UNIVERSIDAD NACIONAL AUTONOMA
DE MEXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
A C A T L A N

DEL DESARROLLO DE SISTEMAS MANEJADORES DE BASE DE DATOS A TRAVES DE LA LOGICA

T E S I S

QUE PARA OBTENER EL TITULO DE
LICENCIADO EN MATEMATICAS APLICADAS Y COMPUTACION
P R E S E N T A :
JOSE DE JESUS GONZALEZ ORIHUELA

ASESOR: FM SERGIO CHAPA VERGARA

MEXICO, D. F. 1992

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

Contenido

Agradecimiento x

Introducción xi

1 Lógica y Programación Lógica 1

1.1 Lógica Simbólica 2

1.1.1 Lógica proposicional 2

1.1.2 Lógica de Primer Orden 5

1.2 Unificación y Resolución 10

1.2.1 Principio de Resolución para la Lógica Proposicional 10

1.2.2 Unificación 10

1.2.3 Principio de Resolución para la Lógica de Primer Orden 13

1.3 Deducción 15

1.3.1 Semánticas: modelo e interpretación 15

1.3.2 Sintaxis: Teoría de Primer Orden 15

1.4 Programación Lógica 17

1.4.1 Constantes 18
1.4.2 Objetos compuestos 19
1.4.3 Variables 20
1.4.4 Procedimientos 21
1.4.5 Cláusulas 23
1.5 Estructura de Prolog 24

2 Lógica y Modelo de Base de Datos Relacional 29

2.1 Modelo Relacional 30
2.1.1 Esquema relacional 30
2.1.2 Algebra relacional 31
2.1.3 Dependencias funcionales 34
2.1.4 Normalización 36
2.1.5 Restricciones de Integridad 37
2.2 Lenguaje relacional. 38
2.2.1 Cálculo relacional de tuplas 38
2.2.2 Cálculo relacional de dominios 42
2.3 Aplicación de la Lógica a las Bases de Datos 46
2.3.1 Vista de sintaxis y semántica de Lógica y el Lenguaje Relacional 51
2.3.2 Lógica y Restricciones de Integridad 52
2.4 Deducción y Base de Datos Deductiva 53

3 Programación Lógica y Base de Datos Relacional 58

3.1 Base de Datos explícita en Prolog 59
3.2 Negación como no probabilidad 60
3.3 Plural distributivo y colectivo 61
3.4 Tratamiento de expresiones de conjuntos 61

3.5 Necesidad de más de dos valores 62

4 Lenguaje Natural 64

4.1 Gramáticas. 64

4.2 Redes de Transición Aumentadas (ATN) 69

4.3 Análisis Gramatical 71

4.3.1 Gramáticas Semánticas 72

4.3.2 Gramáticas de Casos 72

4.3.3 Dependencia Conceptual 73

5 Sistema de Base de Datos basado en Lógica FGH 76

5.1 Reglas gramaticales en Prolog 77

5.2 Dependencia Contextual 81

5.3 Semántica 86

5.3.1 Arboles de Análisis Gramatical (Parse Trees) 86

5.3.2 Significado en Lenguaje Natural 90

5.3.3 Consulta a Bases de Datos 95

5.3.4 Abstracción de Datos 98

5.4 Juntando las piezas 100

6 Conclusiones 102

Anexo 105

Bibliografía 119

Agradecimiento

Debido al irregular paso que mantenía al cursar las materias de la carrera, no fue sino un año después de lo establecido que cursé las materias de Seminario de Tesis eligiendo a un profesor que, aunque enigmático, era la persona indicada para dirigir la tesis con el tipo de tema que había elegido: el reconocimiento de lenguaje natural.

Este profesor me fue dirigiendo por un camino que recorrí a ciegas, cual fantasma que deambula sin saber lo que quiere. No me indicaba paso por paso lo que debía hacer, sino que solo marcaba los objetivos y dejaba que yo me las arreglara por mi mismo para conseguirlos. Me dió una gran lección: lo mejor que se puede hacer por una persona es dejarla sola para que descubra el potencial que tiene guardado. Entonces comprendí el significado del título de la obra de teatro "Cada quién su vida" de don Luis G. Basurto.

Ahora que veo finalizada esta tesis en que inesperadamente aprendí temas como la Lógica y la teoría de las Bases de Datos, vislumbro sombras de lo que conforma la vida misma: la muerte señala el último día en que se aprende, y aún entonces seré un ignorante. Más que a nadie agradezco y dedico este trabajo al Dr. Sergio Víctor Chapa Vergara por su interesada indiferencia. Gracias.

Introducción

El conocimiento que un ser humano "retiene en su memoria" se codifica en algún lugar de su sistema nervioso. Posiblemente se almacena en redes neuronales que implican conexiones sinápticas de conductividades electroquímicas variables; una especie de circuito programado cuya estructura y función pueden modificarse a través de la experiencia. No se conoce la fisiología exacta de ello. Sin embargo, estudios han demostrado que, durante los últimos 30,000 años ha habido poco cambio en la memoria y poder mental en los seres humanos. No obstante, durante este tiempo ha habido un marcado incremento en la habilidad de la humanodad para almacenar y procesar conocimiento. Esto se debe en gran medida al progreso de técnicas para la comunicación de conocimiento, junto con el desarrollo de métodos para el procesamiento mecánico del conocimiento.

En las primeras etapas de la evolución social de la humanidad, se empleaban gestos y habla para comunica conocimiento. Posteriormente se aumentó el medio transitorio del habla con formas de representación de conocimiento más perdurables, incluyendo pinturas, jeroglificos y la palabra escrita. Paralelamente a este crecimiento de técnicas para la comunicación, la humanidad fue desarrollando métodos para la manipulación rutinaria del conocimiento, a fin de aumentar su habilidad para pensar. Por ejemplo, se desarrollaron sistemas y notaciones matemáticas con objeto de facilitar el cálculo numérico. Se desarrollaron lógicas formales para facilitar el razonamiento, y se crearon bibliotecas y sistemas de archivos para facilitar el manejo de grandes volúmenes de conocimiento. En épocas más recientes hemos visto enormes avances gracias a la invención y empleo de sistemas mecánicos, electrónicos y ópticos para la comunicación, almacenamiento y procesamiento del conocimiento. Por ejemplo, los

sistemas manejadores de bases de datos son capaces de permitir a muchos usuarios el acceso a grandes volúmenes de hechos con formato uniforme.

Aunque la lógica ha sido usada como una herramienta para el diseño de las computadoras y los programas de las mismas casi desde sus orígenes, el uso de la lógica directamente como lenguaje de descripción de procedimientos - conocido como *programación lógica* - es muy reciente. La programación lógica implantada como sistema computarizado es una idea que surgió a principios de la década de los setentas y se le conoce con el nombre de PROLOG. Los pioneros de esta idea fueron Robert Kowalski en la Universidad de Edimburgo [KOWA72] [KOWA74] y Alain Colmerauer en la Universidad de Marsella [COLM72]. La popularidad que actualmente posee el lenguaje se debe principalmente a la eficiente implantación hecha por David Warren en Edimburgo [WARR77].

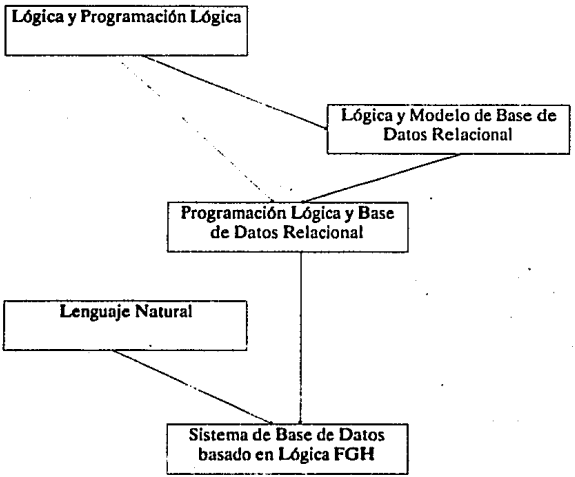
Como es enfatizado por E. F. Codd [CODD82], los estudios teóricos de las bases de datos constituyen la base fundamental para el desarrollo de sistemas manejadores de bases de datos (DBMS) homogéneos y sólidos, que ofrezcan capacidades sofisticadas para el manejo de hechos. Un estudio comprensivo de los muchos problemas que existen en las bases de datos requiere de una formalización precisa de tal modo que sean posibles análisis detallados y que se obtengan soluciones satisfactorias. La mayoría de los estudios formales de bases de datos vigentes en la actualidad utilizan el modelo relacional de datos introducido por Codd [CODD70], y utiliza una teoría de base de datos propia [MAIE83] [ULLM82], o usa otras teorías formales (como la lógica matemática) como su estructura.

El uso de la lógica para la representación y manipulación del conocimiento encuentra sus inicios en el trabajo de C. Green [GREE69]. Su obra fue la base de varios estudios que llevaron a los llamados sistemas de consulta-respuesta, los cuales tienen principalmente una gran manipulación deductiva sobre un pequeño conjunto de hechos, por lo que requerían de un sistema de inferencia basado en lógica. Técnicas similares han sido adaptadas a las bases de datos para manejar grandes conjuntos de hechos, información negativa, consultas abiertas, y otros temas propios de las bases de datos. Estas técnicas han generado lo que se conoce como bases de datos deductivas. Claro, el uso de la lógica para el estudio de las bases de datos no está restringido a proveer de capacidades deductivas a las bases de datos; en el trabajo pionero de Kuhns [KUHN67] usa la lógica en bases de datos convencionales para caracterizar respuestas a consultas.

La presente tesis pretende ilustrar algunas de las ventajas que ofrece el mezclar sistemas de lógica matemática con modelos matemáticos de base de datos y aplicar el resultado al desarrollo de sistemas manejadores de bases de datos. Formalizando el objetivo de este trabajo es:

Desarrollar un sistema de consulta de base de datos, con ayuda de la Lógica, que traduzca enunciados en lenguaje natural y que deduzca las respuestas como una interfase máquina-usuario que facilite la extracción de información de bases de datos.

Esta tesis se encuentra dividida en 5 capítulos, cuya organización se encuentra descrita en el siguiente esquema:



Durante el siglo pasado apareció una nueva corriente en el pensamiento lógico: la lógica matemática. Desde entonces su desarrollo ha sido nutrido por gran cantidad de matemáticos y estudiosos de diferentes corrientes que desean aplicarla a otras áreas del conocimiento científico. Esta tesis comenta algunos de estos aspectos. En el capítulo 1 comienzo presentando dos sistemas clásicos de la lógica matemática: el cálculo proposicional y el cálculo de predicados. Después describo tres requisitos de la programación lógica: unificación, resolución y deducción. Finalmente presento la

programación lógica como sistema formal ilustrándolo con la implantación en computadora de un subconjunto de ésta: el lenguaje de programación Prolog. Este es el núcleo de la teoría que sustenta este trabajo.

Es necesario conocer también los formalismos que rigen en el campo de las Bases de Datos, el capítulo 2 inicia con una presentación del modelo relacional de bases de datos y del lenguaje relacional. Habiendo presentado las bases, termino este capítulo mostrando cómo puede emplearse la lógica para formalizar varios aspectos de la tecnología de base de datos.

El capítulo 3 contiene una discusión de las características que implican el uso de la programación lógica en el desarrollo de las bases de datos y los sistemas que las manejan.

Para demostrar con un ejemplo las aseveraciones hechas hasta ahora, se puso como objetivo de este trabajo construir un pequeño sistema manejador de base de datos que demostrara alguna de las ventajas que ofrece el enfoque aquí expuesto, eligiendose un sistema de consulta en lenguaje natural, para ello incluyo un capítulo complementario, el 4, que nos muestra el panorama actual en materia de tecnologías de reconocimiento de lenguaje natural.

El capítulo 5, corazón de esta tesis, describe los pasos seguidos para la construcción de FGH: un sistema manejador de base de datos basado en lógica con interface en lenguaje natural. Elegí este nombre para bautizar al sistema debido a la coincidencia de que estas tres letras se encuentran juntas en el alfabeto castellano y también en los teclados de máquinas de escribir y computadoras tipo QWERTY.

Además de las conclusiones (que por error al registrar la tesis fueron catalogadas como un capítulo más), finalizo con un anexo que contiene los listados de FGH en el lenguaje de programación Prolog y las referencias bibliográficas.

Capítulo 1

Lógica y Programación Lógica

Podemos decir que el *lenguaje* es un sistema de signos y reglas por medio de los cuales el hombre elabora, expresa y comunica sus pensamientos. Un metalenguaje es, coloquialmente hablando, un lenguaje que describe a un lenguaje (lenguaje objeto). Mejor aún, es un sistema de signos y reglas utilizado para hablar de otro lenguaje.

Formalizando un poco, un lenguaje es un par ordenado (Σ, G) donde Σ es el alfabeto ó conjunto de símbolos y G es la gramática ó conjunto de reglas que determinan la forma adecuada en que debemos de combinar los símbolos para lograr expresar ideas.

La Lógica es la ciencia que estudia la estructura del conocimiento intelectual, es decir , que, prescindiendo de su contenido, se ocupa únicamente de su forma, su estructura (podemos decir que la Lógica es un metalenguaje). Desde la época de los griegos con Aristóteles se ha trabajado en la lógica, conocida ésta como *tradicional*. En el siglo XIX se produjo un cambio de orientación que condujo a la lógica contemporánea, llamada indistintamente *lógica simbólica* o *lógica matemática*. El primer nombre pone de relieve el hecho de que utilice un lenguaje artificial, constituido por símbolos que representan estructuras formales, y el segundo expresa su estrecha relación con las matemáticas, ya que surgió de los avances de ella (especialmente del álgebra). En la lógica contemporánea se estudian las conexiones entre los enunciados, que es lo que se llama lógica de enunciados o lógica proposicional, en la cual se introducen símbolos cuantificadores, lo cual hace que se le conozca como lógica cuantificacional o de predicados. El punto de partida del estudio lógico es siempre un análisis del lenguaje en el cual se hallan contenidos los conocimientos. Ahora bien, la lógica es, a su vez, como todo conocimiento, un nuevo lenguaje, cuyo estudio corresponde al metalenguaje conocido como la metalógica.

Un *programa lógico* es un conjunto de axiomas, o reglas, que definen relaciones entre objetos. Una computación de un programa lógico es una deducción de consecuencias del programa. Un programa define un conjunto de consecuencias, el cual es su significado. Hablaremos sobre la deducción en la sección 1.2.

