instanciadas en el dominio de X. Para algún uso particular de una regla, el dominio de X es toda la regla, incluyendo la cabeza, y extendiéndose hasta el punto al final de la regla. Una regla usa más de una variable en muchos casos. Entonces debe tenerse claro que en cada reglas, cada variable aparecen al menos dos veces.

Una regla no está bien escrita si:

i) las variables aparecen una sola vez en el encabezado y ninguna vez en el cuerpo;pueden ser reemplazadas por variables anónimas

ii) las variables que aparecen una sola vez en el cuerpo y ninguna vez en el encabezado son variables anónimas.

5.7 BASE DE DATOS.

Como ejemplo se puede establecer una base de datos acerca de la familia de la Reina Victoria. Aparte de los hechos con un solo nombre, se puede establecer el predicado: "Progenitores", que tendrían tres argumentos así:

progenitores(X,Y,Z).

Lo que significaría que los progenitores de X son Y y Z.

Y sería el padre y Z la madre.

La base de datos se establecería así:

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

hombre(alberto).

hombre(eduardo).

mujer(alicia).

mujer(victoria).

progenitores(eduardo,victoria,alberto).

progenitores(alicia,victoria,alberto).

La regla para definir el predicado hermana_de, es:

sister_of(X,Y). es verdadera si X es hermana de Y.

En los predicados se pueden unir términos con la línea: _ .

X es una hermana de Y si:

X es mujer, X tiene una madre M y un padre F, y Y tiene la misma madre y padre que tiene X. Se definiría así:

sister_of(X,Y):- mujer(X),progenitor(X,M,P),progenitor(Y,M,P).

M y P indican madre y padre, y aquí se esta usando variables que no aparecen en la cabeza de la regla.

Una persona puede robar algo si la persona es un ladrón y a ella le gusta la cosa y la cosa es valiosa.

puede_robar(P,C):- ladrón(P),gusta(P,C),valioso(C).

aquí el predicado puede_robar, tiene como variables P:persona,

C: cosa.

Esta regla depende de las cláusulas para ladrón, gusta y valioso.

Un predicado puede ser definido por una combinación de hechos y reglas. Estos son llamadas cláusulas para le predicado. Se puede utilizar el término cláusula para referirnos ya sea a un hecho o ya sea a una regla.

En PROLOG se pueden añadir comentarios por conveniencia, que son ignorados de la siguiente manera:

/*...*/

Así se pueden enumerar las cláusulas como ejemplo:

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

/*1*/ladrón(juan).

/*2*/gusta(maría,comida).

/*3*/gusta(maría,vino).

/*4*/gusta(juan,X):-gusta(X,vino).

5.8 SINTAXIS DEL PROLOG

Los programas de PROLOG, son construidos de términos. Pueden darse tres tipos de términos: constante, variables y estructuras.

Cada término se escribe como una secuencia de caracteres. Los caracteres pueden ser divididos en cuatro tipos:

1. Mayúsculas: **A,B,C**

2. Minúsculas **a,b,c**

3. Dígitos **0,1,2**

4. Signos **+ - * / \ <>**

5.8.1 Constantes

Las constantes son entendidas como tipos específicos de cosas o como relaciones específicas. Los átomos pueden ser letras: gusta, maría, juan, o dígitos. ?, :-.

Los enteros son usados para representar números. Son la totalidad de los números consistiendo solo de dígitos y no tienen punto decimal: 0,1,99,512,8192,1475.

Los sistemas Prolog proveen una librería de programas que definen operaciones o números racionales y números de precisión arbitraria.

5.8.2 VARIABLES

Una variable debe ser entendida como la denotación de algún objeto que no estamos en condiciones de nombrar. Las variables pueden representarse como: X,Y,Z, Respuesta,Entrada,Nombre.

Con el carácter _, se establecen las variables anónimas. Un ejemplo es el siguiente:

?-likes(_,juan).

Distintas variables anónimas en la misma cláusula no necesitan tener una interpretación

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

consistente.

5.8.3 ESTRUCTURAS

Una estructura es un simple objeto el cual consiste en una colección de otros objetos, llamados componentes. Los componentes se agrupan juntos una una misma estructura por conveniencia.

Una estructura en la vida real es un tipo de fichero de una biblioteca. Las estructuras ayudan a organizar los datos de un programa porque estas permiten agrupar una determinada información y puede ser tratada como un objeto simple a pesar de estar constituida por entidades separadas. Una estructura puede ser usada cuando hay un tipo común de objetos, los cuales muchos pueden existir.

Una estructura puede ser escrita en PROLOG por especificaciones de su functor y de sus componentes. Los componentes son encerrados en paréntesis redondos y separados por comas. El functor se escribe justo antes de abrir los paréntesis

Se tiene lo siguiente:

"Juan es dueño del libro llamado Wathering Heights, escrito por Emyly Bronte":

dueño(juan,libro(wathering_heigths,autor(emily,bronte))).

Así se puede preguntar:

?-dueño(juan,libro(X,autor(Y,bront))).

dueño(juan,libro(ulyses,autor(james,joyce),3129)).

que representa a "juan es dueño de 3129ava. copia del Ulyses de James Joyce.

La sintaxis para las estructuras es la mima que para los hechos de PROLOG. Un predicado(que usa hechos y reglas) es en las estructuras un functor. Los argumentos de un hecho o de una regla vienen a ser los componentes de la estructura.

5.9 ESTRUCTURAS DE DATOS.

La Recursión es una muy popular y poderosa técnica en el mundo de programación no-numérica. Esta puede sr usada en dos sentidos:

1)Para describir estructuras que tienen otras estructuras como componentes.

2)Para describir programas que necesitan satisfechos una copia de ellos mismos

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

antes de que ellos puedan ejecutarse.

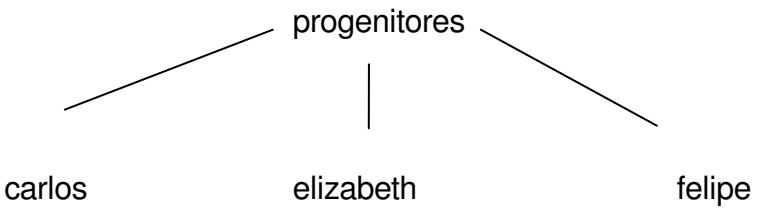
En PROLOG, recursión es un mecanismo normal y natural de organizar las estructuras de datos y los programas.

5.10 ESTRUCTURAS Y ARBOLES.

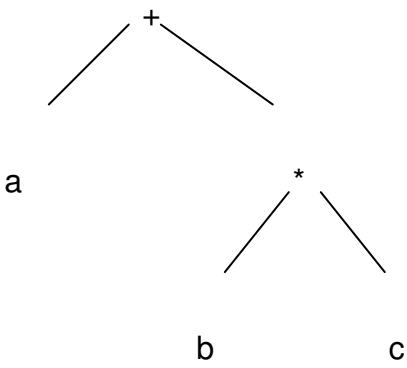
En un árbol cada functor es un nodo y los componentes son sus ramas. En un diagrama la raíz estará en la parte superior y las ramas en la parte inferior

Así se tiene:

progenitores (carlos,elizabeth,felipe).

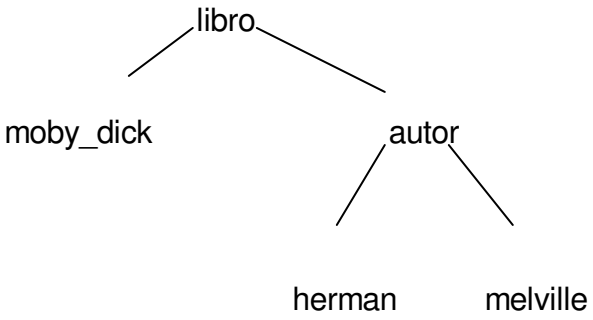


a + b * c (o+(a,*(b,c))



RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

libro(moby_dick,autor(herman,meville)).

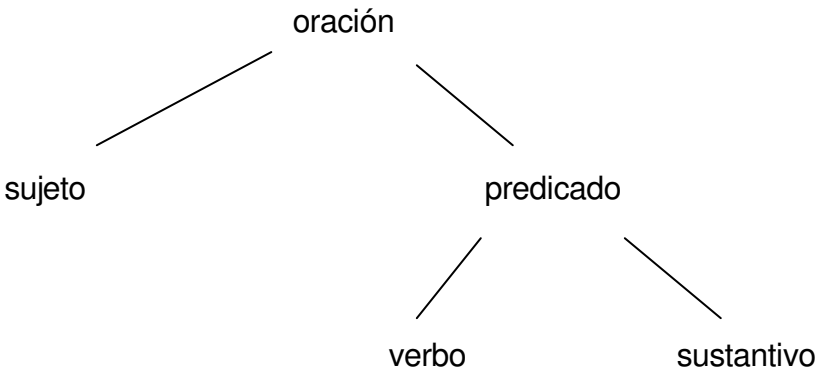


Así se tiene la siguiente oración:

"A Juan le gusta María"

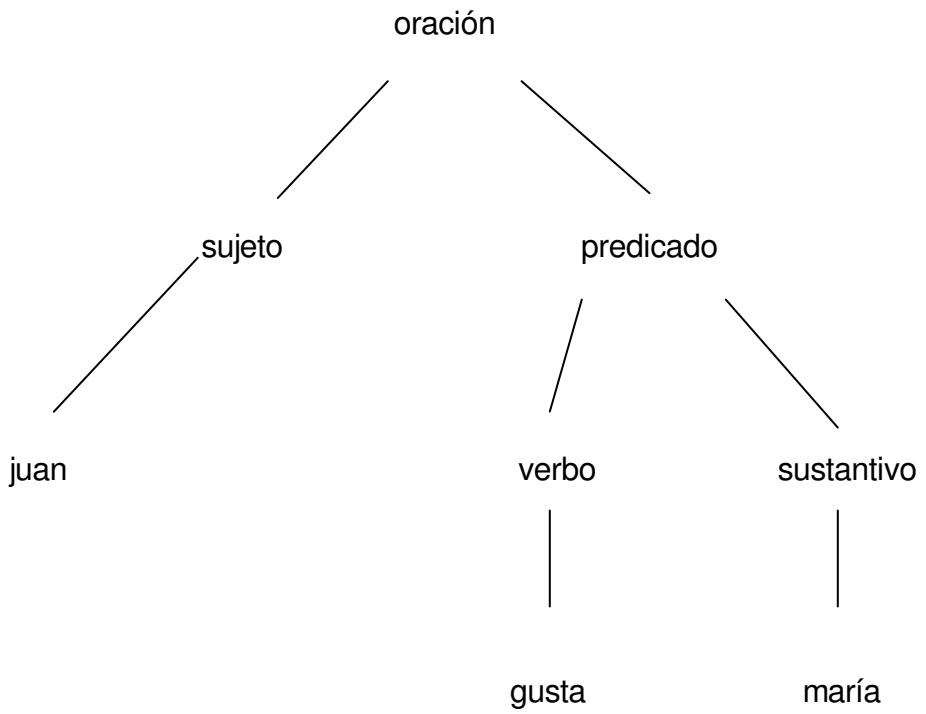
Se necesita representar la sintaxis de la oración. La sintaxis puede ser descrita como un Sujeto y un Predicado. El predicado consistirá también de un verbo y un sustantivo. Esta relación entre las partes de una oración puede ser descrita por una estructura:

oración(sujeto,predicado(verbo,sustantivo))

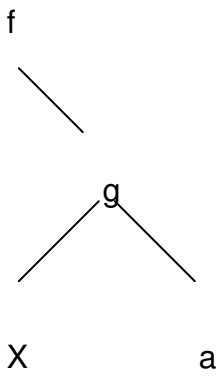


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

oración(sujeto(juan),predicado(verbo(gusta),sustantivo(maría)))



Asimismo: f(X,g(X,a)).



5.11 LISTAS

La lista es una estructura muy común de datos en programación no numérica. La lista está ordenada como una secuencia de elementos que pueden tener cualquier extensión. El

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

ordenamiento significa que el orden de los elementos se presenta en una secuencia de materias. Los elementos de una lista pueden ser, constantes, variables, estructuras, en inclusive otras listas. La listas pueden representar practiacmente cualquier tipo de estructuras que puedan ser requeridas en computación simbólica. La listas pueden representar, árboles, gramáticas, ciudades, mapas, programas de computación, y entidades matemáticas como gráfos, formulas y funciones.

Existe un lenguaje de programación llamado LISP, en el cual las únicas estructuras de datos disponibles son las constantes y las listas. No obstante, en PROLOG la lista es simplemente un tipo particular de estructura.

Las listas pueden ser representadas como un tipo especial de árbol:

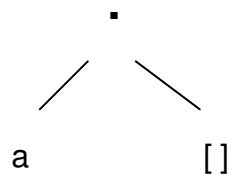
1.- Una lista puede ser una lista vacía, y no tiene elementos

2.- O puede ser una estructura que tiene dos componentes,

la cabeza y la cola.

El final de una lista es comúnmente representada como una cola que es el conjunto de la lista vacía. La lista vacía es escrita como [ ], los cual es un paréntesis cuadrado que abre y uno que cierra.

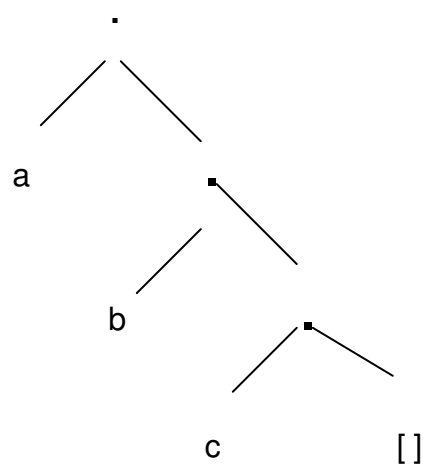
La cabeza y la cola de una lista son los componentes de un functor denominado " . ", que es un punto. Así la lista consistente de un elemento "a" es .(a,[]), y su árbol sería el siguiente:



Asimismo la lista compuesta por los átomos: a,b, y c, puede ser escrita:

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

(a,.(b,.(c,[]))), y graficado como:



El functor punto, se presenta de tal manera que puede ser definido como un operador, si es permisible escribir sobre dos listas como:

a . []

a .(b . (c. []))). a.b.c.[]

El punto es asociativo hacia la derecha.

Las listas están ordenadas como una secuencia ordenada, así la lista a . b es diferente de la lista b . a .

Alguna gente gusta escribir los diagramas arbóreos de una lista con un árbol creciente de izquierda a derecha, y con ramas colgando. Estos es de la siguiente manera:

. _ . _ ._[]

a b c

En este diagrama , el componente cabeza cuelga del functor punto, y el componente cola crece hacia la derecha. El final de la lista está claramente señalado como el último componente de la cola que viene a se la lista vacía.

5.12 LISTAS EN UN PROGRAMA PROLOG

La notación de listas en PROLOG consiste en que los elementos deben presentarse

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

separados por comas y todas la lista debe estar encerrada en paréntesis cuadrados. Es usual para las listas que contienen otras listas variables.

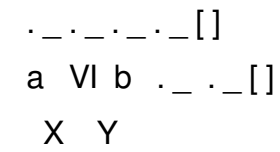
Por ejemplo

[ ]

[el, hombre,[gusta,de pescar]]

[a,Vl,b,[X,Y]]

En un diagrama Vine:



Se puede ver fácilmente en el diagrama que cada nivel horizontal del diagrama es una lista que tiene cierto número de elementos. El nivel superior es una lista que tiene cuatro elementos, de los cuales uno es otra lista. En el segundo nivel, se tienen dos elementos. Las listas son manipulables si se las divide en una cabeza y una cola. La cabeza de una lista es el primer argumento para construir listas. Debe tenerse en cuenta que se habla de "cabeza" de una regla como también se habla de "cabeza" de una lista. Estas dos cosas son diferentes, y aunque ambas se les denomina "cabezas" por un accidente histórico, es claramente reconocible de que "cabeza" se está hablando en cada contexto. La cola de una lista es el segundo argumento del functor . Cuando aparece la notación del paréntesis cuadrado, la cabeza de la lista es el primer elemento de la lista. La cola de la lista es la lista que contiene todos los elementos excepto el primero.