El algoritmo de unificación y el principio de resolución de Robinson [ROBI65] son los dos elementos principales que colaboraron en la implantación de la programación lógica en computadoras, por lo que los abordaremos en la sección 1.3.

Prolog es una implantación en computadora de muchos elementos (no todos) de la programación lógica. Por lo tanto, la programación lógica y el Prolog son equivalentes, pero no son lo mismo. Estos dos conceptos serán abordados con amplitud en la sección 1.4.

1.1 Lógica Simbólica

Hablaremos aquí de los dos sistemas lógicos que usaremos a lo largo de este trabajo, la lógica proposicional y la lógica de predicados, o de primer orden.

1.1.1 Lógica proposicional

El principal interés de la lógica proposicional son los enunciados declarativos que puedan ser verdaderos o falsos, pero no ambos. Estos enunciados son llamados proposiciones. Ejemplos de proposiciones pueden ser "La tarde es tibia", "José es carpintero", "El oxígeno es un elemento químico". El valor "verdadero" o "falso" que es asignado a una proposición es llamado el valor de verdad de la proposición. Por convención - debido a la inicial de cada palabra en inglés ("true" para "verdadero" y "false" para "falso")-, definimos a estos valores como "**T**" para el verdadero y "**F**" para el falso. Es más, convenientemente podemos usar un símbolo en mayúscula o una cadena de símbolos en mayúsculas para denotar una proposición. Por ejemplo, podemos denotar las proposiciones anteriores del siguiente modo:

P: La tarde es tibia

Q: José es carpintero

R: El oxígeno es un elemento químico

Los símbolos, tales como *P*, *Q* y *R* que son usados para denotar proposiciones, son llamados fórmulas atómicas o átomos. De las proposiciones podemos construir proposiciones compuestas usando conectores lógicos. Dentro de la lógica proposicional se dispone de 5 conectores:

~ (no),

∧ (y),

∨ (o),

- $\Rightarrow$ (si...entonces), y
- $\Leftrightarrow$ (si y sólo si).

Una expresión que representa una proposición, como P, o una proposición compuesta, como $((P \wedge Q) \Rightarrow (\neg R))$, es llamada una fórmula bien formada (**fbf**). Las fórmulas bien formadas son definidas recursivamente como sigue:

1.- Un átomo es una fórmula.

2.- Si G es una fórmula, entonces $(\neg G)$ es una fórmula.

3.- Si G y H son fórmulas, entonces $(G \wedge H)$, $(G \vee H)$, $(G \Rightarrow H)$ y $(G \Leftrightarrow H)$ son fórmulas.

4.- Todas las fórmulas son generadas aplicando las tres reglas anteriores.

Los conectores lógicos poseen -para evitar el uso de paréntesis- la siguiente jerarquía de operación: primero el operador unario $\neg$,y después los operadores binarios $\vee$,$\wedge$, $\Rightarrow$, $\Leftrightarrow$.

Sean G y H dos fórmulas. Entonces los valores de verdad de las fórmulas $(\neg G)$, $(G \wedge H)$, $(G \vee H)$,$(G \Rightarrow H)$, $(G \Leftrightarrow H)$ están relacionados a los valores de verdad de G y H del siguiente modo:

- $\neg G$ es verdadera cuando G es falsa, y es falsa cuando G es verdadera. $\neg G$ es llamada la negación de G.
- $(G \wedge H)$ es verdadera si G y H son verdaderas; de otro modo $(G \wedge H)$ es falsa. $(G \wedge H)$ es llamada la conjunción de G y H.
- $(G \vee H)$ es verdadera si por lo menos G o H son verdaderas; de otro modo $(G \vee H)$ es falsa. $(G \vee H)$ es llamada la disyunción de G y H.
- $(G \Rightarrow H)$ es falsa si G es verdadera y H es falsa; de otro modo $(G \Rightarrow H)$ es verdadera. $(G \Rightarrow H)$ se lee como "Si G, entonces H", o "G implica H".
- $(G \Leftrightarrow H)$ es verdadera cuando G y H tengan el mismo valor de verdad; de otra manera $(G \Leftrightarrow H)$ es falsa. $(G \Leftrightarrow H)$ es leída como "G si y sólo si H".

Estas relaciones son representadas convenientemente en la siguiente tabla :

Tabla 1.1 Valores de verdad para fórmulas bien formadas.

G	H	$\neg G$	$(G \wedge H)$	$(G \vee H)$	$(G \Rightarrow H)$	$(G \Leftrightarrow H)$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

La tabla que despliega los valores de verdad de una fórmula G para todos valores de verdad posibles de los átomos que componen G es llamada la tabla de verdad de G.

Dada una fórmula proposicional G, sean $A_1, A_2, \ldots, A_n$ los átomos que aparecen en la fórmula G. Entonces una interpretación de G es una asignación de valores de verdad para $A_1, \ldots, A_n$ donde a cada A_i se le asigna **T** o **F**, pero no ambos.

Si existen n diferentes átomos en una fórmula, entonces habrán 2^n interpretaciones diferentes para la fórmula.

Cuando una fórmula G es verdadera bajo todas sus interpretaciones es llamada fórmula válida o *tautología*. Por el contrario, cuando la fórmula es falsa bajo todas las interpretaciones es llamada fórmula inconsistente o *contradicción*. Una fórmula se dice ser *consistente* si y sólo si existe por lo menos una interpretación bajo la cual la fórmula sea verdadera.

Si una fórmula F es verdadera bajo la interpretación I, entonces decimos que I satisface a F, o que F es satisfecha por I. También I es llamada un *modelo* de F.

A menudo es necesario transformar una fórmula de una forma a otra, especialmente a una más sencilla de manipular. Esto es llevado a cabo reemplazando una fórmula en una fórmula dada por otra fórmula "equivalente" a ésta y repitiendo este proceso hasta que se obtiene la forma deseada. Decimos que dos fórmulas F y G son equivalentes (o que F es equivalente a G), denotado como $F = G$, si y sólo si los valores de verdad de F y G son los mismos bajo todas las interpretaciones de F y G. Es necesario un conjunto de fórmulas equivalentes para poder llegar a formas estándar ó normales, dicho conjunto se muestra en la Tabla 1.2, donde E, G, y H son fórmulas. Cada par de

Tabla 1.2 Leyes del álgebra booleana.

$F \Rightarrow G = (F \Rightarrow G) \wedge (G \Rightarrow F)$	
$F \Rightarrow G = \sim F \vee G$	
$F \vee G = G \vee F$	$F \wedge G = G \wedge F$
$(F \vee G) \vee H = F \vee (G \vee H)$	$(F \wedge G) \wedge H = F \wedge (G \wedge H)$
$F \vee (G \wedge H) = (F \vee G) \wedge (F \vee H)$	$F \wedge (G \vee H) = (F \wedge G) \vee (F \wedge H)$
$F \vee \mathbf{F} = F$	$F \wedge \mathbf{T} = F$
$F \vee \mathbf{T} = \mathbf{T}$	$F \wedge \mathbf{F} = \mathbf{F}$
$F \vee \sim F = \mathbf{T}$	$F \wedge \sim F = \mathbf{F}$
$\sim(\sim F) = F$	
$\sim(F \vee G) = \sim F \wedge \sim G$	$\sim(F \wedge G) = \sim F \vee \sim G$

fórmulas equivalentes es un axioma o una ley que puede ser verificado mediante tablas de verdad.

Una literal es un átomo o la negación de un átomo. Se dice que una fórmula F se encuentra en una forma normal conjuntiva si y sólo si F tiene la forma $F = F1 \wedge ... \wedge Fn$, $n \geq 1$, donde cada F1,...,Fn es una disyunción de literales.

Una fórmula F dícese estar en una forma normal disyuntiva si y sólo si F tiene la forma $F = F1 \vee ... \vee Fn$, $n \geq 1$, donde cada F1,...,Fn es una conjunción de literales.

Si un enunciado o proposición es derivado de otros enunciados antecedentes, decimos que el enunciado obtenido es una consecuencia lógica de sus antecedentes.

Dadas las fórmulas F1, F2,..., Fn y la fórmula G, se dice que G es la consecuencia lógica de F1, ...,Fn si y sólo si para cualquier interpretación I en la cual $F1 \wedge F2 \wedge ... \wedge Fn$ es verdadera, G también es verdadera. F1, F2, ..., Fn son llamadas axiomas (o postulados, premisas) de G.

Teorema 1.1.: Dada una serie de fórmulas F1, ..., Fn y una fórmula G, G es una consecuencia lógica de F1, ..., Fn si y sólo si la fórmula $(F1 \wedge ... \wedge Fn) \Rightarrow G$ es válida.

Teorema 1.2.: Dadas las fórmulas F1,...,Fn y la fórmula G, G es la consecuencia lógica de F1,...,Fn si y sólo si la fórmula $(F1 \wedge ... \wedge Fn \wedge \sim G)$ es inconsistente.

Estos dos teoremas son muy importantes. Muestran que probar que una fórmula particular es una consecuencia lógica de un conjunto infinito de fórmulas es equivalente a probar que cierta fórmula relacionada es válida o inconsistente. Si G es la consecuencia lógica de F1,...,Fn, la fórmula $((F1 \wedge ... \wedge Fn) \Rightarrow G)$ es llamada un teorema, y G es llamada también la conclusión del teorema. En matemáticas como en otras áreas de la ciencia, muchos problemas pueden ser formulados como problemas de demostración de teoremas. De ahí la utilidad práctica de la esta teoría.

1.1.2 Lógica de Primer Orden

En la lógica proposicional, los átomos son tratados como unidades sencillas, su estructura y composición son suprimidas. Sin embargo, existen ideas que no pueden ser representadas de esta sencilla manera. Por ejemplo, consideremos la siguiente deducción de enunciados clásica:

Todos los hombres son mortales.

Sócrates es un hombre.

Ya que Sócrates es un hombre, es mortal.

Este razonamiento es intuitivamente correcto. Sin embargo, si denotamos

P: Todos los hombres son mortales,

Q: Sócrates es un hombre,

R: Sócrates es mortal,

entonces *R* no es una consecuencia lógica de *P* y *Q* dentro de la lógica proposicional. Esto es debido a que las estructuras de *P*, *Q* y *R* no son usadas en la lógica proposicional. Ahora veremos la lógica de primer orden, el cual tiene tres nociones lógicas más (llamadas términos, predicados y cuantificadores) que la lógica proposicional. Podemos usar los siguientes cuatro tipos de símbolos para construir un átomo:

- Símbolos individuales o constantes: son usualmente nombres de objetos, como José, María y 7.
- Símbolos de variables: como *x*, *y*, *z*.
- Símbolos de funciones: por ejemplo **padre** y **más**.
- Símbolos de predicados: como **mayor** y **ama_a**.

"Predicado" en su estricto significado gramatical de aquello que una sentencia dice acerca de su sujeto, es menos amplio que "función proposicional de una variable o propiedad", puesto que, para un predicado, el nombre omitido en el esqueleto de la sentencia ha de ser el sujeto de la sentencia. El esqueleto de una sentencia representa una relación binaria o una función proposicional de dos variables, por ejemplo "*x* ama a *y*".

Cualquier función o símbolo de predicado tiene un número específico de argumentos. Si un símbolo de función *f* tiene *n* argumentos, *f* es llamada un símbolo de función de *n* lugares. Por un símbolo de predicado (de *n* argumentos) entendemos, de acuerdo con ello, un símbolo de predicado de *n* lugares, donde *n* puede ser 0, dando lugar a una proposición, o 1, dando lugar a un predicado en sentido tradicional o una propiedad, o mayor que 1, dando lugar a una relación n-aria. Debido a lo anterior el cálculo de predicados también es conocido como cálculo funcional. Una función es un mapeo de una lista de constantes a una constante. Esto nos lleva a definir un término recursivamente como sigue:

1.- Una constante es un término.

2.- Una variable es un término.

3.- Si *f* es un símbolo de función de *n* lugares, y t_1, ..., t_n son términos, entonces $f(t_1, ..., t_n)$ es un término.

4.- Todos los términos son generados con las tres reglas anteriores.

Un predicado es un mapeo que va de una lista de constantes a **T o F** (verdadero o falso). Entonces, si *P* es un símbolo de predicado de *n* lugares, y t_1, ..., t_n son términos, entonces $P(t_1,..., t_n)$ es un átomo. Ningún otra expresión puede ser un átomo.

Ahora que los átomos fueron definidos, podemos usar los mismos cinco conectores lógicos del cálculo proposicional. Ya que se introdujeron variables en la lógica

de primer orden, existen dos símbolos especiales $\forall$ y $\exists$ para caracterizar variables. Estos símbolos $\forall$ y $\exists$ son conocidos como los cuantificadores universal y existencial, respectivamente. Si x es una variable, entonces $(\forall x)$ se lee como "para toda x", "para cada x", mientras que $(\exists x)$ se lee como "existe una x", "para alguna x" o "para al menos una x".

Son importantes los conceptos de variables libres y ligadas. Para definirlos es importante la idea del alcance de un cuantificador dentro de una fórmula. Este alcance esta definido por todos los valores que puede tomar la variable.

Una ocurrencia de una variable en una fórmula está ligada si y sólo si la ocurrencia está dentro del alcance del cuantificador usado en la variable. Una ocurrencia de una variable en una fórmula está libre si y sólo si la ocurrencia de la variable no está ligada.

Una variable está libre en una fórmula si por lo menos una ocurrencia de ésta está libre en la fórmula. Una variable está ligada en una fórmula si por lo menos una ocurrencia de la misma está ligada.

Ahora es posible definir una fórmula usando átomos, conectores lógicos y cuantificadores. En la lógica de primer orden una fórmula bien formada (fbf), o fórmula para abreviar, es definida recursivamente del siguiente modo.

1.- Un átomo es una fórmula. Cabe mencionar que "átomo" es la abreviatura de una fórmula atómica.

2.- Si F y G son fórmulas entonces $\neg(F)$, $(F \vee G)$, $(F \wedge G)$, $(F \Rightarrow G)$ y $(F \Leftrightarrow G)$ son fórmulas.

3.- Si F es una fórmula y x es una variable libre en F, entonces $(\forall x)F$ y $(\exists x)F$ son fórmulas.

4.- Las fórmulas son generadas sólo por un número finito de aplicaciones de las tres reglas anteriores.

En ocasiones es conveniente convertir fórmulas bien formadas en formas estándar ó normales. Una fórmula bien formada está en la forma normal prenex si todos los cuantificadores aparecen al principio de la fórmula. La fórmula del enunciado "Todo profesor tiene un diploma" es

$$(\forall x \forall y)\ (\text{enseña}(x,y) \Rightarrow (\exists z)\ \text{diploma}(x,z))$$

Es posible colocar al principio de la fórmula todos los cuantificadores para conseguir expresarla en la forma normal prenex. La fórmula quedaría como sigue:

$$(\forall x)\ (\forall y)\ (\exists z)\ (\neg \text{enseña}(x,y) \vee \text{diploma}(x,z))$$

Una fórmula prenex está en la *forma normal Skolem* cuando todos los cuantificadores existenciales son eliminados reemplazando las variables que cuantifican con funciones arbitrarias de todas las variables cuantificadas universalmente que las precedan en la fórmula. Estas funciones son llamadas *funciones Skolem*; una función Skolem de 0 argumentos es llamada *constante Skolem*. Una cláusula es una disyunción de literales, cuyas variables son cuantificadas universalmente de modo implícito. La forma normal Skolem de la fórmula anterior es

$$\forall x \, \forall y \, (\neg \text{enseña}(x,y) \vee \text{diploma}(x,f(x,y))),$$

donde las variables cuantificadas existencialmente son reemplazadas por funciones Skolem. Cuando una **fbf** se encuentra en la forma normal Skolem, todos los cuantificadores que aparecen al principio de la fórmula pueden ser eliminados ya que todas las variables que quedan son, por convención, cuantificadas universalmente. La fórmula precedente sería reemplazada por

$$\neg \text{enseña}(x,y) \vee \text{diploma}(x,f(x,y)).$$

Así

$$\neg A_1 \vee \ldots \vee \neg A_m \vee B_1 \vee \ldots \vee B_n,$$

donde las A_i y las B_j son literales positivas, es una cláusula. Podemos escribir la cláusula en la forma equivalente

$$A_1 \wedge \ldots \wedge A_m \rightarrow B_1 \vee \ldots \vee B_n.$$

En una cláusula, siempre que n sea igual a 0 o 1, se dice que la cláusula es una *cláusula Horn*. Si tanto m como n son iguales a 0, no existen átomos en ambos lados del signo de implicación y la cláusula es llamada la cláusula vacía. Una cláusula (una literal) en la cual no existen variables es llamada cláusula aterrizada (literal aterrizada). Cualquier **fbf** cerrada puede ser escrita en forma de cláusula. Es posible observar que la transformación de una **fbf** a la forma normal prenex guarda equivalencia, pero no es el caso de transformaciones a la forma normal Skolem o la forma de cláusula. Estas últimas transformaciones solamente preservan la cualidad de satisfactibles, lo cual es suficiente para propósitos de evaluación.

En la lógica proposicional, una interpretación es una asignación de valores de verdad a átomos. En la lógica de primer orden, ya que existen variables, es necesario hacer más que eso. Para definir una interpretación hay que especificar el dominio y una asignación a constantes, símbolos de funciones, y símbolos de predicados de la fórmula.

Una *interpretación* (semántica) de una fórmula F en la lógica de primer orden consiste en un dominio no vacío D, y una asignación de "valores" a cada constante, símbolo de función y símbolo de predicado de F como sigue:

- A cada constante, asignamos un elemento en D.

- A cada símbolo de función de n lugares, asignamos un mapeo de D^n a D. (Nótese que $D^n = \{ (x_1, ..., x_n) \mid x_1 \in D, ..., x_n \in D \}$).
- Para cada símbolo de predicado de n lugares, asignamos un mapeo de D^n a { **T**, **F** }.

En ocasiones, para enfatizar el dominio D, hablamos de una interpretación de la fórmula sobre D. Cuando evaluamos el valor de verdad de una fórmula sobre el dominio D, $(\forall x)$ será interpretado como "para todos los elementos x en D", y $(\exists x)$ como "existe un elemento x en D".

Para cada interpretación de una fórmula sobre un dominio D, la fórmula puede ser evaluada en **T** o **F** de acuerdo a las siguientes reglas:

1.- Si los valores de verdad de fórmulas G y H son evaluados, entonces los valores de verdad de las fórmulas $\sim G$, $(G \wedge H)$, $(G \vee H)$, $(G \Rightarrow H)$, y $(G \Leftrightarrow H)$ son evaluadas mediante la Tabla 1.1.

2.- $(\forall x)G$ es evaluado **T** si el valor de verdad de G es evaluado **T** para cada elemento d en D; de otro modo, es evaluado **F**.

3.- $(\exists x)G$ es evaluado **T** si el valor de verdad de G es **T** para por lo menos un d en D; sino es evaluado **F**.

Consideremos las siguientes fórmulas

$$(\forall x)\, P(x) \text{ y } (\exists x) \sim P(x).$$

Sea una interpretación la siguiente:

Dominio: $D = \{ 1, 2 \}$

Asignaciones para P: $P(1) = \mathbf{T}$; $P(2) = \mathbf{F}$.

Es sencillo confirmar que $(\forall x)\, P(x)$ es **F** en esta interpretación porque $P(x)$ no es **T** tanto para $x = 1$ como para $x = 2$. Por el otro lado, ya que $\sim P(2)$ es verdadera en esta interpretación, $(\exists x) \sim P(x)$ es **T** en esta interpretación.

Una vez que las interpretaciones son definidas, los conceptos como validez, inconsistencia, y consecuencia lógica pueden ser definidos análogamente para fórmulas de la lógica de primer orden.

Una fórmula G es consistente si y sólo si existe una interpretación I tal que G es evaluada **T** en I. Si una fórmula G es **T** en una interpretación I, decimos que I es un modelo de G y que I satisface a G.

Una fórmula G es inconsistente si y sólo si no existe una interpretación que satisfaga a G. Una fórmula G es válida si y sólo si toda interpretación de G satisface a G.

Una fórmula G es una consecuencia lógica de las fórmulas $F_1, F_2, ..., F_n$ si y sólo si para cada interpretación I, si $F_1 \wedge ... \wedge F_n$ es verdadera en I, G es también verdadera en I.

Las relaciones entre validez (inconsistencia) y consecuencia lógica establecidas en los teoremas 1.1 y 1.2 también son verdaderas para la lógica de primer orden. De hecho, la lógica de primer orden puede ser considerada como una extensión de la lógica proposicional. Cuando una fórmula de la lógica de primer orden no contiene variables ni cuantificadores, puede ser tratada como una fórmula de la lógica proposicional.

1.2 Unificación y Resolución

Los dos conceptos de esta sección fueron de suma importancia para implantar lenguajes de programación basados en formalismos tales como la programación lógica. Es conveniente conocerlos para entender mejor los mecanismos específicos de procesamiento del Prolog.

1.2.1 Principio de Resolución para la Lógica Proposicional

El Principio de Resolución de Robinson [ROBI65] es una regla de inferencia que permite que una nueva cláusula sea derivada de dos cláusulas dadas. En la lógica proposicional está definido de la siguiente manera: Sean c_1 y c_2 dos cláusulas cualquiera, si existe una literal l_1 en c_1 que sea complementaria a una literal l_2 en c_2, entonces se borran l_1 y l_2 de c_1 y c_2, respectivamente, y se construye la disyunción de lo que queda de las cláusulas. La cláusula construida es un resolvente de c_1 y c_2. Una propiedad importante es el siguiente teorema:

Teorema 1.3: Dadas dos cláusulas c_1 y c_2, un resolvente c de c_1 y c_2 es una consecuencia lógica de c_1 y c_2.

Demostración: Sean c'_1 y c'_2 disyunciones de literales y $c_1 = l \vee c'_1$, $c'_2 = \neg l \vee c'_2$, y $c = c'_1 \vee c'_2$. Supongamos que c_1 y c_2 son verdaderas en una interpretación I. Deseamos probar que el resolvente c de c_1 y c_2 es también verdadero en I. Para hacerlo, nótese que l ó $\neg l$ es falsa en I. Supongamos que l es falsa en I. Entonces c'_1 es forzosamente verdadera en I. Por lo tanto el resolvente c, o sea c'_1 y c'_2, es verdadera en I. Similarmente, podemos demostrar que si $\neg l$ es falsa en I, entonces c'_2 debe ser verdadera en I. Concluimos que c'_1 y c'_2 debe ser verdadera en I. ◇

Dado un conjunto W de cláusulas, una *deducción* c de W es una secuencia finita $c_1, c_2, ..., c_k$ de cláusulas tales que cada c_i es una cláusula de W ó es un resolvente de cláusulas que preceden a c_i, y $c_k = c$. Una deducción de la cláusula vacía a partir de W es llamada *refutación ó demostración* de W.

1.2.2 Unificación

La parte más importante en el proceso de aplicación del principio de Robinson es encontrar una literal en una cláusula que sea complementaria a otra literal en otra cláusula. Para cláusulas que no contienen variables, esto es muy sencillo. Mas para cláusulas que contengan variables no es tan fácil. Por ejemplo, consideremos las siguientes cláusulas:

$$c1 = P(x) \vee Q(x)$$
$$c2 = \neg P(f(x)) \vee R(x)$$

No existe literal alguna en c1 que sea complementaria a alguna literal en c2. Sin embargo, si substituimos $f(a)$ por x en c1 y a por x en c2, obtendremos

$$c'1 = P(f(a)) \vee Q(f(x))$$
$$c2 = \neg P(f(a)) \vee R(a)$$

Sabemos que c'1 y c'2 son instancias de c1 y c2, respectivamente, y que $P(f(a))$ y $\neg P(f(a))$ son literales complementarias. Por lo tanto de c'1 y c'2 obtenemos el resolvente

$$c'3 = Q(f(x)) \vee R(x)$$

Generalizando, si substituimos $f(x)$ por x en c1 obtenemos

$$c^*1 = P(f(x)) \vee Q(f(x))$$

También c^*1 es una instancia de c1. Esta vez la literal $P(f(x))$ en c^*1 es complementaria a la literal $\neg P(f(x))$ de c2. Por lo tanto podemos obtener un resolvente de c^*1 y de c2,

$$c3 = Q(f(x)) \vee R(x)$$

c'3 es una instancia de la cláusula c3. Substituyendo por los términos apropiados en las variables de c1 y c2 como hasta ahora, podemos generar nuevas cláusulas de c1 y c2. Es más, la cláusula c3 es la cláusula más general en el sentido de que todas las demás cláusulas que puedan ser generadas por el proceso anterior son instancias de c3. c3 puede ser llamada una resolvente de c1 y c2. Formalicemos estos conceptos.

Una substitución es un conjunto finito de la forma $\{ t1/v1,...,tn/vn \}$, donde cada vi es una variable y cada ti es un término diferente de vi, y $vi \neq vj$ donde $1 \leq i \leq n$, $1 \leq j \leq n$, e $i \neq j$.

Sea $\alpha = \{ t1/v1,...,tn/vn \}$ una substitución y sea E una expresión. Entonces $E\alpha$ es la expresión obtenida de E al reemplazar simultáneamente cada ocurrencia de la variable vi, donde $1 \leq i \leq n$, en E por el término ti. $E\alpha$ es llamada una instancia de E.

Sean $\alpha = \{ t1/x1,...,tn/xn \}$ y $\beta = \{ t1/y1,...,tn/yn \}$ dos substituciones. Entonces la composición, denotada $\alpha * \beta$ es la substitución que es obtenida del conjunto

$$\{ t1\beta/x1,...,tn\beta/xn, u1/y1,...,un/yn \}$$

borrando cualquier elemento $tj\beta/xj$ para el cual $tj\beta = xj$, y cualquier elemento ui/yi tal que yi está entre $\{ x1,...,xn \}$.

En el procedimiento de resolución, para identificar un par de literales complementario, a menudo es necesario unificar (emparejar) dos o más expresiones. Esto es, tenemos que encontrar una substitución que pueda hacer a varias expresiones idénticas. Por lo tanto consideraremos ahora la unificación de expresiones.

Una substitución α es un unificador para conjunto { E1,...,Ek } si y sólo si E1α = E2α = ... = Ekα. El conjunto { E1,...,Ek } se dice ser unificable si posee un unificador.

Un unificador χ para un conjunto de expresiones { E1,...,Ek } es el unificador general si y sólo si para cada unificador α del conjunto existe una substitución β tal que $\alpha = \chi * \beta$.

Ahora veremos un algoritmo de unificación para encontrar al unificador general de un conjunto unificable finito de expresiones no vacías. Cuando el conjunto no es unificable, el algoritmo también detectará este hecho.

Consideremos las expresiones P (a) y P (x) que no son idénticas. Para unificarlas debemos primero encontrar las diferencias e intentar eliminarlas. Para P (a) y P (x), la diferencia es { a, x }. Como x es una variable, podemos remplazarla por a, y así eliminar la diferencia. Esta es la idea principal del algoritmo de unificación.

El conjunto de diferencias de un conjunto no vacío W de expresiones es obtenido localizando el primer símbolo (contando desde la izquierda) en el cual no todas las expresiones de W son exactamente el mismo símbolo, y luego extraer de cada expresión de W la subexpresión que comienza con el símbolo que ocupa dicha posición. El conjunto de las subexpresiones respectivas es el conjunto de diferencias.

Algoritmo de Unificación

Paso 1 Sea k = 0, Wk = W, y χk = ϵ.

Paso 2 Si Wk es una sola expresión, detenerse; χk es el unificador general para W. De otro modo, encuentra el conjunto de diferencias Dk de Wk.

Paso 3 Si existen elementos vk y tk en Dk tales que vk es una variable que no aparece en tk, ir al Paso 4. De otro modo, detenerse; W no es unificable.

Paso 4 Sea χk + 1 = χk { tk/vk } y Wk + 1 = Wk { tk/vk }. (Nótese que Wk + 1 = Wχk + 1.)

Paso 5 Sea k = k + 1 e ir al Paso 2.

Hay que hacer notar que este algoritmo siempre terminará para todo conjunto finito no vacío de expresiones, pues de otro modo generaría una serie infinita de conjuntos no vacíos de expresiones Wχ0, Wχ1, Wχ2,... con la propiedad de que cada conjunto sucesivo contendrá una variable menos que su predecesor (esto es, Wχk

contendrá a v_k pero $W\chi_k + 1$ no). Esto es imposible ya que W sólo contiene una cantidad finita de variables.

Si W es unificable, entonces el algoritmo de unificación encontrará al unificador general. Esto es comprobado en el siguiente teorema.

Teorema 1.4: (Teorema de Unificación) Si W es un conjunto finito no vacío unificable de expresiones, entonces el algoritmo de unificación siempre terminará en el Paso 2, y el último $W\chi_k$ es el unificador general de W.