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

5.13 CABEZAS Y COLAS

LISTA	CABEZA	COLA
[a,b,c]	a	[b,c,d]
[a]	a	[ ]
[ ]	sin cabeza	sin cola
[[el,gato],sentado]	[el,gato]	[sentado]
[el,[gato,sentado]	el	[[gato,sentado]]
[X + Y, X + Y]	X + Y	[X + Y]

Debe notarse que la lista vacía no tiene ni cabeza ni cola. En el último ejemplo, el operador + es usado como un functor de las estructuras +(X,Y) y +(X,Y). En tanto que la opeacion de dividir una lista en cabeza y cola, existe una notación especial en PROLOG para representar la lista con una cabeza X y una cola Y. Se escribe de la siguiente manera:

[X|Y]

donde el símbolo de separación de X y Y es una barra vertical.

Un modelo de como esta forma de las listas en el cual X puede ser isntanciada a la cabeza de una lista y Y a la cola de la lista es la siguiente:

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

P([1,2,3]).

P([el,gato,sentado,[sobre,la,alfombra]]).

?-P(X|Y]).

Soluciones PROLOG:

X=1 Y=[2,3] ;

X=el Y=[gato,sentado,[sobre,la,alfombra]]

?-(_,_,_,[_|X]).

X=[la,alfombra].

Más ejemplos de la sintaxis de una lista, puede mostrar como varias listas se emparejan, son las siguiente, en los cuales se intenta emparejar las dos listas mostradas, obteniendo las instanciaciones si es posible.

LISTA 1	LISTA 2	INSTANCIACIONES
[X,Y,Z]	[juan,gusta,pescado]	X=juan,Y=gusta Z=pescado
[gato]	[X|Y]	X=gato,Y=[ ]
[X,Y|Z]	[maría,gusta,vino]	X=maría,Y=gustaZ Z=[vino]
[[el,Y]|Z]	[[X,tiene],[esta,aquí]]	X=el,Y=tiene Z=[[esta,aquí]
[X,Y|Z,W]	sintaxis incorrecta	
[oro|T]	[oro,amarillo]	T=[amarillo]

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

[valiente,caballo]	[caballo,X]	falla
[blanco|Q]	[P|caballo]	P=blanco, Q=caballo

Es posible usar la notación de listas para crear estructuras que incorporen listas, pero las cuales no terminan con la lista vacía.

ESTRUCTURA: [blanco|caballo]
cabeza: blanco
cola: caballo(constante)

5.14 Cláusulas de Horn y representación del conocimiento

En el contexto del Prolog, el conocimiento se representa en forma de "**cláusulas de Horn**", cláusulas con, a lo sumo, un literal negativo. Las cláusulas traducen, "hechos" (en el contexto de los vinos y los quesos franceses) tales como:

acuerdo (géromé, pinot_negro),

que puede interpretarse como que "el pinot negro" encaja "con el queso gérome";
o "reglas" tales como:

misma región (X,Y): -regionqueso (X,U),
regionvino (Y,U),

que puede significar: **"Cierto queso X es el de la misma región de cierto vino Y, pues ambos, X e Y, provienen de la región U."**

Se escribirán las **constantes** (como géromé) con minúsculas y las **variables** (como **X, Y o U**) con mayúsculas. La **coma** corresponde a la **"y"** lógica.

Una cláusula del tipo:

(parte izquierda): - (parte derecha),

se lee "para demostrar (parte izquierda), basta con demostrar (parte derecha)".

Un programa PROLOG propiamente dicho se compone de una serie de **cláusulas FL,** a

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

las cuales se agrega, para una ejecución particular del programa una cláusula de la que quiere saberse si es una consecuencia lógica de FL o no lo es. El programa se **"interpreta"** en la ejecución gracias a un mecanismo de resolución controlado por la estrategia llamada del **"soporte",** que funciona en profundidad con vuelta atrás o retrorastreo(backtraking). Por ejemplo, en el mundo de los quesos y los vinos se puede establezcer:

Acuerdo (X,Y) significa que siendo X un queso e Y un vino, X e Y encajan entre sí.

Y se propone el siguiente programa PROLOG (donde se han numerado las cláusulas) para una mejor compresión de la explicación que sigue:

(1) acuerdo (géromé, pinot_negro).

(2) acuerdo (roquefort, chateauneuf _ du_pape) .

(3) acuerdo (laguiole, cahors).

(4) acuerdo (X,Y) : -queso_de_cabra (X), misma región (X,Y).

(5) acuerdo (X,Y):-pasta_blanca (X), color (Y, rojo), cuerpo (Y, cremoso).

(6) acuerdo (chaource, Z): - vino natural de champagne (Z).

(7) vino_natural_de_champagne (buozy).

(8) misma region (X,Y) :-regionqueso (X,U), regionvino (Y,U).

(9) regionvino (chassagne, borgoña).

(10) queso_de_cabra (charolles):

(11) queso_de_cabra (pelardon).

(12) regionqueso (charolles, borgoña).

(13) pasta_blanca (géromé).

(14) color (gewurtztraminer, blanco).

(16) color (morgon, rojo).

(17) color (fleurie, rojo).

(18) cuerpo (morgon, cremoso).

(19) cuerpo (fleurie, liviano).

Una vez que este **programa se haya introducido en la memoria**, y se inicie su ejecución mediante el siguiente interrogatorio:

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

¿Qué vino se servirá con el Camembert?, que se escribirá:

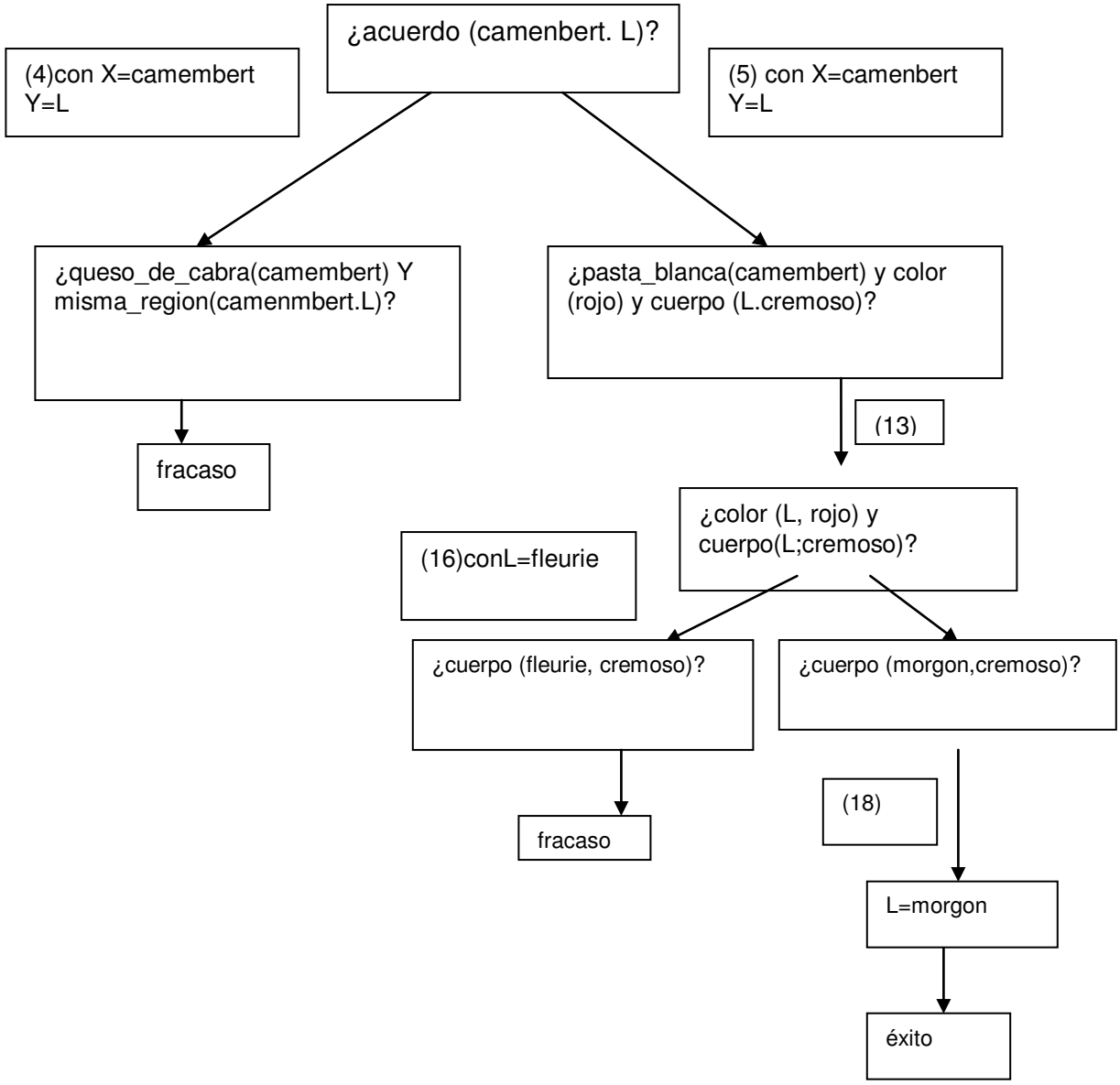
¿acuerdo (camembert. L)?,

donde L es una variable cualquiera.

El intérprete PROLOG recorrerá las cláusulas del programa, tratando de hallar la L que satisfaga a la cláusula-interrogación, y responderá:

L = morgon.

El siguiente esquema indica el razonamiento seguido. A las preguntas:



RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

¿pasta_cocina (F)?, o:

¿acuerdo (M, gewurtztraminer)?

El sistema no responderá. Estas preguntas fuera de su competencia, la primera por el **desconocimiento del predicado**: pasta cocina; la segunda porque no conoce el queso que este de acuerdo con el **gewurtztraminer** y no llega a encontrarlo mediante el razonamiento.

Para terminar se busca qué vino pedemos servir con un queso de cabra. Para eso, se inicia la ejecución del programa con:

¿**acuerdo (V,W), queso_de_cabra(V)?**

La respuesta dada es: V = charolles W=chassagne.

Se observará que el sistema, aunque sabe que el pelardon es un queso de cabra, no puede asociarse ningún vino. La respuesta no menciona, pues, este queso.

En otras situaciones, por ejemplo en la búsqueda de todos los vinos rojos por:

¿color (X, rojo)?

El sistema proporcionará una respuesta múltiple:

X=morgon.

Y=fleurie.

5.15 Bases de datos deductivas

La bases de datos deductivas, son aquellas en las cuales las técnicas de **inferencia lógica** encuentran una aplicación en la resolución de un problema expresado en forma de consulta a una base de datos que permite deducir una información que no esté explícitamente clasificada en la base. Los diferentes modelos actuales corresponden a un acoplamiento mas o menos sólido entre el **sistema de gestión de bases de datos (SGBD) clásico**, adaptado a la manipulación de grandes cantidades de información estructuradas de manera uniforme, y **un mecanismo de deducción del tipo PROLOG**.

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

5.16 Solución de Problemas

Los procesos de un sistema experto se asocia a los procesos inteligentes en el ser humano

Proceso ``normal''

- identificación y definición del problema
- identificación del criterio de evaluación
- generación de alternativas
- búsqueda de una solución y evaluación
- selección de opción y recomendación
- implementación

Solución de problemas en IA: basicamente búsqueda y evaluación

Representación de espacio de estados

- define un espacio de estados (espacio con todas las posibles soluciones potenciales implícita / explícitamente enumerado
- especifica los estados iniciales o situaciones de donde empezar
- especifica los estados finales (metas) o aquellos reconocidos como soluciones
- especifica las reglas que definen las acciones u operaciones disponibles para moverse o ir de un estado a otro dentro del espacio del problema

En este contexto el proceso de solución de problemas trata de encontrar una secuencia de operaciones que transformen al estado inicial en uno final

En la práctica se necesita seguir una estrategia de búsqueda

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

5.16.1 Búsqueda

- Necesaria
- Heurísticas/criterios/métodos/principios/...
- Criterios
- simple
- discriminante entre buenas/malas opciones :
- Solución en un solo paso:
- ver la solución

5.16.2 Alternativas:

- solución incremental
- sistemática
- Medios (espacio de estados):
- transformar (hasta darle)
- construír (poco a poco)

5.16.3 Factores a considerar:

- calidad de la solución (a veces puede no importar, e.g., prueba de teoremas)
- diferencia en complejidad entre una solución y la solución óptima puede ser gigantesca.
- en general, cómo encontrar la solución más barata

5.16.4 Que se necesita:

1. Estructura simbólica que represente subconjunto de soluciones potenciales (código o base de datos o agenda)

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

2. Operaciones/reglas de producción que modifiquen símbolos de la base de datos y produzcan conjuntos más refinados de soluciones potenciales

3. Procedimiento de búsqueda o estrategias de control que decida que operación aplicar a la base de datos

Terminología: nodo, árbol, hoja, nodo-raíz, nodo-terminal, *branching factor*, ramas, padres, hijos, árbol uniforme, ...

nodos expandidos (*closed*) (todos los sucesores)

nodos explorados pero no expandidos (solo algunos sucesores)

nodos generados pero no explorados (*open*)

nodos no generados

Paso computacional primordial: expansión de nodos

5.16.5 Propiedades

La estrategia de control es *sistemática* si:

1. no deja un solo camino sin explorar (completo)
2. no explora un mismo camino más de una vez (eficiencia)

5.16.6 Propiedades de algoritmos de búsqueda (heurísticas):

1. *Completo*: un algoritmo se dice que es completo si encuentra una solución cuando ésta existe
2. *Admisible*: Si garantiza regresar una solución óptima cuando ésta existe
3. *Dominante*: un algoritmo *A*1 se dice que domina a *A*2 si todo nodo expandido por *A*1 es también expandido por *A*2 (``más eficiente que'')
4. *Optimo*: un algoritmo es óptimo sobre una clase de algoritmos si domina todos los miembros de la clase

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

5.16.7 Depth first - backtracking (LIFO)

Crea una agenda de un elemento (el nodo raiz) *hasta* que la agenda este vacia o se alcance la meta *si* el primer elemento es la meta *entonces* acaba *si no* elimina el primer elemento y anade sus sucesores al *frente* de la agenda. Problemas: árboles con caminos de profundidad muy grande variaciones:

- depth-bound (casi todos): limitar la búsqueda hasta cierto límite de profundidad
- Con algo de información: ordena los nodos expandidos

5.16.8 . Breadth first

Crea una agenda de un elemento (el nodo raiz) *hasta* que la agenda este vacia o se alcance la meta *si* el primer elemento es la meta *entonces* acaba *si NO* elimina el primer elemento y añade sus sucesores al *final* de la agenda.Problemas: árboles con arborecencia muy grande.

5.16.9 Búsqueda en Prolog

```
busca(NodoI,NodoF) :-
busca_aux([NodoI],NodoF).

busca_aux([NodoF|_],NodoF).
busca_aux(Agenda,NodoF) :-
nva_agenda(Agenda,NAgenda),
busca_aux(NAgenda,NodoF).

% depth-first
nva_agenda([N1|Agenda],NAgenda) :-
expande(N1,Nodos),
append(Nodos,Agenda,NAgenda).

% breadth-first
nva_agenda([N1|Agenda],NAgenda) :-
expande(N1,Nodos),
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
append(Agenda,Nodos,NAgenda).

% best first
nva_agenda([N1|Agenda],NAgenda) :-
expande(N1,Nodos),
append(Nodos,Agenda,AgendaInt),
sort(AgendaInt,NAgenda).

% hill-climbing
nva_agenda([N1|Agenda],[Mejor]) :-
expande(N1,Nodos),
append(Nodos,Agenda,AgendaInt),
sort(AgendaInt,[Mejor|_]).
```

5.17 Otros contextos lógicos

Más allá de la denominada lógica clásica, se han desarrollado algunas denominadas lógicas alternativas. En el contexto de la lógica clásica, las reglas propuestas se dan como una generalización de la relación que justifica la inferencia, lo que no daba lugar a dudas acerca de la interpretación del signo básico correspondiente. Pero en un sistema como el de la *Begriffsshrift* de Frege o los Principia Mathematica podría ser desarrollado sin atender para nada a la interpretación de sus signos, de suerte que –abstractamente considerado- cupiera ver en él una alternativa a varias posibles.