Demostración: Ya que W es unificable, sea χ cualquier unificador para W. Para $k = 0$, 1, ..., demostraremos que existe una substitución β_k tal que $\alpha = \chi_k \circ \beta_k$ por inducción sobre k. Para $k = 0$, sea $\beta_0 = \alpha$. Entonces $\alpha = \chi_0 \circ \beta_0$, ya que $\chi_0 = \varepsilon$. Supongamos que $\alpha = \chi_k \circ \beta_k$ donde $0 \leq k \leq n$. Si $W\chi_n$ es una sola expresión, entonces el algoritmo de unificación termina en el Paso 2. Ya que $\alpha = \chi_n \circ \beta_n$, χ_n es el unificador general para W. Si $W\chi_n$ contiene más de una expresión, entonces el algoritmo de unificación buscará el conjunto de diferencias D_n de $W\chi_n$. Debido a que $\alpha = \chi_n \circ \beta_n$ es un unificador para W, β_n debe de unificar a D_n. Sin embargo, ya que D_n es el conjunto de diferencias, debe de existir una variable en D_n. Sea t_n cualquier otro elemento diferente de v_n. Entonces ya que β_n unifica a D_n, $v_n\beta_n = t_n\beta_n$. Ahora, si v_n apareciera en t_n, entonces $v_n\beta_n$ aparecería en $t_n\beta_n$. Sin embargo, es imposible ya que v_n y t_n son distintos, y $v_n\beta_n = t_n\beta_n$. Por consiguiente v_n no aparece en t_n. De aquí que el algoritmo de unificación no terminará en el Paso 3, pero irá al Paso 4 para hacer $\chi_n + 1 = \chi_n \circ \{ t_n/v_n \}$. Sea $\beta_n + 1 = \beta_n - \{ t_n\beta_n/v_n \}$. Entonces, ya que v_n no aparece en t_n, $t_n\beta_n + 1 = t_n (\beta_n - \{ t_n\beta_n/v_n \}) = t_n\beta_n$. Entonces tenemos que

$$\begin{aligned} \{ t_n/v_n \} \circ \beta_n + 1 &= \{ t_n\beta_n + 1/v_n \} \cup \beta_n + 1 \\ &= \{ t_n\beta_n/v_n \} \cup \beta_n + 1 \\ &= \{ t_n\beta_n/v_n \} \cup (\beta_n - \{ t_n\beta_n/v_n \}) \\ &= \beta_n \end{aligned}$$

Esto es, $\beta_n = \{ t_n/v_n \} \circ \beta_n + 1$. Por lo tanto

$$\alpha = \chi_n \circ \beta_n = \chi_n \circ \{ t_n/v_n \} \circ \beta_n + 1 = \chi_n + 1 \circ \beta_n + 1.$$

De aquí que, para toda $k \geq 0$, existe una substitución β_k tal que $\alpha = \chi_k \circ \beta_k$. Ya que el algoritmo de unificación debe terminar, y ya que no terminará en el Paso 3, debe de terminar en el Paso 2. Además, ya que $\alpha = \chi_k \circ \beta_k$ para toda k, la última χ_k es el unificador general para W. ◇

1.2.3 Principio de Resolución para la Lógica de Primer Orden

Habiendo introducido el algoritmo de unificación, ahora ya podemos considerar el principio de Resolución para la Lógica de primer orden.

Si dos o más literales (con el mismo signo) de una cláusula c tienen unificador general χ, entonces $c\chi$ es llamado un factor de c.

Sean c1 y c2 dos cláusulas (llamadas cláusulas padres) sin variables en común. Sean l1 y l2 dos literales en c1 y c2, respectivamente. Si l1 y $\sim$ l2 tienen un unificador general χ, entonces la cláusula

$$(c1\chi - l1\chi) \cup (c2\chi - l2\chi)$$

es llamada la resolvente binaria de c1 y c2. Las literales l1 y l2 son las literales sobre las cuales se aplicó la resolución.

Un resolvente de las cláusulas (padres) c1 y c2 es uno de los siguientes resolventes binarios:

- el resolvente binario de c1 y c2,
- el resolvente binario de c1 y un factor de c2,
- el resolvente binario de un factor de c1 y c2,
- el resolvente binario de un factor de c1 y un factor de c2.

Veamos un ejemplo. De

c1: $\sim$ P (a,b,c) $\vee$ Q (d,e), y
c2: P (x,y,z) $\vee$ R (x,y),

se obtiene el resolvente

c3: Q (d, e) $\vee$ R (a, b).

La cláusula c3 es encontrada considerando las literales en las dos cláusulas que tienen el mismo nombre de predicado, pero uno está negado y el otro no. El único predicado de este tipo es P. Entonces uno determina si las dos literales { P (a,b,c), P(x,y,z) } pueden ser unificadas, donde a, b, y c se asume que sean constantes y x, y y z se asume que sean variables. El unificador buscado es { a/x, b/y, c/z }. El factor de c2 es

c'2: P (a,b,c) $\vee$ R (a,b)

Finalmente obtenemos el resolvente binario c3 de las cláusulas c1 y c'2.

El principio de resolución es muy usado para realizar pruebas de refutación: Para demostrar que la cláusula c es derivable del conjunto de cláusulas W, se intenta demostrar que W y $\sim$ c no son simultáneamente satisfactibles. Como la resolución preserva la satisfactibilidad, si uno puede, por resolución de las formas clausuláres de W y $\sim$ c, producir resolventes sucesivos hasta llegar a la cláusula vacía, entonces W y $\sim$ c no pueden ser satisfechas simultáneamente. Esto se logra debido a que el principio de resolución es completo, esto es, que siempre generará la cláusula vacía de un conjunto no satisfactible de cláusulas. (Ver [CHAN73]).

1.3 Deducción

Dos aspectos complementarios de las fórmulas bien formadas son de interés. Uno se refiere a la semántica (o teoría del modelo), la especificación de valores de verdad para fbfs, mientras que el otro trata con la sintaxis (o teoría de prueba), la derivación de una fórmula de un conjunto dado de fbfs.

1.3.1 Semánticas: modelo e interpretación

En las semánticas se trabaja con interpretaciones, donde una interpretación es un conjunto de fbfs que consiste de la especificación de un conjunto no vacío (o dominio) E, para el cual constantes y variables son valores dados. A cada símbolo de función n-ario se le asigna una función de E^n a E. A cada predicado n-ario se le asigna una relación en E^n.

En una interpretación con dominio E, una fbf cerrada puede ser verdadera o falsa, donde una fbf(abierta) con n ($n \geq 1$) variables libres determina un conjunto de n-tuplas (esto es, una relación) en E^n. Cada una de estas n-tuplas es tal que cuando sus componentes son sustituídos por las correspondientes variables libres en la fbf abierta, entonces en esta interpretación, la fbf cerrada que se obtiene es verdadera. Si el conjunto de n-tuplas está vacío entonces la fbf abierta se dice que es falsa, y si el conjunto de n-tuplas coincide con E^n, entonces la fbf abierta se dice que es verdadera. Los valores de verdad de una fbf cerrada se obtienen del siguiente modo. Si R es la relación asignada a un símbolo de predicado de n lugares P, entonces $P(e_1, ..., e_n)$ es evaluado como verdadero si $(e_1, ..., e_n) \in R$; de otro modo, es evaluado como falso. Ahora, si w_1 y w_2 son fbfs cerradas, $\neg w_1$ es verdadera si w_1 es falso; de otro modo ésta será falsa. $w_1 \wedge w_2$ es verdadera si ambas w_1 y w_2 son verdaderas; de otro modo es falsa. $w_1 \rightarrow w_2$ es verdadera si ya sea w_1 falsa o w_2 verdadera; de otro modo es falsa. Fórmulas bien formadas construídas usando los otros símbolos lógicos pueden ser evaluadas de manera similar. Finalmente, si x es una variable en w, $\forall x w(x)$ (respectivamente, $\exists x w(x)$) evalúa verdadera si para todos los elementos e_1 en E (respectivamente, hay un elemento $e_1 \in E$ tal que) $w(e_1)$ es verdadera; de otro modo es falsa.

Un modelo de un conjunto de fórmulas bien formadas es una interpretación en la cual todas las fbfs en el conjunto son verdaderas. Una fbf w se dice que es una consecuencia lógica de un conjunto de fbfs w si y sólo si w es verdadera en todos los modelos de w. Esto se denota por $w \mid = w$.

1.3.2 Sintaxis: Teoría de Primer Orden

El cálculo de predicados de primer orden es un sistema formal que tiene como lenguaje objeto un lenguaje de primer orden, un conjunto de axiomas (los axiomas

lógicos), y dos reglas de inferencia: modus ponens (Teorema 1.1) y generalización. Cuando otras **fbfs** son añadidas como axiomas, el sistema formal resultante es llamado una teoría de primer orden. Los nuevos axiomas son llamados axiomas no lógicos o propios. Una teoría de primer orden es caracterizada sencillamente por sus axiomas no lógicos. Un conjunto de axiomas no lógicos puede ser, por ejemplo,

Hombre(Sócrates),
$(\forall x)$(Hombre(x) $\rightarrow$ Mortal(x)).

Un modelo de una teoría es una interpretación en la cual todos los axiomas son verdaderos; los axiomas lógicos son, de hecho, escogidos para ser verdaderos en todas las interpretaciones. Para la teoría anterior,

Hombre(Sócrates) = **T**,
Mortal(Sócrates) = **T**

produce un modelo ya que hace a todos los enunciados de la teoría anterior verdaderos. Una **fbf** w es derivable de un conjunto de **fbfs** W en una teoría T(W | - w) si y sólo si w es deducible de W y de los axiomas de T por una aplicación finita de las reglas de inferencia.

Usando la regla de inferencia modus ponens, que establecen que de p y $p \rightarrow q$ uno puede concluir q, obtenemos de la teoría anterior el resultado derivado siguiente: Mortal (Sócrates). Si W está vacío, entonces w es un teorema de T (| -$_T$ w, o equivalentemente T | - w). Siempre que T esté clara, podremos escribir W | - w para W | -$_T$ w.

Reglas de inferencia aparte de modus ponens y generalización pueden ser usadas para derivar teoremas; de hecho la mayoría de las técnicas usadas en la demostración de teoremas están basadas en la regla de inferencia establecida por el Principio de Resolución de Robinson [ROBI65], la cual se aplica a **fbfs** en forma clausular.

La relación más importante entre los enfoques semántico y sintáctico es la existente entre los conceptos de solidez (soundness) y completitud.

Un sistema de inferencia es *sólido* si y sólo si para todo W y w, siempre que W | - w implica que W | = w; es *completo* si y sólo si para todo W y w, siempre que W | = w, implica que W | - w.

Las reglas de inferencia modus ponens y generalización son completas y sólidas para el cálculo proposicional. Similarmente la refutación por resolución es completa y sólida para teorías de primer orden: la cláusula vacía es derivada si y sólo si la cláusula inicial (la cual es negada para aplicar resolución) es un teorema en la teoría. Sin embargo, hay un elemento de no-decidibilidad; si la cláusula propuesta para ser probada no es un teorema; el proceso de inferencia puede no terminar. La refutación por resolución es también completa y sólida. Lo que significan la completitud y la solidez es que los resultados obtenidos mediante asignaciones de verdad (semántica),

son los mismos que los obtenidos mediante reglas de inferencia (demostraciones sintácticas). Como referencias al tema de lógica matemática están [ENDE72] y [MEND78] por citar algunos.

1.4 Programación Lógica

Debemos, antes que nada, enfatizar la diferencia que existe entre la programación lógica y la programación en el lenguaje Prolog. Los programas lógicos pueden ser entendidos destacando dos conceptos independientes de las máquinas: verdad o veracidad y deducción lógica. Uno puede preguntar si un axioma en un programa es verdadero bajo alguna interpretación de los símbolos del programa; o investigar si un enunciado lógico es una consecuencia del programa. Estas preguntas pueden ser contestadas independientemente de cualquier mecanismo concreto de ejecución. Por el otro lado, Prolog es un lenguaje de programación, que toma sus construcciones básicas de la lógica. Los programas en Prolog tienen un significado operacional preciso: son instrucciones para ser ejecutadas en una computadora. Los programas en Prolog escritos con buen estilo pueden casi siempre ser leidos como enunciados lógicos. Es más, el resultado de una computación de algún programa en Prolog es una consecuencia lógica de los axiomas que éste contiene. Por lo tanto el lenguaje de programación Prolog es un subconjunto del lenguaje descrito por la programación lógica. Es importante conocer la teoría de la programación lógica para elaborar programas en Prolog que sean eficientes.

La idea de la Programación Lógica surgió en Marsella durante la primera mitad de 1972 cuando Robert Kowalski visitaba el equipo de inteligencia artificial fundado por Alain Colmerauer en la Universidad de Marsella. Colmerauer y su equipo desarrollaron las especificaciones de un lenguaje de programación llamado Prolog [COLM72], fundamentado en los principios formales de la programación lógica. El lenguaje se veía como un demostrador de teoremas, pero tenía las propiedades esenciales que en la actualidad posee. Casi al mismo tiempo Kowalski tomó al cálculo de predicados como un formalismo para expresar algoritmos sin la necesidad de especificar la estrategia a seguir (control) de su ejecución; publicó una breve nota [KOWA72] que fue después desarrollada como un reporte más grande [KOWA74]. Esta filosofía está reflejada en el nombre de su libro [KOWA79] "Algoritmo = Lógica + Control" (parafraseando al popular libro de Nicklaus Wirth [WIRT76]), la cual enfatiza la diferencia entre el "qué" hacer (lógica) y el "como" hacerlo (control). Esto se debe a que la programación lógica casi siempre releva al programador en la tarea de especificar el control del programa. Sin embargo en la práctica el Prolog puede ser tratado como un lenguaje procedural.

Los dos pioneros de la programación lógica tomaron diferentes caminos para convertir esa idea en realidad.

Prolog-10 es un dialecto con bastante difusión (también conocido como Prolog de Edimburgo), fue implantado por David H. D. Warren en una computadora DEC-10. Las referencias se encuentran en [PERE78], [BOWE81] y [CLOC81]. La descripción del lenguaje que a continuación aparece corresponde a este dialecto de Prolog.

1.4.1 Constantes

Las constantes son los "bloques" más sencillos para la construcción de estructuras de datos. Las constantes no tienen estructura, por lo que son llamadas "átomos". Solo se representan a si mismas, puede pensarse que ellas son idénticas a sus nombres.