5.17.1 Las lógicas multivaluadas.

En el trabajo de 1921 de E. L. Post: *"A General Theory of Elementary Propositions",* en el que introducióa el método de las tablas de verdad para la lógica elemental, sugería a la vez, la consideración de un sistema formal alternativo en el que cada variable pudiera tomar no simplemente uno u otro de los valores de verdad v y f, sino cualquiera de los m diferentes valores v_1, v_2 …v_m. Estos sistemas alternativos, constituyen una extensión de la lógica de predicados, con una cantidad n de valores de verdad superior a 2. Esto permite dar cuenta de la ambigüedad o la incertidumbre en la expresión del conocimiento. Si se hace de n tienda hacia el infinito se reencuentra la noción de coeficiente de verosimilitud

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

5.17.2 Las lógicas modales.

Destinadas a expresar mejor la noción de intuitiva de implicación. Se introducen, para eso, varias modalidades (como: posible, necesario, contingente e imposible) que sirven para precisar una formula lógica. El moderno interés por la lógica modal (pues ya Aristóteles le había dedicado algunos apartados de sus tratados lógicos), tiene su inicio en la obra de C.I. Lewis, publicada por vez primera en forma de libro en su *Survey of Simboloic Logic* de 1918. Esta teoría se conoce comúnmente con el nombre lógica de la implicación estricta (strict implication), por haber sido originariamente propuesta frente a una caracterización de la implicación que Lewis juzgaba equivocada, pues tiene la plausibilidad de generar paradojas, una proposisicón falsa podrá implicar cualquier otra proposición y una verdadera podrá ser implicada por cualquier otra. Lewis sostenía, en cambio que una proposición implicaría estrictamente otra si, y sólo si, es **imposible** que la primera sea verdadera y la segunda falsa, escribiendo $P \pi Q$, para expresar tal relación entre las proposiciones expresadad por P y Q. La caracterización definitiva de la lógica modal por parte de Lewis se encuentra en su *Simbolic Logic* de 1932, publicada en colaboración con C.H. Langford. En es se toma el símbolo $\Diamond$ como indefinido y se introduce $P \pi Q$ como abreviatura de $\neg\Diamond(P \wedge \neg Q)$.

Definición:

$$\mathbf{(P \pi Q) \Leftrightarrow \neg\Diamond (P \wedge \neg Q)}$$

Que significará que P implica estrictamente Q y si (y solamente si) es imposible tener al mismo tiempo P y no Q.

De esta manera se establece el operador modal de posibilidad $\Diamond$**, por lo que** $\Diamond$**P, es la abreviatura de "es posible que P". Por otra parte** $\neg\Diamond\neg$**P se interpreta como "es necesario que P", y tendrá como abreviatura:** $\Box$ **P. A partir de ello es posible establecer la siguiente definición:**

$$\mathbf{(P \pi Q) \Leftrightarrow \Box (P \rightarrow Q)}$$

Diversos autores han interpretado algunos sistemas de la lógica modal como derivados de los Principia con la ayuda de algunos axiomas o reglas de inferencia supletorios.

A partir de las operaciones de la lógica modal, autores como Von Wright, han sugerido

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

coparar las **modalidades aléticas** con las **epistémicos**, esto es, las modalidades del tipo de las que manejaba Lewis con las expresadas en cláusulas del tipo: **"se sabe que"**, lo que da la posibilidad al origen de la **Lógica epistémica**, que da lugar al análisis de discursos "epistémicos", que incluyen operadores de **conocimiento y creencia** aplicados por algún agente inteligente. De la misma manera que con los operadores modales, dentro los contextos epistémicos, no parecen regirse por la lógica clásicas proposicional o de primer orden con identidad.

5.17.3 Las lógicas no monótonas.

En las cuales una aseveración verdadera, en un momento dado del razonamiento, puede revelarse como falsa a continuación. Típicamente estos patrones se implantan usando el mecanismo llamado **negación-por-fracaso** que ocurre como un **operador explícito** en lenguajes de programación lógica. En Prolog, por ejemplo, la meta **no-G** tiene éxito **si y solo si G fracasa en tiempo finito**. Puesto que el fracaso de G es equivalente a no haber podido hallar una prueba de G usando el programa Prolog como axiomas, el operador no representa la ausencia de una prueba finita. De esta observación podemos ver **que la negación de Prolog es un operador no-monotónico**. Si G no se puede probar desde ciertos axiomas, ello no quiere decir que tenga que permanecer sin prueba bajo un conjunto de axiomas aumentado. La **negación procedimental** es casi siempre identificada con la negación real –es decir, la negación lógica–. La manera en que la negación procedimental es usada en la práctica en los programas lógicos es invocando la regla de inferencia:***"Del fracaso de G, inferir ~ G".*** Esto es realmente **el supuesto del mundo cerrado**, que se encuentra antes en el contexto de la **representación de información negativa en las bases de datos.** En parte porque es **un operador no-monotónico, la negación procedimental puede a menudo ser usada para implantar otras formas de razonamiento por omisión.**

El siguiente ejemplo, un programa PROLOG para razonar sobre pájaros que vuelan, ilustra esto.

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

vuela(x) si pájaro(x) & no ab(x).
pájaro(x) si avestruz(x).
pájaro(x) si canario(x).
ab(x) si avestruz(x).
avestruz(Eusebio).
canario(Piolín).

no vuela(Eusebio)? tiene éxito.
vuela(Piolín)? tiene éxito.

Nótese que la primera regla usa un predicado ab que significa anormal. Así, esta regla dice que x tiene éxito si x no es un pájaro anormal, en otras palabras, si x es un pájaro normal. La cuarta regla describe una circunstancia bajo la cual algo es anormal, a saber, cuando es una avestruz.... Identificando negación procedimental con negación real podemos derivar que el avestruz Eusebio no vuela, mientras que el pájaro Piolín si lo hace.

Como se ha señalado, una lógica de este tipo permite traducir en partículas **el razonamiento por defecto o niega**: en este caso se menciona un vino **V** (esta mayúscula designa aquí un vino particular, es decir, es una constante) que proviene de la zona de Champagne y, mientras no se demuestre lo contrario vamos a suponer que es espumoso, según la aseveración:

(V x)(regiovino (x, champagne) $\Leftrightarrow$ **textura (x, espumoso)**

Si durante el razonamiento seda cuenta de que el vino V del que hablamos lleva la denominación de “vino de champagne”, mientras sabemos que:

(V x)(vino de champagne (x) $\Rightarrow$ **regionvino(x, champagne))**

(**V x)(vino de champagne (x)** $\Rightarrow$ **textura(x, no espumoso)),**

Se puede entonces en decisión la veracidad de “V es espumoso”, así como todas las consecuencias lógicas que pudieron derivarse de este conocimiento que se supone verdadero. Algunos lenguajes fundados en esta lógica, como el PLANNER, destinado a la

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

generación de planes, permiten dictar reglas del tipo siguiente:

"Si A pertenece a cierta categoría de individuos y si a no-P(A) no se lo conoce como verdadero, entonces suponer P(A)".

5.18 Soporte electrónico sobre Lógica teórica y Prolog

En lo que respecta a programas electrónicos orientados a distintas áreas de la Lógica, hay muchas posibilidades. Se recomienda: ***The Propositional Calculus Program,*** desarrollado por J. Kennedy, en lo referido al cálculo proposicional y Mapas de Karnaugh. Consecuentemente se recomienda visitar el sitio Web denominado: **El Mundo de la Lógica, en: http://clientes.vianetworks.es/empresas/lua911/logica.html,** con recursos en línea sobre lógica formal, en castellano. Muy especialmente se recomienda la página de **Christian Gottschall**, denominada: **Gateway to Logic** (Versiones en alemán e inglés). Es una colección de **programas lógicos basados en la Web** (**Probadores de teoremas, constructores y verificadores de pruebas formales**). Un apoyo a estos programas lógicos en línea y en castellano es: **Cibernous: Lógica: Teoría y Praxis: en: http://www.cibernous.com/logica/logica-central.html y**

http://www.cibernous.com/logica/Otros-programas.html

Adicionalmente se recomienda consultar: **Logic Toolbox:**

http://philosophy.lander.edu/~jsaetti/Welcome.html, donde se pueden encontrar recursos electrónicos aplicables al área de la lógica y se pueden descargar instaladores de software de lógica categorial y proposicional.

Es posible conseguir en inglés, una de las más actualizadas producciones en el campo de la lógica: un libro y su respectivo software en: **Language, Proof and Logic**, de **Jon Barwise, John Etchemendy,** et al. , en la siguiente dirección: **http://www-csli.stanford.edu/LPL/.** De la misma manera para el diseño lógico de puertas y circuitos lógicos, se recomienda el software: ***Electronic WorkBench***, del Center for Engineering Computing.

Existen distintas **versiones de Prolog**. El primero en surgir fue del **Prolog de Marsella**, dearrollado por Colmerauer. Como opuesto a este, surge el Prolog de Edimburgo, un

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

nuevo dialecto que eventualmente se convertiría en el Prolog de "estándar". Cabe señalar el **Prolog de Edimburgo**, surge como un Interprete, creado como una implementación del modelo de Prolog desarrollado por **Clocksin y Melish** (Programing in Prolog), que ha servido de fundamento para lo aquí expuesto. Sobre la sintaxis de este Prolog se construyó en 1989/90 (Open University), el MIKE (Micro Interpreter for Knowledge Engineering). Un ambiente en el interprete del Prolog de Edimburgo, en el que es posible acceder **bases de conocimientos** previamente creadas e interrogarlas por medio de cláusulas de preguntas. Estas bases de conocimientos implementan la experticia de sus creadores. Estas bases de conocimiento en conjunto con sus reglas estructuradas en la sintaxis del Prolog de Edimburgo, y principalmente recursivas, se obtienen entonces pequeños **sistemas expertos.**

Un compilador es el **Turbo Prolog**, creado por la compañía Borland (1986/88), con un ambiente un tanto similar al famoso Turbo-Pascal. Es necesario definir los símbolos por utilizar y diferenciar entre estructuras y cláusulas (reglas). Esto facilita la inicialización a los nuevos programadores, no obstante limita los alcances recursivos del Prolog. Un compilador que supera esta limitación es el **Win-Prolog(1996)** de Logic Programming Associates Ltd.

Desarrollado por Prolog Development Center A/S (H.J. Holst Vej 3-5C - 2605 Broendby, Denmark), esta: **Visual Prolog.** Fue creado, con la posibilidad de construir aplicaciones para plataformas MS Windowos 32. Con el Visual Prolog se pueden implementar estructuras de conocimiento complejo. Se puede descargar una versión gratuita en: **http://www.visual-prolog.com**

5.18.1 DECsystem-10 Prolog

El DECsystem-10 Prolog fue una implementación creada y desarrollada **(1977, 1980)** por **D. Warren y otros (F. Pereira, L. Byrd, L. Pereira) en el Departamento de Inteligencia Artificial de la Universidad de Edimburgo,** para operar en sistemas informáticos DECsystem-10. Junto con el Prolog descrito en la obra referenciada de Clocksin y Mellish, las versiones del DEC-10 Prolog y sus posteriores desarrollos constituyen la base del **estándar o sintaxis de Edimburgo**. ("***DECsystem-10 Prolog User's Manual***". D.L.

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

Bowen (ed.), L. Byrd, F.C.N. Pereira, L.M. Pereira, and D.H.D. Warren, Department of Artificial Intelligence, University of Edinburgh, November 10, 1982.)

Sobre la base de este primer intérprete-compilador para Prolog se creó, en 1983, uno de los primeros intérpretes puros para dicho lenguaje, C-Prolog (F. Pereira, D. Bowen, L. Byrd), escrito en lenguaje C, que contribuyó a consolidar el estándar de Edimburgo, como referencia para posteriores implementaciones.

El Prolog para el DECsystem-10 ofrece al usuario un ambiente de programación con herramientas para construir programas, depurarlos siguiendo su ejecución a cada paso y modificar partes de los programas sin tener que volver a comenzar todo el proceso desde el último error. El texto en un programa en Prolog se crea en un archivo o en un número de archivos usando cualquier editor de texto estándar. El intérprete de Prolog puede ser instruido para que lea los programas contenidos en estos archivos. A esta operación se le llama consultar el archivo.

5.18.1.1 Comienzo

Para correr el intérprete de Prolog, se deberá teclear el siguiente comando:

.r prolog

El intérprete responde con un mensaje de identificación y el indicador "| ?- " se presentará cuando el sistema esté listo para aceptar cualquier entrada, así, la pantalla se verá como sigue:

Edinburgh DEC-10 Prolog version 3.47

University of Edinburgh, September 1982

| ?-

En este momento el intérprete estará ya esperando que el usuario ingrese cualquier comando por ejemplo: una pregunta o una orden (ver sección 1.4). No se pueden introducir cláusulas inmediatamente (ver sección 1.3). Este estado se llama nivel alto del intérprete. Mientras que se teclea un comando, el indicador permanecerá de la siguiente manera:

"| ".

El indicador "?-" aparecerá solamente en la primera línea.

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

Lectura de archivos con programas

Un programa esta compuesto por una secuencia de cláusulas que se pueden intercalar con directivas para el intérprete. Las cláusulas de un procedimiento no necesariamente tienen que ser consecutivas, pero si es necesario recordar que el orden relativo es muy importante.

Para Ingresar un programa desde un archivo, sólo se tiene que teclear el nombre del archivo dentro de corchetes y seguidos de un ENTER:

| ?- [archivo] ENTER

Esta orden le dice al intérprete que tiene que leer (consultar) el programa. La especificación del archivo tiene que ser un átomo de Prolog, también puede contener especificación de dispositivos y/o una extensión, pero no se debe incluir la ruta de ubicación. Notese que es necesario que todas estas especificaciones se coloquen dentro de comillas, por ejemplo:

| ?- ['dska:myfile.pl'].

Después de haber tecleado esto el archivo será leído,

Así las cláusulas que se han guardado en un archivo están listas para ser leídas e interpretadas, mientras que las directivas son ejecutadas al momento de ser leídas. Cuando se encuentra el fin del archivo, el intérprete pone en pantalla el tiempo que se ha gastado en la lectura y el número de palabras ocupadas por el programa. Cuando este mensaje se despliegue quiere decir que el comando se ha ejecutado con éxito.

También se pueden combinar varios archivos de la siguiente manera:

| ?- [miprograma,archivos_extras,archivos_de_tests].

En este caso los tres archivos serán leídos.

Si un nombre de archivo se antecede del signo "menos" (-), por ejemplo:

| ?- [-tests,-fixes].

Entonces este archivo o estos archivos serán releídos. La diferencia entre la lectura simple y la releída es que cuando un archivo es leído entonces todas las cláusulas en el archivo se añaden a la base de datos interna de Prolog, mientras que si se lee el archivo dos veces, entonces se tendrá una copia doble de todas las cláusulas.

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

Sin embargo, si un archivo es releído entonces las cláusulas de todos los procedimientos en el archivo reemplazarán a cualquier cláusula existente anteriormente. La reconsulta es útil para indicarle a Prolog las correcciones que se realizan a los programas.