Las definiciones de tipo en lenguajes comunes como C y Pascal confieren cierta estructura a las constantes y cada una tiene un único atributo: su nombre.

La interpretación de una constante esta en manos del programador. 1991 puede ser el precio de algún artículo, el peso de un camión, la hora del día o el año de nacimiento. Las constantes son las primitivas, y reunirlas en tipos sólo debe de hacerse cuando sea necesario. En Prolog, así como en otros lenguajes simbólicos (como Lisp) no hay necesidad de declarar las constantes o agruparlas en tipos. Uno puede usarlas libremente simplemente escribiendo sus nombres. Esto se debe a que la mayoría de los traductores de estos lenguajes son interpretes.

Un nombre de una constante puede ser uno de los siguientes:

- Una secuencia de dígitos, pudiendo ser antecedidos por un signo de menos; por convención estas constantes son conocidas como enteros;
- Un identificador, que puede contener letras, dígitos y el símbolo de subrayado (_), pero debe de comenzar con una letra minúscula;
- Un símbolo que es una secuencia no vacía de cualquiera de los siguientes caracteres: `+ - * / = . : ? $ & @ # \`
- Cualquiera de los caracteres: `, ; !`
- El símbolo `[ ]` (llamado "nil");
- Un nombre entre comillas, que es una secuencia arbitraria de caracteres encerrados entre apóstrofes.

Todas estas constantes son simbólicas y no tienen interpretación inherente. Sin embargo, algunas operaciones primitivas en Prolog pueden tratarlas de un modo especial:

- Operaciones aritméticas interpretan a los enteros como representaciones de valores enteros;
- Operaciones de comparación interpretan a los enteros como valores enteros, y todas las demás constantes como representación de las

secuencias de caracteres que forman sus nombres (son consideradas bajo un orden lexicográfico);

- Operaciones de Entrada/Salida interpretan a todos los símbolos como secuencias de caracteres formando sus nombres.

La descripción de un objeto compuesto consiste en el nombre, seguido por una secuencia entre paréntesis de descripciones de sus componentes, separados por comas, por ejemplo:

```
rectángulo (19,25)
hora_actual (19,25)
```

La notación es similar a la usada para escribir funciones en matemáticas. La terminología refleja dicha similitud. El nombre es llamado functor, y los componentes son llamados argumentos. Pueden existir functores sin argumentos. Los dos atributos importantes de un functor son su nombre y su aridad (esto es, el número de atributos que posee). Si dos objetos poseen el mismo nombre pero diferente aridad, los objetos son diferentes. La notación más usual es escribirlos separados por una diagonal, por ejemplo:

```
rectángulo/2.
```

Las reglas léxicas para la formación de nombres de functores es la misma para los atributos, pero los enteros y [ ] pueden ser únicamente constantes. Entonces

```
123(a,b)
```

es incorrecto, pero

```
'123'(a,b)
```

es aceptable.

1.4.2 Objetos compuestos

Las descripciones de constantes y de objetos compuestos son referidas como términos. Los argumentos de un término son términos arbitrarios. Por ejemplo, es posible escribir un término describiendo un registro:

```
cliente(nombre(José,Esparza),
  dirección(calle(Matamoros),número(104))).
```

De todos los functores de este ejemplo, el más externo, `cliente/2`, puede decirse que define la estructura general del término. Es llamado el functor principal. Análogamente, `nombre/2` es el functor principal del primer argumento.

Ahora viene otra estructura de datos. Una lista puede ser definida como la lista vacía, o una lista compuesta de cualquier objeto (cabeza) y una lista (una cola).

Los functores prefijos, infijos y postfijos son conocidos comúnmente con el nombre genérico de operadores. Estos no son operadores en sentido convencional, son una conveniencia sintáctica.

Los nombres de los operadores no deben ir con apóstrofes. Si un operador va a ser escrito en forma estándar o con un diferente número de argumentos, entonces debe poseer apóstrofes. Si + es un functor infijo,

```
a + b,
'+'(a,b) y
'+'(a,b,c)
```

son términos correctos, pero

```
+y
+(a,b)
```

no lo son.

También es posible declarar operadores mixtos, esto es, functores como el signo de menos, el cual es prefijo e infijo en la aritmética ordinaria.

Los caracteres son constantes cuyos nombres consisten de caracteres sencillos. Es posible usar nombres entre apóstrofes para caracteres que no sean identificadores validos. Las cadenas son listas de caracteres, y pueden ser escritas entre comillas.

1.4.3 Variables

Los objetos que han sido comentados hasta ahora son constantes. Su estructura esta definida, sabemos todo acerca de ellos y no nos aportan nada nuevo. Es importante el poder usar objetos que sean dinámicos durante la ejecución del programa. En Prolog, estos objetos son llamados variables. El término que la denota se conoce como el nombre de la variable, el cual se escribe como un identificador comenzando con una letra mayúscula o el caracter de subrayado. Una variable es un objeto cuya estructura es desconocida. Así como la ejecución del programa avanza, la variable puede ser instanciada, esto es, se determina una descripción más precisa del objeto. El término con que se define esta descripción es la instanciación de la variable. Una variable instanciada es idéntica al objeto descrito por su instanciación, entonces deja de ser variable, siempre que el objeto pueda ser referido mediante el nombre de la variable. En general, una variable puede ser instanciada a otra variable. Existe una terminología alternativa, se dice que una variable libre se ata a otro término y de aquí en adelante no es distinguible de ese término (el cual es llamado atadura). La variable se aterriza si su atadura no contiene variables. Esta terminología trae a la mente el proceso de atar parámetros formales a los parámetros actuales.

Intuitivamente, las variables en Prolog son parecidas a las variables matemáticas. Cuando decimos que

$$f(x) = e^x + 2x$$

es una función de una variable, significa que la expresión nos permite determinar el valor de la función para cualquier argumento dado. La variable denota una substitución y no es por sí misma un objeto cuyos valores puedan ser asignados. También es posible ver a una variable como un apuntador "invisible". Cuando no esta libre, la referencia al apuntador se realiza automáticamente, por lo que es imposible distinguirlo del objeto apuntado. Si uno ve a un tipo como un conjunto de variables, entonces un término es también una definición de un tipo. El término `Variable2` describe el conjunto de todos los objetos, puesto que la variable puede ser instanciada a cualquier cosa. Por el otro lado, es posible tener una especificación de tipo muy precisa. Por ejemplo, el término `a.b.c.[]` describe un conjunto que contiene un único objeto: la lista de tamaño 3, cuyo primer elemento es a, cuyo segundo elemento es b y el tercer elemento es c. Existe una amplia gama de opciones entre estos dos extremos. El tipo de un objeto compuesto es definido primariamente por la interrelación entre el objeto y sus componentes, más que por el tipo de los componentes. Así como transcurre la ejecución del programa, las variables en varios términos son instanciadas. Como resultado se conoce más de los objetos descritos por esos términos. Entonces es posible hablar de términos instanciados y de términos que son instanciaciones de otros términos. Por ejemplo, `f(X).Tail` es una instanciación de `Head.Tail`; y `f(X)` a su vez puede ser instanciado a una descripción más precisa. Una aproximación de varios pasos a una descripción deseada es una característica de Prolog.

Las variables son usadas como manejadores de los objetos que denotan. Con el nombre podemos ver que sabemos de la forma de un objeto.

En ocasiones no hay interés en ciertos objetos y no es necesario un nombre para referirse a ellos. Estos términos pueden ser denotados por variables anónimas, que pueden ser descritas cada una por el caracter de subrayado _.

1.4.4 Procedimientos

La mayoría de las operaciones en un programa en Prolog son llamadas a procedimientos definidos por el usuario. Operaciones comunes como sumas, comparaciones, de entrada/salida son relativamente poco usadas. Estas son conocidas como procedimientos interconstruidos o del sistema.

Las llamadas a procedimientos son escritas del siguiente modo: el nombre del procedimiento seguido por una lista opcional de términos (parámetros) encerrados por paréntesis y separados por comas. El nombre del procedimiento es llamado símbolo del predicado o predicado. Así como los functores, el símbolo del predicado tiene dos atributos: su nombre y su aridad. Dos procedimientos diferentes pueden

compartir el mismo nombre, pero teniendo distinta aridad. Algunas versiones de Prolog permiten usar notación prefija, infija o postfija para símbolos de predicados (u operadores) como en los functores; pero existe una serie de símbolos usados con mayor frecuencia que son predeclarados para dar al lenguaje una apariencia más convencional. Los functores de dos argumentos `+`,`-`, `*`, `/`, y `mod` son predeclarados como functores infijos con prioridades convencionales.

Secuencias de llamadas a procedimientos usan comas como separadores. Para empezar tenemos dos procedimientos interconstruidos:

- `nl/0` termina una línea de salida;
- `write/1` escribe un término en el dispositivo de salida.

En versiones de Prolog derivadas de Prolog-10, la sintaxis de la definición de un procedimiento no difiere necesariamente de la llamada a ese procedimiento. El significado es definido por contexto. Los sistemas de Prolog funcionan en dos modos: el modo de comandos y el modo de definición. Por omisión es el modo de comandos. Aquí el sistema lee y ejecuta directivas (u objetivos), las cuales son leídas del teclado o de un archivo. Cada directiva es terminada por el símbolo de alto total (el caracter "." seguido de un espacio en blanco y la nueva línea), y puede ser una consulta o un comando. Una consulta es una llamada a un procedimiento o una secuencia de estas separadas por comas. Actúa ejecutando las llamadas e imprimiendo las instanciaciones de variables resultantes. Un comando tiene la forma de una consulta más el prefijo `:-`. Las llamadas son ejecutadas pero las instanciaciones de las variables no son enviadas a la salida automáticamente. Se accesa al modo de definición ejecutando los procedimientos del sistema `consult/1` o `reconsult/1`. El argumento es el nombre del archivo del cual serán leídas las definiciones de procedimientos. En este modo el sistema acepta las definiciones de procedimientos, que deben terminar con alto total. Los comandos son permitidos y ejecutados en este modo.

Un comentario en Prolog comienza con el caracter `%` y se extiende hasta el final de la línea.

Es importante contar con un mecanismo eficiente y general para el paso de parámetros, y esta operación de emparejamiento de términos se llama unificación. La función de unificar elabora pares de parámetros actuales y formales. Si la función devuelve cierto para todos los pares de términos, decimos que la unificación fue exitosa, de otro modo la unificación falló. La unificación falla cuando los términos que describen a los parámetros no coinciden. En un sentido general esto significa que los tipos de los parámetros actuales no son compatibles con aquellos de los parámetros formales. Es posible extender la terminología y decir que -como un par de términos- una llamada a un procedimiento y el encabezado del mismo pueden o no ajustar, esto es, que pueden ser o no ser unificables. Cuando la unificación falla no es posible realizar la instanciación. Términos unificados no son distinguibles, pues describen al

mismo objeto. Si tanto los parámetros formales como los actuales describen variables, la unificación los liga. Las variables que están ligadas describen al mismo objeto, sus nombres se refieren a la misma variable; y es indiferente decir que la variable formal esta instanciada a la variable actual o viceversa.

1.4.5 Cláusulas

Después de que los parámetros son pasados, un procedimiento puede -igual que en otros lenguajes- ejecutar una secuencia de operaciones. El cuerpo del procedimiento es una secuencia de llamadas a otros procedimientos, separados por comas y llevan el prefijo :-. Un encabezado de procedimiento, posiblemente seguido de un cuerpo, es llamado una cláusula.

Ahora, ¿qué sucede cuando falla la unificación? Parte de la respuesta es que un procedimiento puede consistir en un número de cláusulas. Todas estas cláusulas deben tener encabezados con el mismo símbolo del predicado, pero las especificaciones de parámetros pueden variar. Cuando la unificación de una llamada con el primer encabezado de cláusula es exitosa, la primera cláusula ejecuta su cuerpo, si lo tiene. Cuando la unificación falla, sus efectos se deshacen: todas las variables que fueron instanciadas en el intento de unificación son devueltas a su estado original. Entonces la llamada es comparada con el encabezado de la segunda cláusula. Si esta es exitosa, la segunda cláusula es ejecutada; de otro modo se intenta con la tercer cláusula, y así sucesivamente. Todos los términos escritos en una cláusula son locales a esa cláusula. Como en otros lenguajes de programación que tienen recursión, la activación de una cláusula es acompañada por la creación de nuevas instancias de todos sus objetos locales. Los términos que aparecen en la cláusula describen estas instancias. Antes que el intento de unificar la llamada con un encabezado de cláusula pueda ser hecho, una nueva instancia de la cláusula es creada. Cuando la unificación falla, la instanciación es destruida.

¿Qué sucede si una llamada a un procedimiento no coincide con ningún encabezado de cláusula? Si ninguna de las cláusulas se ajusta a la llamada, entonces evidentemente la llamada fue equivocada: su conjunto de parámetros actuales no checa con ninguna de las especificaciones de tipo que describen parámetros aceptables por el procedimiento. Como en otros lenguajes de programación modernos, una llamada errónea no termina anormalmente con la ejecución del programa, sino que activa un mecanismo de manejo de errores. En contraste con otros lenguajes el error no es necesariamente manejado por un procedimiento activo presente en la pila de activación. Prolog utiliza un método más general y toma en cuenta aun aquellos procedimientos que regresan a su llamada después de terminar exitosamente su ejecución. Las instancias de un procedimiento son vistas, una por una, en orden inverso a su activación. Si se efectúa una llamada de un procedimiento, se evalúa la primer cláusula que coincida, y si después existen varias cláusulas que chequen con la llamada, el sistema es capaz de manejar la situación. Cuando se termina de evaluar la primer

cláusula, independientemente si fue instanciada o no, la llamada de procedimiento tiene la oportunidad de ejecutar otras cláusulas. Este proceso es llamado vuelta atrás o backtracking y una llamada que no checa con ningún encabezado de cláusula se dice que falla. La vuelta atrás es muy similar a nuestro comportamiento en la búsqueda sistemática de la solución de un problema. Si llegamos a un callejón sin salida, nos regresamos al punto donde podemos hacer otro intento. Si no existe ningún camino que funcione, regresamos al lugar previo donde aparentemente cometimos el error, y así sucesivamente. En términos del funcionamiento de Prolog, cada vez que una cláusula no es la última en su procedimiento, un registro es introducido en una pila especial de puntos de falla (también llamados puntos de elección). El registro contiene toda la información necesaria para restaurar el estado de la computación. Cuando un procedimiento falla, un punto de falla es sacado de la pila, se restaura el estado descrito en el, y la ejecución prosigue con la siguiente cláusula. Algunos procedimientos del sistema no pueden ser deshechos, como escribir información en pantalla, entonces se dice que son procedimientos con efectos colaterales.

Si en algún caso es necesario detener la vuelta atrás -cuando, por ejemplo, se ha llegado a la respuesta (o respuestas) requerida y no es necesario seguir buscando- existe el procedimiento corte (cut) y se denota con el caracter !.

1.5 Estructura de Prolog

La notación usada para la descripción sintáctica de Prolog es Backus Naur Form (BNF) extendida. Los símbolos no terminales se encuentran en mayúsculas, y algunos de ellos subindizados (esta es la primera extensión). Una regla BNF toma la siguiente forma:

LHSNOTERM :: = RHS1 | ... | RHSN

Cada RHS es una secuencia de símbolos terminales y no terminales.

La segunda extensión: cero o más ocurrencias de la secuencia *s* son denotadas como {*s*}.

CLAUSULA :: = DEFINICION | REGLA_GRAMATICAL | DIRECTIVA

DEFINICION :: = CLAUSULA_NO_UNITARIA | CLAUSULA_UNITARIA

CLAUSULA_NO_UNITARIA :: = CABEZA :- CUERPO

CLAUSULA_UNITARIA :: = CABEZA

Comentario: el functor principal de CABEZA no es el :- binario.

CABEZA :: = NO_VARIABLE_NI_ENTERO

CUERPO :: = CUERPO_ALT { ; CUERPO_ALT }

CUERPO_ALT :: = LLAMADA { ; LLAMADA }

LLAMADA :: = NO_VARIABLE_NI_ENTERO | VARIABLE | (CUERPO)

NO_VARIABLE_NI_ENTERO :: = TERMINO

REGLA_GRAMATICAL :: = LADO_IZQUIERDO --> LADO_DERECHO

LADO_IZQUIERDO :: = NO_TERMINAL CONTEXTO | NO_TERMINAL

NO_TERMINAL :: = NO_VARIABLE_NI_ENTERO

CONTEXTO :: = TERMINALES

LADO_DERECHO :: = ALTERNATIVAS

ALTERNATIVAS :: = ALTERNATIVA { ; ALTERNATIVA }

ALTERNATIVA :: = ELEMENTO_REGLA { , ELEMENTO_REGLA }

ELEMENTO_REGLA :: = NO_TERMINAL | TERMINALES | CONDICION | ! | (ALTERNATIVAS)

TERMINALES :: = LISTA | CADENA

Comentario: sólo se permiten listas cerradas.

CONDICION :: = TERMINO_CICLO

DIRECTIVA :: = COMANDO | CONSULTA

COMANDO :: = :- CUERPO

CONSULTA :: = CUERPO

Comentario: el functor principal de CUERPO no es el : – unario.

TERMINO :: = $TERMINO_{1200}$

$TERMINO_N$ :: = $OP_{fx,N}$ $TERMINO_{N-1}$ | $OP_{fy,N}$ $TERMINO_N$ | $TERMINO_{N-1}$ $OP_{xf,N}$ | $TERMINO_N$ $OP_{yf,N}$ | $TERMINO_{N-1}$ $OP_{xfx,N}$ $TERMINO_{N-1}$ | $TERMINO_{N-1}$ $OP_{xfy,N}$ $TERMINO_N$ | $TERMINO_N$

$OP_{yfx,N}$ $TERMINO_{N-1}$ | $TERMINO_{N-1}$

Comentario: $1 \leq N \leq 1200$; $OP_{Type,N}$ es un operador de tipo Type y prioridad N; $TERMINO_N$ puede ser llamado "término con prioridad N".

$TERMINO_0$ :: = VARIABLE | ENTERO | CADENA | LISTA | NO_OP |

NO_OP (TERMINO { , TERMINO }) | (TERMINO) | TERMINO_CICLO

TERMINO_CICLO :: = { TERMINO }

NO_OP :: = FUNCTOR

OP $_{T,N}$ :: = FUNCTOR

Comentario: T puede ser uno de fx, fy, xf, yf, xfx, xfy, yfx, N está en el rango 1,...,1200, vea también la nota 1.

LISTA :: = [ ] | [TERMINO$_{999}$ { , TERMINO$_{999}$ }] | [TERMINO$_{999}$ { , TERMINO$_{999}$ } | TERMINO]

Comentario: los términos con prioridad 999 pueden ser unidos con seguridad por comas las cuales son functores infijos con prioridad 1000.

FUNCTOR :: = PALABRA | NOMBRE_Q | SIMBOLO | CARACTER_SOLO

PALABRA :: = INICIO_PALABRA { ALFANUMERICO }

INICIO_PALABRA :: = LETRA_CHICA

ALFANUMERICO :: = LETRA_CHICA | LETRA_GRANDE | DIGITO | _

NOMBRE_Q :: = ' { ELEMENTO_Q } '

ELEMENTO_Q :: = '' | NO_APOSTROFE

Comentario: NO_APOSTROFE es cualquier caracter diferente de '

SIMBOLO :: = CARACTER_SIMBOLO { CARACTER_SIMBOLO }

VARIABLE :: = INICIO_VARIABLE { ALFANUMERICO }

INICIO_VARIABLE :: = LETRA_GRANDE | _

ENTERO :: = - DIGITO { DIGITO } | DIGITO { DIGITO }

CADENA :: = " { ELEMENTO_CADENA } "

ELEMENTO_CADENA :: = "" | NO_COMILLAS

Comentario: NO_COMILLAS es cualquier caracter diferente de ".

LETRA_CHICA :: = a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

LETRA_GRANDE :: = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

DIGITO ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

CARACTER_SIMBOLO ::= . | : | - |< | = |> | + | / | * | ? | & | $ | @ | # |¬ | \

CARACTER_SOLO ::= , | ; | !

TOKEN ::= FUNCTOR | VARIABLE | ENTERO | CADENA | SIGNO_AGRUPACION

SIGNO_AGRUPACION ::= (|) | [|] | { | } | |

COMENTARIO ::= % { NO_FIN_DE_LINEA } FIN_DE_LINEA

Comentario: FIN_DE_LINEA es el caracter LF (linefeed); NO_FIN_DE_LINEA es cualquier otro caracter.

ESPACIO_EN_BLANCO ::= CARACTER_ACOMODO

Comentario:CARACTER_ACOMODO es un espacio en blanco o un tab o FIN_DE_LINEA o cualquier caracter no imprimible (en ASCII son los caracteres con códigos ≤ 31).

ALTO_TOTAL ::= . CARACTER_ACOMODO

Observaciones:

1.- Functores mixtos no han sido descritos, pero su inclusión es directa:

termino $_N$ ::= $OP_{[xfy,fx],N}$ termino $_{N-1}$

y otras 11 combinaciones.

2.- Existen numerosas combinaciones ambiguas de operadores contiguos. Esta gramática no los contempla.

3.- No todos los functores pueden ser declarados como operadores. Nombres entre apóstrofes son considerados como functores "normales".

4.- Comentarios y espacios en blanco pueden ser insertados libremente antes y después de un token, pero no pueden ser insertados en medio de un token. Un comentario se extiende hasta el fin de línea.

5.- Un espacio en blanco debe ser insertado entre un entero sin signo y un signo menos, el cual es tratado como functor.

6.- Un término durante la entrada debe ser terminado con un ALTO_TOTAL.

La lógica matemática ha demostrado ser un formalismo robusto para el estudio de otros sistemas de conocimiento. La ciencia de la computación no ha sido la excepción, sino por el contrario, la lógica tuvo un nuevo auge al ser parte importante en el diseño de los circuitos que conforman la arquitectura de una computadora, esto es, su "inteligencia". Más recientemente, con la aparición de conceptos como el principio de resolución de Robinson y el algoritmo de unificación, se han integrado nuevos sistemas lógicos al entorno computacional, por ejemplo, la programación lógica ha sido implantada como el lenguaje de programación **PROLOG**. Aunque físicamente limitados, estos sistemas nos permiten crear programas con capacidades deductivas, tarea difícil en épocas pasadas.

Capítulo 2

Lógica y Modelo de Base de Datos Relacional

La cada vez más exigente necesidad de nuestra sociedad de manipular con rapidez y eficacia grandes cantidades de datos ha dado lugar al desarrollo de una teoría que fundamente la construcción de programas que resuelvan dicha necesidad. Los modelos de bases de datos han aparecido como la estructura de almacenamiento más eficiente de la actualidad. Varias personas han propuesto modelos de bases de datos que pretenden representar con la mayor fidelidad posible los hechos recopilados de la realidad. En la sección 2.1 conoceremos el modelo relacional de bases de datos, propuesto en 1970 por E. F. Codd [CODD70].

Para utilizar la información almacenada en una base de datos es necesario un lenguaje que nos permita enviar las órdenes de las acciones que se pretenden realizar, como consultar datos, ordenarlos, etc. El lenguaje propio del modelo relacional es descrito en la sección 2.2.

Como fue comentado, el objetivo de este trabajo es hablar de la formalización de las bases de datos desde el punto de vista de la lógica matemática, para integrar las ventajas de ambos en un marco de trabajo que permita desarrollar DBMS con mayores facultades. Dicha formalización se encuentra en la sección 2.3.

Dada la facilidad que la lógica posee para hacer inferencias, es deseable transferir esta capacidad a los DBMS, como un paso más en la evolución de dichos sistemas. La sección 2.4 nos habla de lo que es una base de datos deductiva.

2.1 Modelo Relacional

El modelo relacional de bases de datos es un formalismo matemático propuesto por E. F. Codd [CODD70] para representar, con la mayor fidelidad posible, grandes cantidades de hechos en una computadora. Conozcamos las partes que lo conforman.

2.1.1 Esquema relacional

El concepto matemático en que está basado el modelo relacional de bases de datos es la *relación* de conjuntos, que es un subconjunto del producto Cartesiano de una lista de dominios. Un *dominio* es simplemente un conjunto de valores. Por ejemplo, el conjunto de los enteros es un dominio. También lo son el conjunto de cadenas de caracteres, el conjunto de cadenas de 20 caracteres, los números reales, el conjunto {0, 1}, etc. El *producto Cartesiano* de dominios $D_1, D_2, ..., D_k$, escrito $D_1 \times D_2 \times ... \times D_k$, es el conjunto de todas las k-tuplas $(v_1, v_2, ..., v_k)$ tales que v_1 está en D_1, v_2 está en D_2, y así sucesivamente. Por ejemplo, si tenemos $k = 2$, $D_1 = \{0, 1\}$, y $D_2 = \{a, b, c\}$, entonces $D_1 \times D_2$ es { (0,a), (0,b), (0, c), (1, a), (1, b), (1, c) }.

Una *relación* es cualquier subconjunto del producto Cartesiano de uno o más dominios. Dada la naturaleza de las bases de datos, sobra el hablar de relaciones infinitas, por lo que asumiremos que una relación es finita, a menos de señalar lo contrario. Por ejemplo, { (0, a), (0, c), (1, b) } es una relación, un subconjunto de $D_1 \times D_2$ antes definido . El conjunto vacío es otro ejemplo de una relación.

Los miembros de una relación son llamados *tuplas*. Cada relación que es un subconjunto de $D_1 \times D_2 \times ... \times D_k$ se dice que tiene *aridad* k; la aridad se conoce también como *grado*. Una tupla $(v_1, v_2, ..., v_k)$ tiene k componentes; el i-ésimo componente es v_i. A menudo se abrevia $v_1 v_2 ... v_k$ para denotar la tupla $(v_1, v_2, ..., v_k)$.

En ocasiones es de utilidad ver a la relación como una tabla, donde cada renglón es una tupla y cada columna corresponde a un componente. Las columnas pueden tener nombres conocidos como *atributos*. El conjunto de los nombres de atributos de una relación es llamado el *esquema de la relación*. Si denominamos a la relación R, y su esquema de relación tiene los atributos $A_1, A_2, ..., A_k$, a menudo escribimos el esquema de relación como $R(A_1, A_2, ..., A_k)$. Por ejemplo, en la figura 2.1 vemos una relación cuyos atributos son CIUDAD, ESTADO, y HABS. La aridad de la relación es tres. Por ejemplo,

(Navojoa, Sonora, 13880)

es una tupla. El esquema de relación de esta relación es {CIUDAD, ESTADO, HABS}; si la relación se llamara INFOCIUDAD, podemos escribir el esquema de relación como INFOCIUDAD(CIUDAD, ESTADO, HABS).

CIUDAD	ESTADO	HABS
Piedras Negras	Coahuila	21870
Navojoa	Sonora	13880
Ciudad Camargo	Chihuahua	12640

Figura 2.1 Una relación.

La colección de esquemas de relación usada para representar información es llamada *esquema de base de datos* (*relacional*), y los valores actuales de las correspondientes relaciones es llamada *base de datos* (*relacional*).

2.1.2 Algebra relacional

La notación para expresar consultas a la base de datos es usualmente la parte más significativa de los lenguajes de manipulación de datos. Los aspectos que no son de consulta dentro de un lenguaje de manipulación de datos relacional son a menudo instrucciones directas de inserción, modificación y borrado de tuplas. En cambio las consultas, que en el caso más general son funciones arbitrarias aplicadas a relaciones, usan un lenguaje más rico y de más alto nivel en sus expresiones. Los lenguajes de consulta para el modelo relacional están divididos en dos clases:

- Lenguajes algebraicos, donde las consultas son expresadas aplicando operadores especializados a las relaciones, y
- Lenguajes de cálculo de predicados, donde las consultas describen un conjunto de tuplas deseado especificado mediante un predicado que las tuplas deben satisfacer.

Los lenguajes basados en cálculo se dividen a su vez en dos, según si los objetos primitivos son tuplas o elementos de un dominio de algún atributo, haciendo un total de tres clases distintas de lenguajes de consulta. Hablaremos del álgebra relacional en esta sección y en la siguiente trataremos las dos formas del cálculo relacional, llamadas cálculo relacional de tuplas y cálculo relacional de dominios.

Recordemos que una relación es un conjunto de k-tuplas donde k es la aridad de la relación. En ocasiones es conveniente dar nombres a los componentes de las tuplas, los cuales son los atributos de la relación; así como a veces es conveniente que los componentes sean anónimos refiriéndonos a ellos por nombres. Al definir el álgebra relacional asumimos que las columnas no necesitan nombre, y el orden en las tuplas es

significante. Al trabajar con relaciones como una base de datos, asumimos que todas las relaciones son finitas. Esta restricción introduce algunas dificultades en la definición del álgebra y del cálculo relacionales. Por ejemplo, no es posible la operación de complemento, ya que $\sim R$ denota generalmente una relación infinita, el conjunto de todas las tuplas que no se encuentran en R. No hay modo alguno de listar la relación $\sim R$, aunque el lenguaje de consulta permita tal expresión.