Ingreso de las cláusulas a la terminal

Las cláusulas se puede ingresar directamente en la terminal pero esto es recomendado solamente cuando éstas no se necesitan permanentemente y cuando son pocas. Para ingresar cláusulas desde la terminal, se debe dar el comando especial: give the special command:

| ?- [user].

|

y el indicador "| " mostrara que el intérprete está ahora en el estado de espera de cláusulas o directivas. Para ingresar al nivel más alto del intérprete se deberá teclear Contro + Z (^Z). Este es el equivalente al fin de archivo de archivo temporal virtual llamado 'user'

Directivas: Preguntas y comandos

Las directivas pueden ser preguntas y/o comandos. Ambas pueden ser formas de dirigir al sistema para alcanzar alguna o algunas metas. En el siguiente ejemplo, la función de pertenencia a una lista se defina como sigue:

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```

(Notese que las variables sin valor asignado se representan como "_".)

La sintáxis completa para una pregunta es "?-" seguida por una secuencia de metas, por ejemplo:

?- member(b,[a,b,c]).

Al nivel más alto del intérprete (en el que el indicador es como este: "| ?- "), una pregunta se puede teclear abreviando el "?-" que de hecho ya está incluído en el indicador. Así, en el nivel más alto, una pregunta se vería de la siguiente manera:

| ?- member(b,[a,b,c]).

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

Recuerde que los términos en Prolog deben terminar con el punto ".", por lo que el Prolog no ejecutará nada hasta que Usted haya tecleado este punto y un ENTER al final de la pregunta.

Si la meta que se ha especificado en la pregunta puede ser satisfecha, y si no hay variables a considerar como en este ejemplo, entonces el sistema responde:

yes

Y la ejecución de la pregunta ha terminado.

Si en la pregunta se incluyen variables, entonces el valor final de cada variable es desplegado (excepto para las variables anónimas). Así la pregunta:

| ?- member(X,[a,b,c]).

Podría ser respondida como:

X = a

En este momento el intérprete estará esperando que el usuario introduzca un ENTER o un punto "." Seguido de un ENTER. Si se teclea solamente ENTER, termina la pregunta y el sistema responde "yes". Sin embargo, si se teclea ".", el sistema realiza un backtracking y busca soluciones alternativas. Si no hay soluciones alternativas entonces el sistema responde:

no

La salida para algunas preguntas como la que se muestra abajo, donde un número precede a un "_" significa un nombre generado por el sistema para una variable, por ejemplo:

```
| ?- member(X,[tom,dick,harry]).
X = tom ;
X = dick ;
X = harry ;
no
| ?- member(X,[a,b,f(Y,c)]), member(X,[f(b,Z),d]).
X = f(b,c),
Y = b,
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

Z = c

yes

| ?- member(X,[f(_),g]).

X = f(_52)

yes

| ?-

En el caso de los comandos, son similares a las preguntas excepto porque:

1. Las variables asociadas con un valor no se muestran cuando la ejecución del comando tiene éxito.
2. No existe la posibilidad de hacer backtracking para encontrar otras alternativas.

Los comandos inician con el símbolo ":-" . A nivel alto del intérprete simplemente basta con iniciar con el indicador "| ?- ". Cualquier salida pretendida debe ser programada explícitamente, por ejemplo, en el comando:

:- member(3,[1,2,3]), write(ok).

Dirije al sistema para que verifique si el número 3 pertenece a la lista [1,2,3], y a que redirija la salida hacia la variable "ok" si así es. La ejecución de un comando termina cuando todas las metas en el comando se han ejecutado de manera exitosa. No se muestran soluciones alternativas. Si no se encuentra solución entonces el sistema indica:

?

como una advertencia.

El uso principal de los comandos (en oposición al uso de las preguntas) es permitir a los archivos el contener las directivas que llaman a varios procedimientos, para los cuales el usuario no requiere salida impresa. En estos casos solo se requiere llamar a los procedimientos por su efecto, por ejemplo, si no quiere tener interacción con el intérprete durante la lectura de un archivo. Un ejemplo muy útil es el uso de una directiva que consulte una lista completa de archivos.

:- [bits, bobs, main, tests, data, junk].

Si un comando como este se encuentra contenido en el archivo 'myprog' entonces, teclear lo siguiente sería la forma más rápida para cargar el programa entero:

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

| ?- [myprog].

Cuando se interactúa al nivel más alto del intérprete de Prolog, la distinción entre comandos y preguntas es normalmente muy importante. A este nivel se teclean las preguntas en forma normal. Solo si se lee de un archivo, entonces para realizar varias metas se deben usar comandos, por ejemplo; una directiva en un archivo debe ser precedida por un ":-", de otra manera serían tratados como cláusulas.

5.18.1.2 Errores de sintaxis

Los errores de sintaxis se detectan durante la lectura. Cada cláusula o directiva, o en general cualquier término leido de un archivo o de la terminal, que provoca una falla, es desplegado en la terminal tan pronto como es leído por el sistema. Una marca indica el punto en la cadena de símbolos en donde al parser ha fallando en su análisis, por ejemplo:

member(X,X:L).

Implica que el parser muestre:

***** syntax error *****
member(X,X
***** here *** : L).**

Si el operador "." No ha sido declarado como operado infijo.

Notese que no se muestran comentarios que especifiquen el tipo de error. Si usted tiene duda acerca de qué cláusula contiene el error, puede usar el predicado:

listing/1

Para enlistar todas las cláusulas que han sido eficientemente leídas, por ejemplo:

| ?- listing(member).

5.18.1.3 Predicados no definidos.

El sistema puede ocasionalmente también darse cuenta de que se han llamado a predicados que no tienen cláusulas, el estado de esta facilidad de identificación puede ser:

- **'trace'**, que causa que este tipo de errores sean desplegados en la terminal y que el proceso de depuración muestre la última vez que ha ocurrido.

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

O,

- **'fail',** que causa que estos predicados fallen (este es el estado por default)

El predicado evaluable:

unknown(OldState,NewState)

unifica a OldState con el estado actual y coloca a State como Newstate. Este falla si los argumentos no son los apropiados. La depuración de predicados evaluables imprime el valor de este estado a lo largo de otras informaciones. Es importante notar que se lleva casi un 70% más de tiempo correr al intérprete en esta modalidad. Se espera que esta facilidad se pueda mejorar en el futuro.

5.18.1.4 Ejecución de programas e interrupción de éstos.

La ejecución de un programa se incia cuando se le da al intérprete una directiva en la cual se continenen llamadas a uno de los procedimientos del programa.

Sólo cuando la ejecución de una directiva se ha realizado por completo es cuando el intérprete va por otra directiva. Sin embargo, uno puede interrumpir la ejecución normal de ésta al presionar ^C mientras el sistema está corriendo. Esta interrupción tiene el efecto de suspender la ejecución y se despliega el siguiente mensaje:

function (H for help):

En este momento el intérprete acepta comando de una sola letra, que correspondan con ciertas acciones. Para ejecutar una acción simplemente se deberá teclear el carácter correpondiente (en minúsculas) seguido de un ENTER. Los posibles comandos son:

a aborta el comando actual tan pronto como sea posible

b detiene la ejecución (ver apartado 1.9)

c continuar la ejecución

e Salir de prolog y cerrar todos los archivos

g manejar posibilidad de recobrar aspectos ya borrados con anterioridad

h Lista todos los comandos disponibles

m Sale a nivel Monitorl (puede teclear Monitor o CONTINUE)

n desactiva el comando trace

t activa el comando trace

Notese que no es posible romper la ejecución directamente desde el código compilado, cada una de las opciones "a", "b" y "t" piden que el intérprete realice una acción, por lo que la acción se seguirá realizando hasta que el código compilado devuelva en mando al intérprete.

Note también que la opción ("a") no saca al usuario cuando esta en modo [user] es decir insertando cláusulas desde la terminal. Tiene que teclear ^Z para parar este proceso o teclear ("e") para poder salir de Prolog.

Si se trata de abortar un programa con ^C y accidentalmente se llegara al nivel Monitor, (quizá porque se apretó ^C demasiadas veces), deberá teclear CONTINUE para regresar a la interrupción ^C.

5.18.1.5 Salida del intérprete

Para salir del intérprete y regresar al nivel monitor, deberá teclear ^Z a nivel alto del intérprete, o llamar al procedimiento "halt" (pre-cargado por default), o usar el comando "e" (exit) seguido de una interrupción ^C. ^Z es diferentes de los otros dos métodos de salida, ya que imprime algunas estadísticas. Es posible regresar usando el comando del Monitor CONTINUE después de ^Z, si se arrepiente de haber querido salir.

Ejecuciones anidadas, Break y Abort.

El sistema prolog integra una manera de suspender la ejecución de un programa e introducir nuevas directivas o cambiar metas a nivel alto del intérprete. Esto se logra si se llama a un predicado evaluable Break o tecleando "b" despues de una interrupción con ^C (ver sección 1.7).

Esto causa que el intérprete suspenda la ejecución inmediatamente, a lo que el interprete contestará

[Break (level 1)]

Que quiere decir que se ha parado desde el nivel Break y que se pueden introducir preguntas de la misma manera en que se había hecho desde el nivel alto del intérprete.

El intérprete indica el nivel Break en el que el sistema se encuentra (por ejemplo, en Breaks anidados), imprimiendo el nivel de Break después de un yes o un no final a las preguntas. Por ejemplo, a nivel Break 2, se podría ver la respuesta así: uld look like:

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

| ?- true.

[2] yes | ?-

Un ^Z cierra la interrupción que se había realizado y se reaundan las tareas. Una ejecución suspendida se puede abortar usando la directiva:

| ?- abort.

5.18.1.6 Dentro de un break.

En este caso no se necesitaa un ^Z para cerrar el break; Todos los niveles break se descartarán y se llega automáticamente al nivel más alto del intérprete.

Guardado y restauración de estados de programas

Una vez que el programa ha sido leído, el intérprete tendrá disponible toda la información que sea necesaria para su ejecución, esta información se llama estado de programa.

El estado de un programa se puede grabar en disco para ejecuciones posteriores. Para guardar un programa en un archivo, realice la siguiente directiva:

| ?- save(file).

Este predicado se puede llamar en cualquier momento, por ejemplo, puede ser muy útil a fin de guardar los estados de un programa inmediatos.

Cuando el programa ha sido guardado en un archivo, la siguiente directiva recuperaría el estado de un programa:

| ?- restore(file).

Después de esta ejecución el programa entonces podrá ser regresado exactamente al mismo punto en que se dejó al momento de salvarlo la ´´ultima vez.

| ?- save(myprog), write('programa restaurado').

Así, al ejecutarse esta orden con el mensaje propio del usuario, se obtendrá el mensaje "programa restaurado", por ejemplo.

Notese que cuando una versión de Prolog nueva se instala, todos los programas grabados se vuelven obsoletos.

5.18.1.7 Inicialización

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

Cuando prolog inicia busca un archivo llamado 'prolog.bin' en la ruta default del usuario. Si se encuentra este archivo entonces se asume que el estado de un programa se ha restaurado. El efecto es el mismo que si se hubiera tecleado:

| ?- restore('prolog.bin').

Si no se encuentra este archivo, entonces se busca en forma similar, el archivo 'prolog.ini'. Si se encuentra este archivo, sería lo mismo que haber tecleado:

| ?- ['prolog.ini'].

La idea de este archivo es que el usuario pueda tener aquí los procedimientos y directivas que se usan usualmente de manera que no se tengan que teclear cada vez que se necesiten.

El archivo **'prolog.bin'** parece ser que es útil cuando se usa con el plsys(run(_,_)) que permite correr otros programas desde dentro de Prolog. Esta facilidad permite también regresar al mismo punto si se corrió prolog desde otros programas. En este caso, el predicado evaluable save/2 debe ser usado paara salvar el estado del programa dentro de 'prolog.bin'.

5.18.1.8 Entrada a Prolog

Cuando se entra a prolog directamente en una interacción con la terminal, se escribe autoimaticamente el archivo 'prolog.log' en modo append. Esto es, si ya existe el archivo, en el directorio del usuario, entonces la nueva información se añade, si no es así, y el archivo no existe, entonces el archivo se crea.

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

5.19 Análisis de casos: Programas en Prolog

5.19.1 PROGRAMA FAMILIA

El siguiente es un programa elaborado para TURBO PROLOG. En el se deben definir previamente los dominios, predeterminado si las constantes son símbolos o son números enteros. Asimismo se deben predeterminar los predicados que se utilizar n en el programa, con sus respectivos argumentos. Una vez realizado lo anterior se pasa a definir las clásulas para los HECHOS, Estos son predicados que establecen la base de datos con la cuál el Programa PROLOG ir a operar. Finalmente como otras cl sulas se definen las reglas, estableciendo una cabeza de la cláusula y un cuerpo. Las reglas determinan las búsquedas lógicas, que se realizan por medio de una búsqueda en la base de datos.

```
 domains
    persona = symbol
predicates
    male(persona)
    female(persona)
    parent(persona,persona)
    mother(persona,persona)
    father(persona,persona)
clauses
    male(juan).
    male(luis).
    female(rosa).
    female(marta).
    parent(juan,luis).
    parent(juan,marta).
    parent(rosa,luis).
    parent(rosa,marta).
```

```
mother(X,Y):-female(X),parent(X,Y).
father(X,Y):-male(X),parent(X,Y).
```

En el presente caso se tienen dos reglas, "mother",que establece en la cabeza que X es madre de Y, y en el cuerpo se define por medio de conjunciones cuales son los atributos que debe cumplir X para ser madre. En este caso X es madre de Y, si X es mujer y X es progenitor de Y. Lo mismo suceder con la regla "father"

5.19.2 Programa candiadatos

Este programa es elaborado para TURBO PROLOG, definiendo en los dominios a candidato como un símbolo, es decir una constante simbólica no numérica. Lo mismo sucede con partido. La diferencia se presenta en año, pues este tomo se define como un entero. Asimismo como requisito endógeno al TurboProlog, se definen predicados para cada una de la cláusulas que se utilizarán en la base de datos y de las reglas.

En breve, los dominios especifican si los tomos utilizados son o símbolos o si son enteros numéricos. Los predicados determinan que es lo que se va a usar si símbolos o enteros en cada caso de las cl sulas ya sean hechos o reglas. El anterior programa establece una base de datos de los candidatos a la presidencia de la Republica de Costa Rica entre los años 1958 y 1994. Se establecen cl sulas para cada uno de los ganadores y cada uno de los perdedores, estableciendo a su vez como argumentos del hecho el apellido del candidato en año del inicio del período presidencial, el a¤o de la finalización del período presidencial y el partido al que pertenecía. La parte lógica del programa se establece por medio de una serie de reglas que determinan las posobles preguntas que se pueden hacer al programa. Se establecen reglas para un ganador y luego perdedor, pra un perdedor y luego ganador, para un doble perdedor, para un perdedor del partido de gobierno, para un ganador del partido de gobierno, para determinar un antecesor y un sucesor as¡ como el ganador en la primer mitad de un año, así como un ganador para la segunda mitad del año.

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
domains
candidato=symbol
ano=integer
partido=symbol

predicates
ganador(candidato,ano,ano,partido)
perdedor(candidato,ano,ano,partido)
ganador_perdedor(candidato)
perdedor_ganador(candidato)
doble_perdedor(candidato)
gobierno_perdedor(symbol,ano)
gobierno_ganador(symbol,ano)
antecesor(candidato,candidato)
sucesor(candidato,candidato)
imitad(candidato,ano)
iimitad(candidato,ano)

clauses

ganador(echandi,1958,1962,republicano).
ganador(orlich,1962,1966,liberacion).
ganador(trejos,1966,1970,unificacion).
ganador(figueres,1970,1974,liberacion).
ganador(oduber,1974,1978,liberacion).
ganador(carazo,1978,1982,unidad).
ganador(monge,1982,1986,liberacion).
ganador(arias,1986,1990,liberacion).
ganador(calderonf,1990,1994,unidad).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
ganador(figuereso,1994,1998,liberacion).
perdedor(orlich,1958,1962,liberacion).
perdedor(calderong,1962,1966,republicano).
perdedor(oduber,1966,1970,liberacion).
perdedor(echandi,1970,1974,unificacion).
perdedor(monge,1978,1982,liberacion).
perdedor(calderonf,1982,1986,unidad).
perdedor(calderonf,1986,1990,unidad).
perdedor(castillo,1990,1994,liberacion).
perdedor(rodriguez,1994,1998,unidad).

ganador_perdedor(X) :-
        ganador(X,Y,_,W),
        perdedor(X,R,_,W),
        R>Y.
perdedor_ganador(X) :-
        perdedor(X,Y,_,W),
        perdedor(X,K,_,W),
        Y<>K.
doble_perdedor(X) :-
        perdedor(X,Y,Z,W),
        perdedor(X,K,L,W),
        Y<>K,
        Z<>L.
gobierno_perdedor(Q,R) :-
        ganador(_,Y,_,W),
        perdedor(Q,R,_,W),
        X=Y+4.
```

```
gobierno_ganador(Q,R) :-
                ganador(_,Y,_,W),
                ganador(Q,R,_,W),
                R=Y+4.
antecesor(X,Q) :-
                ganador(X,Y,Z,W),
                ganador(Q,R,S,T),
                R=Y+4,
                S=Z+4.
sucesor(Q,X) :-
          ganador(X,Y,Z,W),
          ganador(Q,R,S,T),
          R=Y+4,
          S=Z+4.
imitad(X,K) :-
         ganador(X,Y,Z,W),
         K>Y,
         K<=Z.
iimitad(X,K) :-
          ganador(X,Y,Z,W),
          K<Z,
          K>=Y.
```