Los operandos del álgebra relacional son dos, constantes de relación y variables de relación, denotando relaciones con aridad fija. La aridad asociada con una variable será mencionada cuando sea importante. Existen 5 operaciones básicas que sirven para definir el álgebra relacional.

- *Unión*. La unión de las relaciones R y S, denotado $R \cup S$, es el conjunto de tuplas que están en R ó en S ó en ambas. Sólo se aplica el operador de unión a relaciones con igual aridad, de tal modo que todas las tuplas en el resultado tengan el mismo número de componentes.
- *Diferencia de conjuntos*. La diferencia de las relaciones R y S, denotada $R - S$, es el conjunto de tuplas en R pero que no están en S. También es necesario que ambos conjuntos tengan la misma aridad.
- *Producto Cartesiano*. Sean R y S relaciones de aridad k_1 y k_2, respectivamente. Entonces $R \times S$, el producto Cartesiano de R y S, es el conjunto de $(k_1 + k_2)$-tuplas cuyos primeros k_1 componentes forman una tupla en R y cuyos últimos k_2 componentes forman una tupla en S.
- *Proyección*. La idea de esta operación es que tomando una relación R, removamos algunos de los componentes y/o volvamos a arreglar algunos de los componentes restantes. Si R es una relación de aridad k, sea $\pi i_1 i_2 \ldots i_m(R)$, donde las i_js son distintos enteros en el rango 1 a k, la proyección de R sobre los componentes $i_1, i_2, \ldots, i_m$, esto es, el conjunto de m tuplas $a_1 a_2 \ldots a_m$ tales que existe una k-tupla $b_1 b_2 \ldots b_k$ en R para el cual $a_j = b_{ij}$ para $j = 1, 2, \ldots, m$. Por ejemplo, $\pi_{3,1}(R)$ se calcula tomando cada tupla t en R y formando una 2-tupla del tercero y del primer componente de t, en ese orden.
- *Selección*. Sea F una fórmula que involucra:
  - operandos que son constantes ó números de componentes,
  - los operadores aritméticos de comparación $<$, $=$, $>$, $\leq$, $\neq$, y $\geq$, y
  - los operadores lógicos $\wedge$ (conjunción), $\vee$ (disyunción), y $\sim$ (negación).

  Entonces $\sigma_F(R)$ es el conjunto de tuplas t en R tales que cuando, para toda i, substituimos el i-ésimo componente de t por todas las ocurrencias del número i en la fórmula F, que hagan a la fórmula F verdadera (T). Por ejemplo, $\sigma_{2>3}(R)$ denota el conjunto de tuplas en R cuya segunda componente es mayor que su tercera componente, mientras que

$\sigma_{1='López' \vee 1='Pérez'}(R)$ es el conjunto de tuplas en R cuyo primer componente contiene el valor 'López' ó 'Pérez'.

Existe además un número de operaciones útiles que pueden ser expresadas en términos de las cinco antes mencionadas, pero que han recibido nombres en la literatura y en ocasiones son usadas como operaciones primitivas.

- *Intersección.* $R \cap S$ es la abreviación de $R - (R - S)$.
- *Cociente.* Sean R y S relaciones con aridad r y s, respectivamente, donde $r > s$, y $S \neq 0$. Entonces $R \div S$ es el conjunto de $(r-s)$-tuplas t tales que para todas las s-tuplas u en S, la tupla tu está en R. Para expresar $R \div S$ usando las cinco operaciones básicas del álgebra relacional, sea $T \pi_{1,2,\ldots,r-s}(R)$. Entonces $(T \times S) - R$ es el conjunto de r-tuplas que no están en R, pero que están formadas tomando las primeras r-s componentes de una tupla en R y seguidas por una tupla en S. Entonces sea

$$V = \pi_{1,2,\ldots,r-s}((T \times S) - R)$$

  V es el conjunto de $(r-s)$-tuplas que son las primeras r-s componentes de una tupla en R tales que para alguna s-tupla u en S, tu no está en R. De aquí que $T - V$ es $R \div S$. Podemos escribir $R \div S$ como una sola expresión en el álgebra relacional reemplazando T y V por las expresiones que representan. Esto es,

$$R \div S = \pi_{1,2,\ldots,r-s}(R) - \pi_{1,2,\ldots,r-s}((\pi_{1,2,\ldots,r-s}(R) \times S) - R)$$

- *Junta.* La θ-junta de R y S en las columnas i y j se escribe $R \otimes_{i\theta j} S$ donde θ es un operador aritmético de comparación, es la abreviación de $\sigma_{i\theta(r+j)}(R \times S)$, si R es de aridad r. Esto es, la θ-junta de R y S son aquellas tuplas del producto Cartesiano de R y S tales que el i-ésimo componente de R está en relación θ con el j-ésimo componente de S. Si θ es $=$, la operación se conoce como *equijunta*.
- *Junta natural.* La junta natural, escrita $R \otimes S$, sólo es aplicable cuando ambas R y S tienen columnas etiquetadas con nombres de atributos. Para calcular $R \otimes S$ debemos
  - Calcular $R \times S$.
  - Para cada atributo A que nombra tanto a una columna de R como una columna en S seleccionar de $R \times S$ aquellas tuplas cuyos valores concuerden en las columnas para $R.A$ y $S.A$. $R.A$ es el nombre de la columna de $R \times S$ correspondiente a la columna A de R, y $S.A$ es definido análogamente.
  - Para cada atributo A proyectar la columna $S.A$.

  Entonces, formalmente, si $A_1, A_2, \ldots, A_k$ son todos los nombres de atributos usados por R y por S, la junta natural es

$$R \otimes S = \pi_{i1,i2,\ldots,im}\, \sigma_{R.A1 = S.A1 \wedge \ldots \wedge R.Ak = S.Ak}(R \times S)$$

donde $i_1, i_2, ..., i_m$ es la lista de todos los componentes de $R \times S$, en orden, excepto los componentes $S.A_1, ..., S.A_k$.

2.1.3 Dependencias funcionales

Cuando se diseña una base de datos usando el modelo relacional, nos encontramos a menudo con que debemos de elegir entre varios conjuntos de esquemas relacionales. Algunos esquemas son más convenientes que otros por varias razones. Veremos aquí algunas de las propiedades deseadas de los esquemas de relación.

Una de las ideas principales del diseño de esquemas de bases de datos es la *dependencia de datos*, esto es, una restricción sobre las posibles relaciones que pueden ser el valor actual de un esquema relacional. Por ejemplo, si un atributo determina a otro, como el atributo NOMBRE aparentemente determina al atributo DIRECCION en la relación PERSONAS, decimos que existe una *dependencia funcional* de DIRECCION en NOMBRE.

Antes de hablar del buen diseño de un esquema de base de datos, veamos por qué algunos esquemas son inadecuados. En particular hablaremos del esquema relacional

PROVEEDOR(PNOMBRE, PDIRECCION, PRODUCTO, PRECIO)

Encontramos varios problemas en este esquema:

- *Redundancia.* La dirección del proveedor se repite por cada producto.
- *Inconsistencia potencial al actualizar anomalías.* Como consecuencia de la redundancia, prodríamos actualizar la dirección del proveedor en una tupla, mientras dejamos otra sin actualizar. Así no tendríamos una dirección única como sería de esperarse.
- *Inserción de anomalías.* No podemos registrar la dirección de un proveedor si éste no nos surte por lo menos un producto. Podríamos poner valores nulos en los componentes PRODUCTO y PRECIO de la tupla del proveedor, pero entonces, cuando ingresemos un producto de dicho proveedor, ¿recordaremos borrar la tupla con valores nulos? Además, si con PNOMBRE y PRODUCTO formamos una clave única para la relación, será muy difícil o imposible buscar tuplas con valores nulos en la clave.
- *Borrado de anomalías.* La inversa del problema anterior es que si borramos todos los productos proporcionados por un proveedor, perderemos la dirección del mismo.

Todos estos problemas desaparecerían si reemplazáramos el esquema PROVEEDOR por los dos esquemas de relación siguientes

PD(PNOMBRE, PDIRECCION)
PPP(PNOMBRE, PRODUCTO, PRECIO)

El primer esquema, PD, tiene la dirección de cada proveedor exactamente una sola vez, por lo que no hay redundancia. Es más, podemos registrar la dirección de un proveedor aunque actualmente no nos surta ningún artículo. El segundo esquema, PPP, contiene los proveedores, los productos que surte y el precio que el proveedor carga por cada producto.

En muchos casos, los hechos conocidos del "mundo real" implican que no todo conjunto finito de tuplas puede ser el valor actual de alguna relación, aún si las tuplas tuvieran la aridad correcta y tuvieran elementos escogidos de los dominios correctos. Podemos distinguir dos tipos de restricciones en relaciones.

- *Restricciones que dependen de la semántica de los elementos del dominio.* Estas restricciones dependen en comprender que significan los componentes de tuplas. Por ejemplo, ninguna persona mide más de 3 metros, y ninguno tendrá 37 años de antigüedad en un empleo y 27 años de edad. Es útil que el DBMS cheque estos valores, que probablemente sean producto de algún error al capturar o procesar los datos. Más adelante hablaremos de estas "restricciones de integridad".
- *Restricciones en relaciones que dependan solamente de la igualdad o desigualdad de valores.* Existen otras restricciones que no dependen en los valores que tenga una tupla en cualquier componente dado, sino en dos tuplas que coincidan en ciertos componentes. Hablaremos de la más importante de estas restricciones para el diseño de esquemas, llamadas dependencias funcionales.

Sea $R(A_1,A_2,...,A_n)$ un esquema de relación, y sean X y Y subconjuntos de $\{A_1,A_2,...,A_n\}$. Decimos que $X \rightarrow Y$, léase "X determina funcionalmente a Y" ó "Y depende funcionalmente de X" si cualquier relación r es el valor de R, no es posible que r tenga dos tuplas que concuerden en los componentes para todos los atributos en el conjunto X y que, sin embargo, no coincidan en uno o más componentes de atributos en el conjunto Y. Por ejemplo, si R representa un conjunto de entidades cuyos atributos son $A_1,A_2,...,A_n$, y X es un conjunto de atributos que forma una clave para el conjunto de entidades, entonces podemos afirmar que $X \rightarrow Y$ para cualquier subconjunto Y de atributos. Esto es debido a que las tuplas de cada posible relación r representan entidades, y las entidades son identificadas por el valor de los atributos en la clave. Por lo tanto, dos tuplas que coincidan en los atributos del conjunto X pueden representar a la misma entidad y por lo tanto ser la misma tupla.

Debe ser enfatizado que las dependencias funcionales son enunciados sobre todas las posibles relaciones que pudieran ser valores del esquema de relación R. No podemos observar una relación particular r para el esquema R y deducir las dependencias funcionales de R. Podemos, sin embargo, ser capaces de ver alguna relación en particular para R y descubrir algunas dependencias que no tenga R.

La única manera para determinar las dependencias funcionales que posee el esquema relacional R es considerando cuidadosamente lo que significan los atributos. En este sentido, las dependencias son actualmente afirmaciones del mundo real; no pueden ser demostradas, pero podemos hacer que sean verificadas por un DBMS si se lo indicamos al diseñador de base de datos. Muchos sistemas verifican aquellas dependencias funcionales que siguen del hecho de que una clave determina los demás atributos de una relación, y algunos checan dependencias funcionales arbitrarias. Aunque existe un precio por esta eficiencia, en que el almacenamiento de cierta información se vuelve imposible. Por ejemplo, si declaramos que NOMBRE determina funcionalmente a DIRECCION, bajo ninguna circunstancia podremos almacenar dos direcciones de una misma persona en nuestra base de datos.

2.1.4 Normalización

Ha sido definido un número de propiedades o "formas normales" para esquemas de relación con dependencias. Las más significantes de éstas son conocidas como la "tercera forma normal" y la "forma normal Boyce-Codd". Estas formas normales garantizan que los problemas de redundancia y anomalías, mencionados en la sección anterior, no ocurran.

La más fuerte de estas formas normales es la Boyce-Codd. Un esquema de relación R se dice estar en la *forma normal Boyce-Codd* si siempre que $X \rightarrow A$ en R y que A no aparezca en X, entonces X es una superclave para R; esto es, X es ó contiene una clave. En otras palabras, las únicas dependencias no triviales son aquellas en que una clave determina funcionalmente uno o más de los otros atributos.

Sucede que en ciertas circunstancias la forma normal Boyce-Codd es demasiado fuerte como condición, en el sentido de que no es posible encontrar un esquema de relación en dicha forma sin perder la habilidad de preservar las dependencias. La tercera forma normal ha sido vista como una condición que tiene casi todos los beneficios de la forma normal Boyce-Codd, tantos como la eliminación de anomalías nos interesa.

Antes de definir la tercera forma normal necesitamos de una definición preliminar. Llamemos a un atributo A en la relación R un atributo *primo* si A es un miembro de cualquier llave para R (recuerde que pueden haber varias llaves). Si A no es miembro de clave alguna, entonces A es un atributo no-primo.

Un esquema de relación R se encuentra en la *tercer forma normal* si siempre que $X \rightarrow A$ en R y que A no aparezca en X, entonces X es una superclave para R ó A es primo. Nótese que las definiciones de la forma normal Boyce-Codd y de la tercer forma normal son idénticas excepto para la cláusula "ó A es primo" que hace de la tercer forma normal una condición más débil que la forma normal Boyce-Codd.

Si $X \rightarrow A$ viola la tercer forma normal, entonces ocurre uno de los dos casos siguientes:

- X es un subconjunto propio de una clave, ó
- X es un subconjunto propio de ninguna clave.

En el primer caso, decimos que $X \rightarrow A$ es una *dependencia parcial*, y en el segundo es una *dependencia transitiva*. El término "transitiva" viene del hecho de que si Y es una clave, entonces $Y \rightarrow X \rightarrow A$ es una cadena no trivial de dependencias. No es trivial debido a que sabemos que X no es un subconjunto de Y, por el segundo caso, A es tal que no se encuentra en X, y A no puede estar en Y porque A es no primo. Si R no tiene dependencias parciales, y pueda tener dependencias transitivas, decimos que R está en la *segunda forma normal*.