5.19.3 Programa Cajas

En este programa se ponen a prueba técnicas de gran potencia en PROLOG, como lo es la construcción de estructuras complejas, manipulación de listas y sus componentes dentro de las estructuras, membresía a listas, asi se implementan una variedad de cl sulas con reglas recursivas, un instrumento de los m s potentes en PROLOG. El programa fue escrito para TURBOPROLOG, por lo que se establecieron los dominios para los tomos que se usan en el programa ya sean estos num,ricos o no numéricos asimismo se definieron los predicados para las distintas cláusulas ya sean estas reglas o predicados. El programa es un solucionador de problemas, Busca la administración de tres pilas de cajas. La pilas de cajas pueden ser configuradas como listas que contienen elementos, para este caso las cajas. Se tienen tres pilas, la a que contiene una caja roja, una verde y una negra de arriba a abajo, la pila b no tiene cajas y la pila c tiene solo una caja azul. El programa primero manipula las cajas dentro de pilas individuales, luego maneja las cajas y grupos de pilas como también manipula la composición de las pilas. La composición de las pilas es vertical y se utilizan las relaciones de arriba y abajo.

La elaboración del programa parte de establecer como tomos a las cajas que ser n manejadas como símbolos no numéricos. Inmediatamente después se definir n las listas de las cajas que junto con el nombre de la pila llegan a conformar las pilas propiamente dichas. Finalmente posible establecer el estado de las pilas por medio de la estructura que tiene como functor de estado. El programa implementa la regla de membresía "member" que sirve para determinar por un m,todo recursivo si un elemento es miembro de una lista determinada. Asimismo se implementan reglas de sublista: "sublist", asimismo de implementa la regla "last" que sirve para determinar el ultimo elemento de una lista. Dos reglas importantes son la de "prefix" y "suffix" que servirán para determinar el prefijo y el sufijo de una determinada lista. Son dos reglas recursivas que sirven para

recorrer las listas y determinar una cabeza o una cola de una lista. Su caracter recursivo hacen que las listas puedan ser recorridas de tal manera que cada vez que se determinan una lista cola, se puede desmembrar en otra cabeza y otra lista cola y as¡ sucesivamente hasta llegar a la lista vacía.

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

Cada pila se representa con una estructura: pila(NombrePila,ListaCAjas). Se definir n entonces las tres pilas en cláusulas PROLOG. El paso subsiguiente es definir la cláusula para una pila vacía, otra para una pila no vac¡a, otra para determinar cuando se tiene el tope de una pila.

Así pilaVacía tiene ,xito si la Pila tiene una lista vacía de cajas. PilaNoVac¡a en caso contrario. Tope tiene éxito cuando se encuentra a la caja de la pila buscada. Asimismo se tiene la regla "base" que tiene éxito cuando encuentra la caja que est en la base. Arriba tiene éxito cuando la caja buscada est arriba de otra caja previamente señalada, aunque no sean continuas. Lo mismo pero lo contrario sucede con debajo. Se puede señalar entonces que en medio eM cumple su cometido si una caja C1 est en medio de las cajas C2 y C3, aunque no sean continuas. También se definen justo arriba JA, justo en medio JEM y justo debajo, JD. El primero tiene éxito si la caja buscada, C1

est justo arriba de la caja C2 en la pila buscada. Justo en medio tiene éxito cuando una caja C1, est justo en medio de la caja C2 y la caja C3 en la pila buscada. El Justo Debajo, tiene éxito si la caja C1 est justo debajo de la caja C2 en la pila buscada.

El programa CAJAS, puede trabajar con m s de una pila a la vez. Para ello se utiliza la estructura con el functor estado, que consiste en una lista de pilas, es decir una lista de listas. A partir de la definición del estado, se establece una cláusula que da nombre a la pila dNP, que cumple su cometido cunado el nombre de la Pila señalado es el nombre de una de las pilas del estado. Asimismo la cláusula darPila, tiene éxito cuando la Pila es una de las pilas del Estado. La cláusula tope2 tienen éxito si en nombre pila es el nombre pila de uan pila de Estado y si su lista de cajas no esta vacía. Asimismo la cláusula base2 resulta exitosa si NP es el nombre-pila de una pila de estado y si en esa pila, la caja C1,esta en la base de esa pila.

La clásulas con los functores jA2,jD2,jEM2 son justo Arriba 2,justo Debajo2,justo En Medio2. jA2 es exitoso cuando NP es el nombre-pila de una pila de estado y si esa pila, la caja está justo C1 est justo arriba de la caja C2. Lo mismo pero alrevés sucede con jD2, sin NP es el nombre-pila de uan pila de estado y si en esa pila, la caja C1 está justo debajo de la caja C2. Con jEm2, se tiene ,xito si NP es el nombre-pila de una pila de

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

estado y si en esa pila la caja C1 est justo en medio de caja C2 y caja C3. Las clásulas con functores ar2, eM2, debajo2, representan para el estado el Arriba, el en Medio y el Debajo. ar2 tiene éxito si NP es el nombre-pila de uan pila de estado y si esa pila, la caja C1 est arriba de la caja C2, aunque no sean contiguas, debajo 2 tiene é,xito si NP es el nombre-pila de una pila de estado y si en esa pila la caja C1 está debajo de la caja C2,aunque no sean contiguas. eM2 tiene éxito si NP es el nombre-pila de una pila de estado y si en esa pila, la caja C1 está en medio de caja C2 y caja C3, aunque no están contiguas. El programa CAJAS también opera con cl sulas que transforman un estado. Para poder lograr esto se implementan un estado original y un nuevo estado o estado final. El progama tiene los functores pTope y qTope para poner y quitar tope respectivamente. ponerTope tiene ,xito si NP es un nombre-pila de uan pila de de un Estado, Caja no está en ninguna pila de este Estado1; además un Estado 2 E2 se obtiene del Estado1 E1 al poner Caja en el tope de NP dejando todo lo dem s igual. qTope tiene éxito si NP, es el nombre-pila de una pila E1, además E2 se obtiene de E1 al quitar la caja que está en el tope de NP dejando todo los demás igual. Asimismo se tiene los functores qPila y pPila para quitar pila y poner pila respectivamente. pPila tiene exito si NP es un nombre-pila que no aparece en E1(nuevo); además E2 se obtiene de E1 al quitar la caja que est en el tope de NP dejando todo lo demás igual. qPila tiene éxito si NP es un nombre-pila que aparece en E1; adem s E2 se obtiene de E1 al quitar la pila con¡ nombre-pila NP, dejando todo lo demás igual. Finalmente se establece la clasula moverT(mover Tope) que tiene ,xito si NP1 y NP2 son nombres-pila de dos pials distintas de E1; adem s E2 se obtiene de E1 al quitar el tope de

la pila con nombre NP1 y poner dicha caja como tope de la pila con nombre-pila NP2, dejando todo lo dem s igual. Para ello se establecen la cl sulas auxilares qTope y pTope, quitar y poner tope respectivamente.

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

Programa Cajas

```
domains
caja = symbol
listacajas = caja*
nombrepila = symbol
pila = pila(nombrepila,listacajas)
estado = pila*

predicates

estado(estado)
pilaVacia(pila)
pilaNovacia(pila)
tope(pila,caja)
base(pila,caja)
jA(pila,caja,caja)
jEM(pila,caja,caja,caja)
arriba(pila,caja,caja)
eM(pila,caja,caja,caja)
jD(pila,caja,caja)
debajo(pila,caja,caja)
last(caja,listacajas)
prefix(listacajas,listacajas)
suffix(listacajas,listacajas)
sublist(listacajas,listacajas)
pila(nombrepila,listacajas).

member(caja,listacajas).
member2(pila,estado).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
dNP(nombrepila,estado).
darPila(pila,estado).
tope2(nombrepila,caja,estado).
base2(nombrepila,caja,estado).
jA2(nombrepila,caja,caja,estado).
jD2(nombrepila,caja,caja,estado).
jEM2(nombrepila,caja,caja,estado).
ar2(nombrepila,caja,caja,estado).
debajo2(nombrepila,caja,caja,estado).
eM2(nombrepila,caja,caja,caja,estado).
cEP(caja,pila).
cEE(caja,estado).
pTope(caja,nombrepila,estado,estado).
qTope(caja,nombrepila,estado,estado).
pPila(nombrepila,estado,estado).
qPila(nombrepila,estado,estado).
moverT(nombrepila,nombrepila,estado,estado).

clauses

member(X,[X|_]).
member(X,[_|Y]):-member(X,Y).
member2(X,[X|_]).
member2(X,[_|Y]):-member2(X,Y).
last(X,[X]).
last(X,[_|Y]):-last(X,Y).
prefix([X|Xs],[X|Ys]):-prefix(Xs,Ys).
prefix([],Ys).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
suffix(Xs,Xs).
suffix(Xs,[Y|Ys]):-suffix(Xs,Ys).
sublist(Xs,Ys):-prefix(Xs,Ss),suffix(Ss,Ys).
pila(a,[roja,verde,negra]).
pila(b,[]).
pila(c,[azul]).
pilaVacia(pila(_,[])).
pilaNovacia(pila(_,[_|_])).
tope(pila(_,[X|_]),X).
base(pila(_,Ys),C):-last(X,Ys),X=C.
jA(pila(_,Ys),C,C2):-sublist([C,C2],Ys).
jEM(pila(_,Ys),C,C2,C3):-sublist([C2,C1,C3],Ys).
arriba(pila(_,Ys),C1,C2):-jA(pila(_,Ys),C1,C2).
arriba(pila(_,Ys),C1,C2):-jA(pila(_,Ys),C3,C2),arriba(pila(_,Ys),C1,C3).
eM(pila(_,Ys),C1,C2,C3):-jEM(pila(_,Ys),C1,C2,C3).
eM(pila(_,Ys),C1,C2,C3):-arriba(pila(_,Ys),C2,C1),debajo(pila(_,Ys),C3,C1).
jD(pila(_,Ys),C1,C2):-sublist([C2,C1],Ys).
debajo(pila(_,Ys),C1,C2):-jD(pila(_,Ys),C1,C2).
debajo(pila(_,Ys),C1,C2):-jD(pila(_,Ys),C3,C2),debajo(pila(_,Ys),C1,C3).

estado([pila(a,[roja,verde,negra]),pila(b,[ ]),pila(c,[azul])]).
dNP(NP,[pila(NP,_)|_]).
dNP(NP,[_|ListaPIlas]):-dNP(NP,ListaPIlas).
darPila(Pila,[Pila|_]).
darPila(Pila,[_|Ys]):-darPila(Pila,Ys).
tope2(NP,Caja,Estado):-member2(Pila,Estado),
                       Pila = pila(NP,_),
                       tope(Pila,Caja).
base2(NP,Caja,Estado):-member2(Pila,Estado),
```

```
                Pila = pila(NP,_),
                base(Pila,Caja).
jA2(NP,C1,C2,Estado):-member2(Pila,Estado),
          Pila = pila(NP,_),
          jA(Pila,C1,C2).
jD2(NP,C1,C2,Estado):-member2(Pila,Estado),
          Pila = pila(NP,_),
          jD(Pila,C1,C2).
jEM2(NP,C1,C2,Estado):-member2(Pila,Estado),
          Pila = pila(NP,_),
          jEM(Pila,C1,C2,C3).

ar2(NP,C1,C2,Estado):-member2(Pila,Estado),
          Pila = pila(NP,_),
          arriba(Pila,C1,C2).
debajo2(NP,C1,C2,Estado):-member2(Pila,Estado),
             Pila = pila(NP,_),
             debajo(Pila,C1,C2).
eM2(NP,C1,C2,C3,Estado):-member2(Pila,Estado),
                Pila = pila(NP,_),
                eM(Pila,C1,C2,C3).

cEP(Caja,Pila):-Pila=pila(NP,Ys),member(Caja,Ys).

cEE(Caja,Estado):- member2(Pila,Estado),
           cEP(Caja,Pila).

pTope(Caja,NP,[pila(NP,Xs)|Ys],[pila(NP,[Caja|Xs])|Ys]).
pTope(Caja,NP,[pila(NP2,Xs)|Ys],[pila(NP2,Xs)|Zs]):-
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
NP2<>NP,pTope(Caja,NP,Ys,Zs).

qTope(Caja,NP,[pila(NP,[Caja|Xs])|Ys],[pila(NP,Xs)|Ys]).
qTope(Caja,NP,[pila(NP2,Xs)|Ys],[pila(NP2,Xs)|Zs]):-
NP<>NP2,qTope(Caja,NP,Ys,Zs).

qPila(NP,[pila(NP,_)|Ys],Ys).
qPila(NP,[pila(NP2,X)|Ys],[pila(NP2,X)|Zs]):-
    NP<>NP2,qPila(NP,YS,Zs).

pPila(NP,Ys,[Pila|Ys]):-Pila=pila(NP,[ ]),
            not(member2(pila(NP,_),Ys)).

moverT(NP1,NP2,Estado,E3):-member2(pila(NP1,_),Estado),
              member2(pila(NP2,_),Estado),
              NP1<>NP2,
    qTope(Caja,NP1,Estado,E2),
    pTope(Caja,NP2,E2,E3).
```

5.19.4 Resolucionador de problemas proposicionales

Problemas lógicos basado en: "El enigma de Drácula de: Dr. Raymond Smulyan(Indiana University)

Este programa busca ser un resolucionador de problemas lógicos que involucran cálculos lógicos tanto el proposicional como el el de predicados. En ambos casos solo se intentan resolver problemas de lógica extensional. El problema lógico de base es tomado del libro de Raymond Smullyan: ***¨¿Cómo se llama este libro***?, donde viene un apartado dedicado al enigma de Drácula que plantea el siguiente problema:

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

En Transilvania aproximadamente la mitad de los habitantes eran humanos y la otra mitad eran vampiros. Los humanos y los vampiros son indistinguibles en su apariencia externa, pero los humanos dicen siempre la verdad, mientras que los vampiros siempre mienten. Lo que complica la situación enormemente es que la mitad de los habitantes de Transilvania est n totalmente locos y completamente engañados por lo que respecta a sus creencias -creen que todas las proposiciones verdaderas son falsas y que todas las proposiciones falsas son verdaderas. La otra mitad est¤a completamente cuerda y sabe que proposiciones son verdaderas y que proposiciones son falsas. As¡ pues, los habitantes de TRansilvania son de cuatro tipos:

1)humanos cuerdos;
2)humanos locos
3)vampiros cuerdos
4)vampiros locos

Todo lo que un humano cuerdo dice es verdadero; todo lo que un humano loco dice es falso y todo lo que un vampiro cuerdo dice es falso, y todo lo que un vampiro loco dice es verdadero. Se añade a esto la definición de formales e informales. Por definición un transilvano es formal si es humano cuerdo o vampiro loco y es informal si es humano loco o un vampiro cuerdo. La gente formal es la que dice enunciados verdaderos; la gente informal es aquella que hace enunciados falsos. Para poder desarrollar un resolucionador del problema lógico en concreto, se estableció una estructura para cada tipo de habitante y para cada estado posible de los habitantes. Es decir para humano, vampiro, cuerdo, y loco. Asimismo, dado que algunos de los problemas planteados tienen que ver con que Drácula está, vivo o muerto, se estableció una estructura para vivo y otra para muerto.
Para ir cubriendo otras posibilidades lógicas, se estableció estructura para determinar si dos individuos son del mismo tipo o de diferente tipo, es decir humano o vampiros y otras estructuras para determinar el estado es decir loco o cuerdo y otras estructuras para determinar la situación es decir si están vivos o muertos.

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

Determinar estructuras donde se planten todas las posibilidades lógicas de los habitantes es el mecanismo de base para encontrar las soluciones buscadas. Pero los procesos de resolución se realizan con el establecimiento de las reglas lógicas. Se implementan en clásulas PROLOG los conectores lógicos de Y , O. El Y se establece por un condicional, que establezca una conjunción en el cuerpo de la misma. El O es posible definirlo con base en la conjunción a través de un De Morgan.

Un problema que se presentó a la hora de encontrar un resolutcionador de problemas veritativo-funcionales es que en PROLOG, ninguna de las negaciones como el Not, o el No señala una falsedad que es una categoría semántica. Por consecuencia para cada conector lógico y para cada regla era imprescindible definir su respectiva negación. Por eso se define Y y No-Y, O y No-O. Asimismo pasa con las definiciones del condicional IF y el Bicondicional IFF que hay que definir sus negaciones. El programa revela que es necesario acudir a mecanismos sintácticos para definir la falsedad semántica, fundamental en los cálculos proposicional y de predicados. El No de PROLOG, no indica falsedad sino que no hay una relación que haga pareja con la buscada. Así el Not no es un mecanismo de negación, sino un mecanismo sintáctico que indica que si no se encuentra lo buscado entonces se falla y se inicia un retrorastreo. Debe tenerse en cuenta que del acertijo original, este resolutor de problemas se encarga solo de las preguntas de carácter extensional no de las de carácter intensional que involucran operadores epistémicos que por el momento no son manejables en PROLOG.

PROGRAMA: Vampiros-Humanos

```
nombre(N,hab(N,_,_,_)).

humano(hab(_,humano,_,_)).
vampiro(hab(_,vampiro,_,_)).
cuerdo(hab(_,_,cuerdo,_)).
loco(hab(_,_,loco,_)).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
vivo(hab(_,_,_,vivo)).
muerto(hab(_,_,_,muerto)).

mismotipo(hab(_,humano,_,_),hab(_,humano,_,_)).
mismotipo(hab(_,vampiro,_,_),hab(_,vampiro,_,_)).

diftipo(hab(_,humano,_,_),hab(_,vampiro,_,_)).
diftipo(hab(_,vampiro,_,_),hab(_,humano,_,_)).

mismoestado(hab(_,__,cuerdo,_),hab(_,_,cuerdo,_)).
mismoestado(hab(_,_,loco,_),hab(_,_,loco,_)).

difestado(hab(_,_,cuerdo,_),hab(_,_,loco,_)).
difestado(hab(_,_,loco,_),hab(_,_,cuerdo,_)).

mismasitua(hab(_,_,_,vivo),hab(_,_,_,vivo)).
mismasitua(hab(_,_,_,muerto),hab(_,_,muerto)).

difsitua(hab(_,_,_,vivo),hab(_,_,_,muerto)).
difsitua(hab(_,_,_,muerto),hab(_,_,_,vivo)).

y(P,Q):-P,Q.

/*INCLUSION DE UN !(CUT), PARA EVITAR BACKTRAKING)*/
o(P,Q):-!,member(R,[P,Q]),R.

no(y(P,Q)):-!,o(no(P),no(Q)).
no(o(P,Q)):-!,y(no(P),no(Q)).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
no(no(P)):-!,P.

no(humano(P)):-!,vampiro(P).
no(vampiro(P)):-!,humano(P).
no(loco(P)):-!,cuerdo(P).
no(cuerdo(P)):-!,loco(P).

formal(hab(_,humano,cuerdo,_)).
formal(hab(_,vampiro,loco,_)).
informal(hab(_,humano,loco,_)).
informal(hab(_,vampiro,cuerdo,_)).

no(formal(P)):-informal(P).
no(informal(P)):-formal(P).

dice(P,A):-vivo(P),formal(P),A.
dice(P,A):-vivo(P),informal(P),no(A).

no(dice(P,A)):-dice(P,no(A)).

no(vivo(A)):-muerto(A).
no(muerto(A)):-vivo(A).

if(P,Q):-o(no(P),Q).
iff(P,Q):-o(y(P,Q),y(no(P),no(Q))).
no(if(P,Q)):-no(o(P),Q).
no(iff(P,Q)):-no(o(y(P,Q),y(no(P),no(Q)))).
```

```
member(X,[X|_]).
member(X,[_|Xs]):-member(X,Xs).

/*  necesita TIPO
no(mismotipo(P1,P2)):-difftipo(P1,P2).
no(difftipo(P1,P2)):-mismotipo(P1,P2).
*/

/*NOTA: NO A = ~A; A y B = A & B; A o B = A # B; A implica B = A -> B;
 A equivalente B = A <-> B *, SINTAXIS PROLOG PARA SIMBOLOS LOGICOS */

modus_ponens(P,Q):-y(if(P,Q),P),Q.
modus_tollens(P,Q):-y(if(P,Q),no(Q)),no(P).
```

5.19.5 Programa de misioneros y caníbales

Este programa es un un resolucionador de problemas lógicos, acudiendo a ténicas de búsqueda propias de la Inteligencia Artificial clásica. En estas ténicas de búsqueda se acude a gráficos de espacio-estado. Los nodos del gr fico son los estados del problema. Una etapa existe entre los nodos si se da una regla de transición. Una regla de transición pude ser llamada también como una movida, que transforma un estado en el siguiente. Resolver problemas significa encontrar el camino desde un estado inicial hasta un estado solución deseado por medio de la aplicación de una secuencia de reglas de transición. Los movimientos son especificados por un predicado binario:

movida(Estado,Movida)

donde Movida es una movida aplicable al estado.

El predicado: actualiza(Estado,Movida,Estado1) encuentra el estado llegando a aplicar la movida: mueva a estado Estado. La validez de las posibles movidas es revisada por el predicado:

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

legal(Estado)

el cual revisa si el estado problema Estado satisface las restricciones del problema.

Historia: el programa debe tener una historia de los estados visitados para prevenir un enciclamiento (looping). La previsión que el enciclamiento no ocurra se hace revisando que el nuevo estado aparezca en la historia de los estados. La resolución de un problema parte de una Estructura, tal que el programador debe decidir como los estados ser representados y axiomatizar la movida, la actualización y los procedimientos legales.

El problema concreto a resolver fue el siguiente: Tres misioneros y tres caníbales se encuentran al lado izquierdo de un r¡o. Hay un pequeño bote para transportarlos a través del río, pero en el bote solo puede ir uno dos como máximo. Ellos desean cruzar el río. Si en alguna de las riveras del r¡o se encontraran m s misioneros que caníbales, los misioneros convertirían a los caníbales. SE debe encontrar pues una serie de transportes a través del r¡o sin exponer a ninguno de los caníbales a la conversión. El programa que anterior resuelve el problema correctamente, e inclusive da cuatro soluciones al mismo. Esto se realizó estableciendo una Estructura, donde se establece un Estado inicial,un Estado final, una Historia, las Movidas, la actualización de las movidas y la legalidad.

Posteriormente se desarrolla el programa, estableciendo por medio de estructuras y de reglas cada uno de los elementos citados anteriormente. Hay una estrutura para para estado final y para estado inicial, as¡ como se definen las reglas para movidas, actualizaci¢n y legalidad. Se a¤ade una serie de reglas especiales que establecen como llenar el bote.

```
/* ESTRUCTURA */
resolver_dfs(Estado,Historia,[0,0]):- estado_final(Estado).
resolver_dfs(Estado,Historia,[Movida|Movidas]):-
movida(Estado,Movida),
        actualiza(Estado,Movida,Estado1),
legal(Estado1),
        not(member(Estado1,Historia)),
resolver_dfs(Estado1,[Estado1|Historia],Movidas).
```

```
/*PRUEBA A LA ESTRUCTURA*/
prueba_dfs(Problema,Movidas):- estado_inicial(Problema,Estado),
resolver_dfs(Estado,[Estado],Movidas).
/* MISIONEROS-CANIBALES */
estado_inicial(mis_Can,mis_Can(izquierda,[3,3],[0,0])).
estado_final(mis_Can(derecha,[0,0],[3,3])).

movida(mis_Can(izquierda,I,D),Carga):-llenar(Carga,I).
movida(mis_Can(derecha,I,D),Carga):-llenar(Carga,D).
actualiza(mis_Can(B,I,D),Carga,mis_Can(B1,I1,D1)):-actualiza_bote
(B,B1),
      actualiza_rivera(Carga,B,I,D,I1,D1).
actualiza_bote(izquierda,derecha).
actualiza_bote(derecha,izquierda).
actualiza_rivera([M,C],izquierda,[M1,C1],[M2,C2],[M3,C3],[M4,C4]):-
      M3 is M1-M, C3 is C1-C, M4 is M2+M,C4 is C2+C.
actualiza_rivera([M,C],derecha,[M1,C1],[M2,C2],[M3,C3],[M4,C4]):-
      M3 is M1+M,C3 is C1+C , M4 is M2-M, C4 is C2-C.
legal(mis_Can(B,I,D)):-legal_rivera(I),legal_rivera(D).
legal_rivera(R):- R=[M,C],C>0, M=<C.
legal_rivera(R):- R=[M,C], C=0.
llenar([1,0],[M,_]):-M>=1.
llenar([2,0],[M,_]):-M>=2.
llenar([0,1],[_,C]):-C>=1.
llenar([0,2],[_,C]):-C>=2.
llenar([1,1],[M,C]):-C>=1,M>=1.
member(X,[X|_]).
 member(X,[_|Ys]):-member(X,Ys).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

5.19.6 Programa: Maridos celosos, Dudney 1917

Con base en las técnicas de búsqueda profunda propias ide la Inteligencia Artificial cl sica citados para el programa anterior se propuso un resolucionador de problemas lógicos de mayor complejidad. El problema lógico planteado es el siguiente:

Cinco maridos celosos de Dudeney,1917:

Durante cierta inundación, cinco parejas de esposos, se encontraron rodeado por agua, y para escapar de incomoda posición tenían un bote en el que cab¡an solo tres presonas al mismo tiempo. Cada marido era muy celoso y no enviar¡a a sus esposa para que estuviese en el bote o en la otra rivera con otro hombre y hombres a menos que el estuviera presente. Se debe encontrar un camino para que estos cinco hombres y cinco mujeres puedan cruzar y encontrarse seguros.

Para resolverlo, se planteo una estructura general con su Estado inicial, el final, su Historia, sus Movidas, su actualización y sus legalidades. Asimismo y de manera subsecuente se definieron con estructuras los estados inicial y final para las cinco parejas. Se establecieron las movidas y la legalidad de ellas. Se acudió a una regla recursiva no-determinística denominada "sel", que operaba seleccionando de manera heurística los miembros a ser tranportados en el bote, de lo que hab¡a que corroborar también la legalidad de las movidas. En la actualización de las riveras también se acudió al "sel" no determínistico. Este "sel" no determinístico no permitió que el resolucionador pudiera ser implementado en un tiempo real pues lo máximo con que se logró trabajar fueron tres parejas, ya con cuanto la o se saturaba la memoria o no reslovía. Para las resolución del problema fue necesario establecer clásulas de esposo como hechos y como reglas la definicióon de hombre, mujer y si había hombres o mujeres en un estado determinado. Sobre esto fue posible definir las reglas de legalidad de acuerdo a las restricciones propuestas por el problema.

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
/* ESTRUCTURA */
resolver_dfs(Estado,Historia,[]):- estado_final(Estado).
resolver_dfs(Estado,Historia,[Movida|Movidas]):-
movida(Estado,Movida),
                actualiza(Estado,Movida,Estado1),
               /* legal(Estado1),*/
                not(member(Estado1,Historia)),
              /* writeestado(Estado),
                writemovida(Movida),
                writeestado(Estado1),nl,*/
resolver_dfs(Estado1,[Estado1|Historia],Movidas).
/*PRUEBA A LA ESTRUCTURA*/
prueba_dfs(Problema,Movidas):- estado_inicial(Problema,Estado),
resolver_dfs(Estado,[Estado],Movidas).
/*ESTADOS*/
estado_inicial(celosos,celosos(izquierda,[a,aa,b,bb,c,cc,d,dd],[])).
estado_final(celosos(derecha,[],[a,aa,b,bb,c,cc,d,dd])).
/*MOVIDAS*/
movida(celosos(izquierda,I,D),Carga):-sel(I,_,Carga),legal_bote(C
arga).
  /* write("Carga:"),nl,
    write("    "),writeln(Carga).*/
movida(celosos(derecha,I,D),Carga):-sel(D,_,Carga),legal_bote(Car
ga).
  /* write("Carga:"),nl,
    write("    "),writeln(Carga).*/
/*ACTUALIZACION*/
actualiza(celosos(B,I,D),Carga,celosos(B1,I2,D2)):-actualiza_bote
(B,B1),
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
actualiza_rivera(Carga,B,I,D,I1,D1),
                                sort(I1,I2),
                                sort(D1,D2).
                               /* write("Movida:"),nl,
                                write("
C"),writeln(Carga),
                                write("  B"),write(B1),nl,
                                write("  I"),writeln(I2),
                                write("  D"),writeln(D2).*/
actualiza_bote(izquierda,derecha).
actualiza_bote(derecha,izquierda).
actualiza_rivera(Carga,izquierda,L,R,L1,R1):- sel(L,Carga,L1),
                                        append(Carga,R,R1),
                                        legal_rivera(L1),
                                        legal_rivera(R1).
actualiza_rivera(Carga,derecha,L,R,L1,R1):- sel(R,Carga,R1),
                                      append(Carga,L,L1),
                                      legal_rivera(L1),
                                      legal_rivera(R1).
/*REGLAS SUBYACENTES*/
/*
select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]):-select(X,Ys,Zs).
insert(X,Ys,Zs):-select(X,Zs,Ys).*/
member(X,[X|_]).
member(X,[_|Y]):-member(X,Y).
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]):-append(Xs,Ys,Zs).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
/*
selects(X,Ys):-select(X,Ys,Ys1),selects(Xs,Ys1).
selects([],Ys).
selects2([X|Xs],Ys,Zs):-select(X,Ys,Ys1),selects2(Xs,Ys,Zs).
selects2([],Ys,Zs).
insert(X,[],[X]).
insert(X,[Y|Ys],[Y|Zs]):-X>Y,insert(X,Ys,Zs).
insert(X,[Y|Ys],[Y|Ys]):-X=<Y. */
sel([X|Xs],[X|Ys],Zs):-sel(Xs,Ys,Zs).
sel([X|Xs],Ys,[X|Zs]):-sel(Xs,Ys,Zs).
sel([],[],[]).
sort(L,L1):-sort(L,L1,[],[cc,c,bb,b,aa,a]).
sort(L,L1,Acc,[M1|Ms]):-member(M1,L),!,sort(L,L1,[M1|Acc],Ms).
sort(L,L1,Acc,[M1|Ms]):-not(member(M1,L)),!,sort(L,L1,Acc,Ms).
sort(_,Acc,Acc,[]).
lenght([X|Xs],N):-lenght(Xs,N1),N=N1+1.
lenght([],0).
/*
writeln([X|Xs]):-write(X),writeln(Xs).
writeln([]):-nl.
writeL([X|Xs]):-write(X), write(" "), writeL(Xs).
writeL([]).
writeestado(celosos(Bote,RI,RD)):-write(Bote),write("  I: "),
   writeL(RI), write("  D: "), writeL(RD), nl.
writemovida(Movida):-writeL(Movida),nl. */
/*HECHOS Y CLASULAS DE BASE*/
esposo(a,aa).
esposo(b,bb).
esposo(c,cc).
```

```
esposo(d,dd).
esposo(e,ee).
hombre(X):-member(X,[a,b,c,d]).
mujer(X):-member(X,[aa,bb,cc,dd]).
hay_hombres(E):-hombre(X),member(X,E),!.
hay_mujeres(E):-mujer(X),member(X,E),!.
/*REGLAS DE LEGALIDAD*/
legal(celosos(_,I,D)):-legal_rivera(I),legal_rivera(D).
legal_rivera(E):-E=[],!.
legal_rivera(E):-hay_mujeres(E),not(hay_hombres(E)),!.
legal_rivera(E):-hay_hombres(E),not(hay_mujeres(E)),!.
legal_rivera(E):-not(ilegal(E)).
legal_bote(B):-lenght(B,N),N=<3,N>=1,!,legal_rivera(B).
ilegal(E):-member(X,E),mujer(X),esposo(Y,X),not(member(Y,E)).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

5.19.7 Programa Eliza

El programa ELIZA creado por Wizenbaum (1965), considerado uno de los programas dela Inteligencia Rtificial clásica, generaba un ambiente por medio del cual se podía entablar una conversación seria sobre cualquier tema, de hecho se tomaba prestadas y manipùlaba las oraciones que mediante un teclado proporcionaba el sr humano. El usuario digita una oración, y Eliza responde con una pregunta o comentario adecuado. En un sentido real, obviamente Eliza no entiende, sino que responde a una entrada (input) específica por medio del reconocimiento de patrones de plabras y respondiendo por un correspondiente patrón de respuesta. Para hacer más creíble estos patrones de respuesta, ELIZA adopta una actitud de psiquiatra, que cuestiona aquellas expresiones que no reconoce.

Ejemplo de conversación con ELIZA:

I am unhappy

How long have you been unhappy

Six months. Can you help me.

What makes you think I help you.

You remind me of my father and brother.

Can you tell me more about father.

I like teasing father.

Does anyone else in youre family like teasing father.

No, only me.

Please go on.

El corazón de ELIZA es un procedimiento para acoplar (matching) la oración de entrada dentro de un patrón. El resultado de este acoplamiento es la aplicación de otro patrón que determina la respuesta del programa. Una pareja de patrones puede ser considerado como una pareja de estímulo-respuesta, donde la entrada se acopla contra el estímulo, y la salida es generada desde la respuesta.

Una típica pareja de estímulo-respuesta (sacado del ejemplo anterior):

I am (oración) **How long have you been (oración)?**

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

Usando esta pareja, la respuesta de le programa a la oración de entrada: “I am unhappy”, tendr como pregunta: “How long have you been unhappy?”. La oración puede ser descrita como una estructura vacía a ser completada.

El programa aquí presentado, es una versión simplificada del programa original ELIZA. Este implementa el siguiente algoritmo:

- **Leer la oración de entrada**
- **Mientras la entrada no sea bye**
    - ✓ **Escoja la pareja estímulo-respuesta**
    - ✓ **Acople la entrada con el estímulo**
    - ✓ **Generae una salida desde la respuesta y el acople anterior**
    - ✓ **Exprese la respuesta**
    - ✓ **Lea la próxima entrada**

A continuación se presenta el programa:

PROGRAMA ELIZA

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]):-append(Xs,Ys,Zs).
lookup(Key,[(Key,Value)|Dictionary],Value).
lookup(Key,[(Key1,Value1)|Dictionary],Value):-
        Key\=Key1,lookup(Key,Dictionary,Value).
        readwordlist(Ws):-
        get0(C),readwordlist(C,Ws).
readwordlist(C,[W|Ws]):-
        wordchar(C),
        readword(C,W,C1),
        readwordlist(C1,Ws).
readwordlist(C,Ws):-
        fillchar(C),
        get0(C1),
        readwordlist(C1,Ws).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
readwordlist(C,[]):-
        endwordchar(C).
readword(C,W,C1):-
        wordchars(C,Cs,C1),
        name(W,Cs).
wordchars(C,[C|Cs],C0):-
        wordchar(C),!,
        get0(C1),
        wordchars(C1,Cs,C0).
wordchars(C,[],C):-
        not(wordchar(C)).
wordchar(C):-97=<C,C=<122.
wordchar(C):-65=<C,C=<90.
wordchar(95).
wordchar(39).
wordchar(63).
fillchar(32).
endwordchar(46).
endwordchar(33).
eliza:-readwordlist(Input),eliza(Input),!.
eliza([hello]):-
        writeL("Hello.How are you?"),nl,
        readwordlist(Input),
        eliza1(Input).
eliza1(Input):-
        pattern(Stimulus,Response),
        match(Stimulus,Dictionary,Input),
        match(Response,Dictionary,Output),
        transfor(Output,Output1),
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
        reply(Output1),
        readwordlist(Input1),!,
        eliza1(Input1).
eliza1([bye]):-
        writeL("Goodbye. Come again soon").
match([N|Pattern],Dictionary,Target):-
        integer(N),lookup(N,Dictionary,LeftTarget),
        append(LeftTarget,RightTarget,Target),
        match(Pattern,Dictionary,RightTarget).
match([Word|Pattern],Dictionary,[Word|Target]):-
        atom(Word),match(Pattern,Dictionary,Target).
match([],Dictionary,[]).
pattern([hello],[hello,how,are,you,?]).
pattern([1,am,crazy],[1,are,a,patient]).
pattern([X,1,Y,2,?],[1,X,Y,2,very,much]):-
        verb([X|_],verb(X),[]),
        verb([Y|_],verb(Y),[]),good(Y).
pattern([X,1,Y,2,?],[why,X,1,Y,2]):-
        verb([X|_],verb(X),[]),
        verb([Y|_],verb(Y),[]),
        bad(Y).
pattern([i,like,1],[does,anyone,else,in,your,family,like,1,?]).
pattern([1,feel,X],[do,1,often,feel,X]):-
        adje([X|_],adje(X),[]).
pattern([my,X,2],[can,you,tell,me,more,about,X,?]):-
        important(X).
pattern([no],[please,go,on]).
pattern([why,are,psychiatrists,so,1,?],[what,do,you,mean,by,1,?]).
pattern([1,means,2],[perhaps,it,is,you,that,are,2]).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
pattern([not,so,1,as,my,X],[does,your,X,agree,?]).
pattern([i,want,to,X,1],[try,to,X,1,quickly]):-
        verb([X|_],verb(X),[]),
        good(X).
pattern([1,want,to,X,2],[do,1,really,want,to,X,2]):-
        verb([X|_],verb(X),[]),
        bad(X).
pattern([go,to,1],[please,accompany,me]).
important(father).
important(mother).
important(son).
important(sister).
important(brother).
important(daughter).
good(fly).
good(tell).
good(know).
good(find).
good(help).
good(like).
bad(die).
bad(run).
bad(kill).
bad(hate).
bad(hit).
transfor(Output,Output1):-
        sentence(Output,Sent,[]),
        sustnp(i,you,Sent,Sent1),
        sustvp(you,me,Sent1,Sent2),
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
        conv(Sent2,Output1).
transfor(Output,Output):-not(sentence(Output,Sent,[])).
sustnp(X,Y,sent(Sent),sent(Sent1)):-sustnp(X,Y,Sent,Sent1).
sustnp(X,Y,ora(NP,VP),ora(NP1,VP)):-
        functor(NP,np,_),
        functor(VP,vp,_),
        sust(X,Y,NP,NP1).
sustnp(X,Y,preg1(VP,Oracion),preg1(VP,Oracion1)):-
        functor(Oracion,ora,_),
        sustnp(X,Y,Oracion,Oracion1).
sustnp(X,Y,preg2(VP,NP),preg2(VP,NP)).
/* sust(X,Y,Est1,Est2: sustituye el pronombre X por el
        pronombre Y en Est1 y da Est2 */
sust(X,Y,np(det(Det),noun(Noun)),np(det(Det),noun(Noun))).
sust(X,Y,det(Det),det(Det)).
sust(X,Y,np(noun(Noun)),np(noun(Noun))):-
        sust(X,Y,noun(Noun),noun(Noun)).
sust(X,Y,noun(Noun),noun(Noun)).
sust(X,Y,np(pron(Pronoun)),np(pron(Pronoun1))):-
        sust(X,Y,pron(Pronoun),pron(Pronoun1)).
sust(X,Y,pron(Z),pron(Z)):-
        X\=Z,Y\=Z.
sust(X,Y,pron(Z),pron(Y)):-
        X=Z.
sust(X,Y,pron(Z),pron(X)):-
        Y=Z.
sust(X,Y,np(Noun,AP),np(Noun,AP)):-
        functor(Noun,noun,_),
        functor(AP,ap,_).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
sust(X,Y,np(pron(Pronoun),AP),np(pron(Pronoun1),AP)):-
        functor(AP,ap,_),
        sust(X,Y,pron(Pronoun),pron(Pronoun1)).
sust(X,Y,np(pron(Pronoun),Modif),np(pron(Pronoun1),Modif)):-
        functor(Modif,modif,_),
        sust(X,Y,pron(Pronoun),pron(Pronoun1)).
sust(X,Y,np(ap(Modif,AP)),np(ap(Modif,AP))):-
        sust(X,Y,ap(Modif,AP),ap(Modif,AP)).
sust(X,Y,np(ap(AP)),np(ap(AP))):-
        sust(X,Y,ap(AP),ap(AP)).
sust(X,Y,np(NP,AP),np(NP,AP)):-
        functor(NP,np,2),
        functor(AP,ap,_).
sust(X,Y,ap(adje(Adjective)),ap(adje(Adjective))):-
        sust(X,Y,adje(Adjective),adje(Adjective)).
sust(X,Y,adje(Adjective),adje(Adjective)).
sust(X,Y,ap(Modif,adje(Adjective)),ap(Modif,adje(Adjective))):-
        functor(Modif,modif,_).
        sustvp(X,Y,sent(Sent),sent(Sent1)):-
        sustvp(X,Y,Sent,Sent1).
sustvp(X,Y,ora(NP,VP),ora(NP,VP1)):-
        functor(NP,np,_),
        functor(VP,vp,_),
        sust(X,Y,VP,VP1).
sustvp(X,Y,preg1(VP,Oracion),preg1(VP,Oracion1)):-
        functor(Oracion,ora,_),
        sustvp(X,Y,Oracion,Oracion1).
sustvp(X,Y,preg2(VP,NP),preg2(VP,NP1)):-
        functor(NP,np,_),
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
        sust(X,Y,NP,NP1).
sust(X,Y,vp(VP,NP),vp(VP,NP1)):-
        functor(VP,vp,_),
        functor(NP,np,_),
        sust(X,Y,NP,NP1).
sust(X,Y,vp(verb(Verb),NP),vp(verb(Verb),NP1)):-
        functor(NP,np,_),
        sust(X,Y,NP,NP1).
sust(X,Y,vp(verb(Verb)),vp(verb(Verb))):-
        sust(X,Y,verb(Verb),verb(Verb)).
sust(X,Y,verb(Verb),verb(Verb)).
sust(X,Y,vp(verb(Verb),VP),vp(verb(Verb),VP1)):-
        functor(VP,vp,_),
        sust(X,Y,VP,VP1).
sust(X,Y,vp(prep(Prep),VP),vp(prep(Prep),VP1)):-
        functor(VP,vp,_),
        sust(X,Y,VP,VP1).
sust(X,Y,vp(word(Word),VP),vp(word(Word),VP)):-
        functor(Vp,vp,_).
sust(X,Y,vp(verb(Verb),Modif),vp(verb(Verb),Modif)):-
        functor(Modif,modif,_).
sust(X,Y,vp(Modif,VP),vp(Modif,VP1)):-
        functor(Modif,modif,_),
        functor(VP,vp,_),
        sust(X,Y,VP,VP1).
sust(X,Y,prep(Prep),prep(Prep)).
sust(X,Y,word(Word),word(Word)).
sust(X,Y,modif(adverb(Adverb)),modif(adverb(Adverb))):-
        sust(X,Y,adverb(Adverb)).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
sust(X,Y,adverb(Adverb),adverb(Adverb)).
sust(X,Y,modif(adverb(Adverb),adverb(Adverb)),modif(adverb(Adverb),
adverb(Adverb))).
conv(sent(Oracion),Output):-
        functor(Oracion,ora,_),
        conv(Oracion,Output).
conv(sent(Preg1),Output):-
        functor(Preg1,preg1,_),
        conv(Preg1,Output),!.
conv(sent(Preg2),Output):-
        functor(Preg2,preg2,_),
        conv(Preg2,Output).
conv(ora(NP,VP),L3):-
        functor(NP,np,_),
        conv(NP,L1),
        functor(VP,vp,_),
        conv(VP,L2),
        append(L1,L2,L3).
conv(preg1(VP,Oracion),L3):-
        functor(VP,vp,_),
        conv(VP,L1),
        functor(Oracion,ora,_),
        conv(Oracion,L2),
        append(L1,L2,L3).
conv(preg2(VP,NP),L3):-
        functor(VP,vp,_),
        conv(VP,L1),
        functor(NP,np,_),
        conv(NP,L2),
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
        append(L1,L2,L3).
conv(np(det(Det),noun(Noun)),L3):-
        conv(det(Det),L1),
        conv(noun(Noun),L2),
        append(L1,L2,L3).
conv(det(Det),[Det]).
conv(np(noun(Noun)),L1):-
        conv(noun(Noun),L1).
conv(noun(Noun),[Noun]).
conv(np(pron(Pronoun)),[Pronoun]):-
        conv(pron(Pronoun),[Pronoun]).
conv(pron(Pronoun),[Pronoun]).
conv(np(ap(Modif,AP)),L3):-
        conv(ap(Modif,AP),L3).
conv(np(ap(AP)),L3):-
        conv(ap(AP),L3).
conv(np(NP,AP),L3):-
        functor(NP,np,2),
        conv(NP,L1),
        functor(Ap,ap,_),
        conv(AP,L2),
        append(L1,L2,L3).
conv(np(noun(Noun),AP),L3):-
        conv(noun(Noun),L1),
        functor(AP,ap,_),
        conv(AP,L2),
        append(L1,L2,L3).
conv(np(pron(Pronoun),Modif),L3):-
        conv(pron(Pronoun),L1),
```

```
        functor(Modif,modif,_),
        conv(Modif,L2),
        append(L1,L2,L3).
conv(np(pron(Pronoun),AP),L3):-
        conv(pron(Pronoun),L1),
        functor(AP,ap,_),
        conv(AP,L2),
        append(L1,L2,L3).
conv(ap(adje(Adjective)),L1):-
        conv(adje(Adjective),L1).
conv(adje(Adjective),[Adjective]).
conv(ap(Modif,adje(Adjective)),L3):-
        functor(Modif,modif,_),
        conv(Modif,L1),
        conv(adje(Adjective),L2),
        append(L1,L2,L3).
conv(vp(verb(Verb),NP),L3):-
        conv(verb(Verb),L1),
        functor(NP,np,_),
        conv(NP,L2),
        append(L1,L2,L3).
conv(vp(verb(Verb)),L1):-
        conv(verb(Verb),L1).
conv(verb(Verb),[Verb]).
conv(vp(verb(Verb),VP),L3):-
        functor(VP,vp,_),
        conv(verb(Verb),L1),
        conv(VP,L2),
        append(L1,L2,L3).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
conv(vp(prep(Prep),VP),L3):-
        conv(prep(Prep),L1),
        functor(VP,vp,_),
        conv(VP,L2),
        append(L1,L2,L3).
conv(vp(word(Word),verb(Verb)),L3):-
        conv(word(Word),L1),
        conv(verb(Verb),L2),
        append(L1,L2,L3).
conv(vp(word(Word),VP),L3):-
        conv(word(Word),L1),
        functor(VP,vp,_),
        conv(VP,L2),
        append(L1,L2,L3).
conv(vp(verb(Verb),Modif),L3):-
        conv(verb(Verb),L1),
        functor(Modif,modif,_),
        conv(Modif,L2),
        append(L1,L2,L3).
conv(vp(Modif,VP),L3):-
        functor(Modif,modif,_),
        conv(Modif,L1),
        functor(VP,vp,_),
        conv(VP,L2),
        append(L1,L2,L3).
conv(prep(Prep),[Prep]).
conv(word(Word),[Word]).
conv(modif(adverb(Adverb)),L1):-
        conv(adverb(Adverb),L1).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
conv(adverb(Adverb),[Adverb]).
conv(modif(adverb(Adverb),adverb(Adverb1)),L3):-
        conv(adverb(Adverb),L1),
        conv(adverb(Adverb1),L2),
        append(L1,L2,L3).
reply([Head|Tail]):-write(Head),write(' '),reply(Tail).
reply([]):-nl.
writeL([A|L]):-put(A),writeL(L).
writeL([]).
sentence(S0,sent(Oracion),S):-
        oracion(S0,Oracion,S).
sentence(S0,sent(Preg1),S):-
        pregunta1(S0,Preg1,S),!.
sentence(S0,sent(Preg2),S):-
        pregunta2(S0,Preg2,S),!.
oracion(S0,ora(NP,VP),S):-
        nounphrase(S0,NP,S1),
        verbphrase(S1,VP,S).
pregunta1(S0,preg1(VP,Oracion),S):-
        verbphrase(S0,VP,S1),
        oracion(S1,Oracion,S).
pregunta2(S0,preg2(VP,NP),S):-
        verbphrase(S0,VP,S1),
        nounphrase(S1,NP,S).
nounphrase(S0,np(Det,Noun),S):-
        determiner(S0,Det,S1),
        noun(S1,Noun,S).
nounphrase(S0,np(Noun),S):-
        noun(S0,Noun,S).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
nounphrase(S0,np(Pronoun),S):-
        pronoun(S0,Pronoun,S).
nounphrase(S0,np(Noun,AP),S):-
        noun(S0,Noun,S1),
        adjephrase(S1,AP,S).
nounphrase(S0,np(Pronoun,AP),S):-
        pronoun(S0,Pronoun,S1),
        adjephrase(S1,AP,S).
nounphrase(S0,np(NP,AP),S):-
        functor(NP,nounphrase,_),
        nounphrase(S0,NP,S1),
        functor(AP,adjephrase,_),
        adjephrase(S1,AP,S).
nounphrase(S0,np(AP),S):-
        adjephrase(S0,AP,S).
nounphrase(S0,np(Pronoun,Modif),S):-
        pronoun(S0,Pronoun,S1),
        modif(S1,Modif,S).
adjephrase(S0,ap(Adjective),S):-
        adje(S0,Adjective,S).
adjephrase(S0,ap(Modif,Adjective),S):-
        modif(S0,Modif,S1),
        adje(S1,Adjective,S).
verbphrase(S0,vp(Verb),S):-
        verb(S0,Verb,S).
verbphrase(S0,vp(Verb,VP),S):-
        verb(S0,Verb,S1),
        verbphrase(S1,VP,S).
verbphrase(S0,vp(Prep,VP),S):-
```

```
        prep(S0,Prep,S1),
        verbphrase(S1,VP,S).
verbphrase(S0,vp(Verb,NP),S):-
        verb(S0,Verb,S1),
        nounphrase(S1,NP,S).
verbphrase(S0,vp(Word,Verb),S):-
        word(S0,Word,S1),
        verb(S1,Verb,S).
verbphrase(S0,vp(Word,VP),S):-
        word(S0,Word,S1),
        verbphrase(S1,VP,S).
verbphrase(S0,vp(Verb,Modif),S):-
        verb(S0,Verb,S1),
        modif(S1,Modif,S).
/*verbphrase(S0,vp(VP,Modif),S):-
        functor(VP,vp,2),
        arg(1,vp(prep(Prep),VP),Prep),
        verbphrase(S0,VP,S1),
        modif(S1,Modif,S).*/
verbphrase(S0,vp(Modif,VP),S):-
        modif(S0,Modif,S1),
        verbphrase(S1,VP,S).
modif(S0,modif(Adverb),S):-
        adverb(S0,Adverb,S).
modif(S0,modif(Adverb,Adverb1),S):-
        adverb(S0,Adverb,S1),
        adverb(S1,Adverb1,S).

determiner([the|S],det(the),S).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
determiner([a|S],det(a),S).
noun([psychiatrists|S],noun(psychiatrists),S).
noun([psychiatrist|S],noun(psychiatrist),S).
noun([family|S],noun(family),S).
noun([something|S],noun(something),S).
noun([patient|S],noun(patient),S).
pronoun([i|S],pron(i),S).
pronoun([you|S],pron(you),S).
pronoun([me|S],pron(me),S).
verb([am|S],verb(am),S).
verb([are|S],verb(are),S).
verb([want|S],verb(want),S).
verb([tell|S],verb(tell),S).
verb([kill|S],verb(kill),S).
verb([do|S],verb(do),S).
verb([does|S],verb(does),S).
verb([try|S],verb(try),S).
verb([feel|S],verb(feel),S).
verb([can|S],verb(can),S).
verb([help|S],verb(help),S).
verb([like|S],verb(like),S).
verb([hate|S],verb(hate),S).
verb([run|S],verb(run),S).
prep([to|S],prep(to),S).
adje([silly|S],adje(silly),S).
adje([stupid|S],adje(stupid),S).
adje([unhappy|S],adje(unhappy),S).
adje([crazy|S],adje(crazy),S).
adverb([so|S],adverb(so),S).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
adverb([really|S],adverb(really),S).
adverb([quickly|S],adverb(quickly),S).
adverb([often|S],adverb(often),S).
adverb([very|S],adverb(very),S).
adverb([much|S],adverb(much),S).
word([why|S],word(why),S).
word([what|S],word(what),S).
word([how|S],word(how),S).
word([please|S],word(please),S).
se(1, sent(preg2(vp(verb(try),vp(prep(to),vp(verb(tell)))),np(pron(me))))).
se(2, preg2(vp(verb(try),vp(prep(to),vp(verb(tell)))),np(pron(me)))).
se(3, vp(verb(try),vp(prep(to),vp(verb(tell)))) ).
se(4, verb(try) ).
se(5, vp(prep(to),vp(verb(tell))) ).
se(6, prep(to) ).
se(7, vp(verb(tell)) ).
se(8, verb(tell) ).
se(9, np(pron(me)) ).
st(a, [do,you,want,to,kill,me]).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

5.19.8 Bibliotecas Prolog

```
/*APPEND*/
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]):-append(Xs,Ys,Zs).
adjacent(X,Ys,Zs):-append(As,[X,Y|Ys],Zs).
 member(X,Ys):-append(As,[X|Xs],Ys).
```

```
/*LIST*/
 list([]).
 list(X,[X|Xs]):-list(Xs).
```

```
/*FINDING THE LENGHT OF A LIST*/
/*lenght(Xs,N):-N is the length of the list Xs*/
lenght([X|Xs],N):-lenght(Xs,N1),N=N1+1.
lenght([],0)
member(X,[X|_]).
member(X,[_|Y]):-member(X,Y).
```

```
/*PREFIX*/
prefix([],Ys).
prefix([X|Xs],[X|Ys]):-prefix(Xs,Ys).
```

```
/*NAIVE REVERSE*/
reverse1([],[]).
reverse1([X|Xs],Zs):-reverse1(Xs,Ys),append(Ys,[X],Zs).
```

```
/*REVERSE ACCUMULATE*/
reverse2(Xs,Ys):-reverse2(Xs,[],Ys).
reverse2([X|Xs],Acc,Ys):-reverse2(Xs,[X|Acc],Ys).
reverse2([],Ys,Ys).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

```
/*Select*/
select(X,[X|Xs],Xs).
 select(X,[Y|Ys],[Y|Zs]):-select(X,Ys,Zs).
```

```
/*INSERT*/
insert(X,Ys,Zs):-select(X,Zs,Ys).
```

```
/*TESTING FOR A SUBSET*/
/* selects(Xs,Ys):- The lists is a subset of the list Ys*/
selects([X|Xs],Ys):-select(X,Ys,Ys1),selects(Xs,Ys1).
selects([],Ys).
select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]):-select(X,Ys,Zs).
```

```
/*INSERTION SORT*/
/*sotr(Xs,Ys):-Thelist Ys is an ordered permutation ofthe list*/
sort([X|Xs],Ys):-sort(Xs,Zs),inser(X,Zs,Ys).
sort([],[]).
insert(X,[],[X]).
insert(X,[Y|Ys],[Y|Zs]):-X>Y,insert(X,Ys,Zs).
insert(X,[Y|Ys],[X,Y|Ys]):-X=<Y.
```

```
/*SUBLIST*/
prefix([],Ys).
prefix([X|Xs],[Y|Ys]):-prefix(Xs,Ys).
sublist(Xs,Ys):-prefix(Xs,Ys).
sublist(Xs,[Y|Ys]):-sublist(Xs,Ys).
```

```
/*SUFFIX*/
suffix(Xs,Xs).
suffix(Xs,[Y|Ys]):-suffix(Xs,Ys).
```


RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

5.19.9 Bibliografía:

- Aikins, Janice; Ballantyne, Michael; Bledsoe, W. W.; Doyle, Jon; Moore, Robert C.; Pattis, Richard; Rosenschein, Stanley J.(1982) *Automatic Deduction*. The Handbook of Artificial Intelligence, vol. III, editado por Paul R. Cohen y Edward A. Feigenbaum. Addison-Wesley.
- Bellman, R.E. (1978). An Introduction to Artificial Intelligence: Can Computers Think?. Boyd & Frazer Publishing Company, San Francisco.
- Boden,Margaret(1977).Inteligencia artificial y hombre natural. Tecnos: Madrid. 1984.
- Boden,M.(1988).Computer Models Of Mind. Cambridge, University Press: New York.
- Boolos,George & Richard Jefrey (1974). Computability and Logic. Cambridge University Press: Cambridge, UK.
- Bundy, A. 1983. *The Computer Modelling of Mathematical Reasoning*, Academic Press.
- Chang, C.; Lee, R.C.(1973). *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.
- Charniak,E. & McDermott,D.(1985).Introduction to Artificial Intelligence. Addison-Wesles Reading: Mass.
- Charniak, E., Riesbeck, C., McDermott, D. & Meehan,J.(1987). Artificial Intelligence Programing. Lawrence Erlbaum Associates: Potomac,Maryland. 2nd edition.
- Cloksin, W. F. & C.S. Mellish (1981/84). Programing in Prolog. Springer-Verlag: Germany.
- Davies,R. & D. B. Lenat(1982).Knowledge-Based Systems in Artificial Intelligence. McGraw-Hill: International Book Company: USA
- Davis,Nartin(1958).Computability & Unsolvability. McGraw-Hill: New York.
- Duffy, David(1991).*Principles of Automated Theorem Proving*. John Wiley and Sons.
- Elithorn,A. y D.Jones(1973).Artificial and Human Thinking. Elsevier Scientific Publishing Company: Amsterdam

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

- Genesereth,M.R. y N. Nilson(1987).Logical Foundations of Artificial Intelligence. Morgaan Kaufmann Publishers, Inc.: Los Altos, CA.
- Feingenbaum, E.A. & J. Feldman (1963). Computers and Thought. McGraw-Hill Book Company: USA.
- Freund, M.(1995). Lógica Epistémica. En: Alchourrón, C.E. *Lógica.* Madrid: Editorial Trotta.
- Gallier, J.H. (1986) *Logic for Computer Science*, Harper and Row.
- Gardner, Martin(1983) *Logic Machines and Diagrams*. Segunda edición, Harvester Press.
- Gutiérrez,Claudio y M. Castro(1990). La Sociedad Computarizada. EUNED: San José.
- Gutiérrez,Claudio y M. Castro(1992). Informática y Sociedad. EUNED: San José.
- Gutiérrez,Claudio(1993). Epistemología e Informática.Guía de Estudio. EUNED: San José.
- Gutiérrez,Claudio y M.Castro(1993). Epistemología e Informática. Antología. EUNED: San José.
- Hopcroft, J.E. and Ullman, J.D. (1979). *Introduction to Authomata Theory Languages, and Computation*. Addison Wesley.
- Kneale, W. y M. Kneale.(1972). *El Desarrollo de la Lógica.* Madrid: Editorial Tecnos.
- Kowalski, Robert. 1979. *Logic for Problem Solving*. North-Holland.
- Lloyd, J. W. (1984) *Foundations of Logic Programming*. Springer-Verlag.
- Odifreddi, P. (ed). (1990). *Logic and Computer Science*, Academic Press.
- Rodríguez R., R.J.(1994).Epistemología e Inteligencia Artificial: Es posible una epistemología androide. En: Repertorio Científico. Vol.2.No.1/Enero-Abril,1994. EUNED.
- Russel,S.J. y P.Norving(1995).Artificial Inteligence. Pretince Hall: New Yersey.
- Shapiro,S.(1979).Techniques of Artificial Intelligence. Techniques of Artificial Intelligence. D. Von Nostrand Company: New York.

RODOLFO J. RORIGUEZ R.
E-mail: rodolfor@cariari.ucr.ac.cr

- Shapiro, S. C. & D. Eckroth (1987). Encyclopedia of Artificial Intelligence. J. Wiley: New York.
- Shoham,Yoav(1990).Non momotoning Reasoning and Causation, en: Cognitive Science: 14: 213-252.
- Sterling,L. y E.Shapiro(1986).The Art of Prolog. Advanced Programing Techniques. The MIT Press:CAmbridge:Mass.
- Shoham,Yoav(1988).Chronological Ignorance: Experiments in Nonmonotonic Temporal Reasoning, en: Artificial Intelligence, 36: 279-331.
- Stachniak, Zbigniew. (1996). *Resolution Proof Systems: An Algebraic Theory*. Kluwer.
- Sterling,L. & E. Shapiro(1986). The Art of Prolog. Advanced Programing Techniques. The MIT Press: Cambridge,Mass.
- Turing,A.M.(1950).Maquinaria computadora e inteligencia. En: Anderson, A.R.(1964).
- Von Newuman,J.(1958).The Computer and the Brain. Yale University Press.: New Haven,Connecticut.
- Watson,M.(1991).Common LISP Modules. Artificial Intelligence in the Era of Neural and Chaos Theroy. Springer Verlag: New York.
- Winston,P.H.(1992).Artificial Intelligence. Addison-Wesley,Reading
- Wisnton,P.H. y K.A. Prendergast(1984).The A.I. Business. The Commercial Uses of Artificial Intelligence. The MIT Press.
- Wos, Larry. (1998). "Programs that Offer Fast, Flawless, Logical Reasoning". *Communications of the ACM*, Junio, vol. 41, no. 6, pp. 87-95.
- Wos, Larry; Overbeek, Ross; Lusk, Ewing; Boyle, Jim.(1984). *Automated Reasoning: Introduction and Applications*. Prentice Hall. Segunda edición, McGraw-Hill, 1992.