La forma más sencilla es la *primera forma normal*. Esta forma simplemente requiere que el dominio de cada atributo consista de valores indivisibles, no conjuntos de tuplas de valores que vengan de dominios más elementales. La primera forma normal es tan básica en el diseño de base de datos que el término "relación" puede ser visto como sinónimo de "relación en la primera forma normal".

2.1.5 Restricciones de Integridad

En cualquier DBMS completo podemos encontrar facilidades para prevenir que datos incorrectos sean almacenados en la base de datos. Existen dos fuentes de datos incorrectos: los accidentes como el teclear mal algún dato o un error en la programación, y el uso malicioso de la base de datos.

El aspecto de la *preservación de la integridad* concierne a errores accidentales y su prevención. Por ejemplo, es razonable esperar que un DBMS ofrezca facilidades para declarar que el valor en el campo EDAD no sea mayor a 150. También el DBMS puede ayudar a detectar algunos errores de programación, tales como un procedimiento que inserte un nuevo registro con los mismos valores en los campos clave que un registro que ya exista en la base de datos (asumiendo que le indicamos al sistema que no deseamos que esta inserción sea hecha). En este caso, el programa debe ser reescrito para checar si existe el registro en conflicto. Solo comentaremos que el segundo tipo de fuente de errores es manejado por la seguridad o control de acceso del DBMS.

Existen dos tipos de restricciones que un DBMS pueda checar. Un primer tipo son las dependencias funcionales vistas en la sección anterior. El segundo tipo concierne en los valores actuales almacenados en la base de datos. Típicamente estas restricciones limitan el valor de un campo a algún rango o expresan alguna relación aritmética entre varios campos. Por ejemplo, si el registro de un curso contiene los campos E%, T% Y L%, indicando los porcentajes de la calificación que poseen los examenes, las tareas y el laboratorio, debemos esperar que en cada registro la suma de los valores en estos campos es igual a 100.

Comentaremos algunos detalles de la implantación de restricciones de integridad en bases de datos cuando hablemos de la aplicación de la lógica a las bases de datos.

2.2 Lenguaje relacional.

Comenzaremos esta sección viendo el cálculo relacional de tuplas. Para facilitar las cosas, presentaremos inicialmente un cálculo que permita definir relaciones infinitas. Después veremos las modificaciones necesarias para asegurarnos que cada fórmula en el cálculo relacional denota una relación finita. Finalmente hablaremos del cálculo de tuplas y de como se relacionan cada uno de estos lenguajes. Incidentalmente, el término "cálculo relacional" no implica ninguna conexión con la rama de las matemáticas usualmente llamada "cálculo", o más precisamente, "cálculo diferencial e integral". El cálculo relacional viene del cálculo de predicados de primer orden, del campo de la Lógica.

2.2.1 Cálculo relacional de tuplas

Las expresiones del cálculo relacional de tuplas son de la forma $\{ t \mid \psi(t) \}$, donde t es una *variable de tupla*, esto es, una variable que denota una tupla de tamaño definido (usaremos $t^{(i)}$ para indicar que t tiene aridad i), y ψ es una fórmula construida por átomos y una colección de operadores a definir.

Los *átomos* de las fórmulas ψ son de tres tipos.

- $R(s)$, donde R es un nombre de relación y s es una variable de tupla. Este átomo afirma que s es una tupla en R.
- $s[i]\ \theta\ u[j]$, donde s y u son variables de tuplas y θ es un operador aritmético de comparación ($<$, $=$, etc.). Este átomo establece que el i-ésimo componente de s está en relación θ con el j-ésimo componente de u. Por ejemplo, $s[1] < u[2]$ significa que el primer componente de s es menor que el segundo componente de u.
- $s[i]\ \theta\ a$ y $a\ \theta\ s[i]$, donde θ y $s[i]$ están definidos como en el inciso anterior, y a es una constante. El primero de estos átomos afirma que el i-ésimo componente de s está en relación θ con la constante a, y el segundo tiene un significado análogo. Por ejemplo, $s[1] = 3$ significa que el valor del primer componente de s es 3.

Cuando se definen los operadores del cálculo relacional es de utilidad definir al mismo tiempo las nociones de variables de tuplas *libres* y *ligadas*. Estos conceptos son exactamente los mismos que en el cálculo de predicados. Informalmente, una ocurrencia de una variable en una fórmula esta ligada si esa variable ha sido introducida por algún cuantificador ($\forall$ ó $\exists$), de otro modo la variable está libre.

La noción de "variable libre" es análoga a aquella de las variables globales de un lenguaje de programación, esto es, una variable definida fuera del procedimiento actual. Una "variable ligada" es como una variable local, aquella que es definida dentro del procedimiento y que no puede ser referenciada fuera del mismo. En efecto, los cuantificadores del cálculo relacional desempeñan el papel de declaraciones en un lenguaje de programación.

Las *fórmulas*, y las ocurrencias *libres* y *ligadas* de variables de tuplas en estás fórmulas, son definidas recursivamente del siguiente modo.

1.- Cada átomo es una fórmula. Todas las ocurrencias de variables de tuplas enunciadas en el átomo están libres en la fórmula.

2.- Si $\psi 1$ y $\psi 2$ son fórmulas, entonces $\psi 1 \wedge \psi 2$, $\psi 1 \vee \psi 2$, y $\neg \psi 1$ son fórmulas que aseveran "$\psi 1$ y $\psi 2$ son ambas verdaderas," "$\psi 1$ ó $\psi 2$, ó ambas, son verdaderas," y "$\psi 1$ no es verdadera," respectivamente. Las ocurrencias de variables de tuplas son libres o ligadas en $\psi 1 \wedge \psi 2$, $\psi 1 \vee \psi 2$, y $\neg \psi 1$ dependiendo si son libres o ligadas en $\psi 1$ ó $\psi 2$, dependiendo de donde aparezcan. Nótese que una ocurrencia de una variable s puede estar ligada en $\psi 1$, mientras que otra ocurrencia de s esté libre en $\psi 2$, o viceversa.

3.- Si ψ es una fórmula, entonces $(\exists s)(\psi)$ es una fórmula. Las ocurrencias de s que están libres en ψ, están ligadas en $(\exists s)(\psi)$. Otras ocurrencias de variables de tuplas en ψ, incluyendo posibles ocurrencias de s que estuviesen ligadas en ψ, están libres o ligadas en $(\exists s)(\psi)$ dependiendo de su estado en ψ. La fórmula $(\exists s)(\psi)$ asevera que existe un valor de s tal que cuando substituimos dicho valor en todas las ocurrencias libres de s en ψ, la fórmula ψ es verdadera. Por ejemplo, $(\exists s)(R(s))$ indica que la relación R no está vacía, esto es, que existe una tupla s en R.

4.- Si ψ es una fórmula, entonces $(\forall s)(\psi)$ es una fórmula. Las ocurrencias libres de s en ψ están ligadas a $(\forall s)$ en $(\forall s)(\psi)$, y otras ocurrencias de variables de variables en ψ son tratadas como en el inciso anterior. La fórmula $(\forall s)(\psi)$ nos dice que substituyendo cualquier valor de tupla con la aridad apropiada en las ocurrencias libres de s en ψ, la fórmula ψ es verdadera.

5.- Podemos colocar paréntesis en las fórmulas como sea necesario. Asumimos que el orden de precedencia es: los operadores aritméticos de comparación son primero, después los cuantificadores $\exists$ y $\forall$, finalmente $\neg$, $\wedge$ y $\vee$.

6.- Ningún otra expresión es una fórmula.

Una expresión de tupla del cálculo relacional es una expresión de la forma

$$\{ t \mid \psi(t) \}$$

donde t es la única variable de tupla libre en ψ. Por ejemplo, la unión de R y S es expresada mediante

$$\{ t \mid R(t) \vee S(t) \}$$

dicho con palabras, lo anterior significa "el conjunto de tuplas t tales que t está en R o en S, o en ambos." Observe que la unión tiene sentido solamente si R y S poseen la misma aridad, y similarmente la fórmula $R(t) \vee S(t)$ tiene sentido si R y S son de la misma aridad, ya que se asumen que la variable de tupla t es de tamaño fijo.

Restringiendo el cálculo relacional para manejar relaciones finitas

El cálculo relacional de tuplas, como fue definido, nos permite definir algunas relaciones infinitas como $\{ t \mid \neg R(t) \}$, el cual denota todas las posibles tuplas que no estén en R, pero que son del tamaño asociado con t (dicho tamaño es también la aridad de R para que la expresión tenga sentido). Como no podemos imprimir todas las posibles tuplas, debemos regular dichas expresiones carentes de sentido. Lo que se hace usualmente es restringir las consideraciones a ciertas expresiones $\{ t \mid \psi(t) \}$, llamadas "seguras".

Para definir la seguridad, definamos primero DOM(ψ) como el conjunto de símbolos que tanto aparecen explícitamente en la expresión ψ, o que son componentes de alguna tupla en alguna relación R mencionada en ψ. Está elección de DOM(ψ) no es necesariamente el conjunto más pequeño de símbolos que podemos usar, pero será suficiente. Informalmente, una expresión ψ es *segura* si cada componente de cualquier tupla t que satisface a ψ debe ser un miembro de DOM(ψ).

Observe que DOM(ψ) no está determinado por la búsqueda en ψ, pero es una función de las relaciones actuales a substituir por las variables de relación en ψ. Sin embargo, como todas las relaciones se suponen finitas, DOM(ψ) es siempre finita. Por ejemplo, si $\psi(t)$ es $t[1] = a \vee R(t)$, donde R es una relación binaria, entonces DOM(ψ) es la relación unaria (para propósitos prácticos, una relación unaria es un conjunto de símbolos) dada por la fórmula del álgebra relacional $\{a\} \cup \pi 1(R) \cup \pi 2(R)$.

Decimos que una expresión del cálculo de tuplas $\{ t \mid \psi(t) \}$ es segura si

- Siempre que t satisface a ψ, cada componente de t es un miembro de DOM(ψ).
- Para cada subfórmula de ψ de la forma $(\exists u)(\omega(u))$, si ω es satisfecha por u para cualquiera valores de las otras variables en ω, entonces cada componente de u es un miembro de DOM(ω).
- Para cada subfórmula de ψ de la forma $(\forall u)(\omega(u))$, si cualquier componente de u no está en DOM(ω), entonces u satisface a ω para todos los valores de las otras variables libres en ω.

El propósito de los dos últimos incisos es de garantizar que podemos determinar la verdad de una fórmula cuantificada $(\exists u)(\omega(u))$ o $(\forall u)(\omega(u))$ considerando solamente aquellas $u$es compuestas de símbolos en DOM(ω). Por ejemplo, cualquier fórmula

$(\exists u)(R(u) \wedge \ldots)$ satisface el segundo inciso, y cualquier fórmula $(\forall u)(\sim R(u) \vee \ldots)$ satisface al tercero. Observe que en la definición de seguridad, no asumimos que cualquier variable libre de ω, además de u, tienen necesariamente valores en DOM(ω). Las reglas segunda y tercera deben ser manejadas independientemente de los valores de dichas variables.

Aunque la última regla puede parecernos poco intuitiva, debemos fijarnos que la fórmula $(\forall u)(\omega(u))$ es lógicamente equivalente a $\sim(\exists u)(\sim\omega(u))$. La última fórmula no es segura si y sólo si existe una u_0 para la cual $\sim\omega(u_0)$ sea verdadera, y u_0 no se encuentre en el dominio de la fórmula $\sim\omega$. Como los dominios de ω y $\sim\omega$ son el mismo, la tercer regla nos dice que la fórmula $(\forall u)(\omega(u))$ es segura exactamente cuando la fórmula $\sim(\exists u)(\sim\omega(u))$ sea segura.

Reduciendo el álgebra relacional al cálculo relacional de tuplas

Probaremos que el conjunto de funciones de relaciones expresables en el álgebra es exactamente el mismo que el conjunto de funciones expresables mediante fórmulas seguras del cálculo relacional de tuplas. Una dirección de esta equivalencia será demostrada ahora, y la otra dirección será demostrada después de presentar el cálculo relacional de tuplas, que es una tercera notación equivalente.

Teorema 2.1: Si E es una expresión del álgebra relacional, entonces existe una expresión segura del cálculo relacional de tuplas equivalente a E.

Demostración: Procederemos por inducción sobre el número de ocurrencias de operadores en E.

Base: Cero operadores. Entonces E es una relación constante $\{t_1,t_2,\ldots,t_n\}$ ó una variable de relación R. En el último caso, E es equivalente a $\{t \mid R(t)\}$, la cual es una expresión segura. Formalmente E es equivalente a $\{t \mid t = t_1 \vee t = t_2 \vee \ldots \vee t = t_n\}$, donde $t = t_i$ es la abreviación de $t[1] = t_i[1] \wedge \ldots \wedge t[k] = t_i[k]$; donde k es la aridad de t. Es fácil ver que $t[i]$ es uno de los conjuntos finitos de símbolos que aparecen explícitamente como el i-ésimo componente de alguna tupla constante t_j; están por lo tanto en el DOM de esta expresión.

Inducción: Supongamos que E tiene por lo menos un operador, y que el teorema es verdadero para expresiones con menos ocurrencias de operadores que las que E tiene.

Caso 1: $E = E_1 \cup E_2$. Entonces cada uno E_1 y E_2 tiene menos ocurrencias de operadores que E, y por la hipótesis inductiva podemos encontrar expresiones seguras del cálculo de tuplas $\{t \mid \varphi_1(t)\}$ y $\{t \mid \varphi_2(t)\}$ equivalentes a E_1 y E_2, respectivamente. Entonces E es equivalente a $\{t \mid \varphi_1(t) \vee \varphi_2(t)\}$. Si t satisface $\varphi_1(t) \vee \varphi_2(t)$, entonces cada componente de t está en DOM(φ_1) o en DOM(φ_2). Como DOM($\varphi_1(t) \vee \varphi_2(t)$) = Dom($\varphi_1$) $\cup$ DOM(φ_2), E es equivalente a una expresión segura. Esto es, la fórmula completa $\varphi(t) = \varphi_1(t) \vee \varphi_2(t)$ es verdadera sólo cuando t está en DOM(φ), y cualquier

subfórmula $(\exists u)(\omega(u))$ ó $(\forall u)(\omega(u))$ en ψ debe estar en ψ_1 ó en ψ_2, por lo que la hipótesis inductiva garantiza que éstas subfórmulas no violan la condición de seguridad.

Caso 2: $E = E_1 - E_2$. Entonces E_1 y E_2 tienen expresiones seguras, como en el Caso 1. Claramente E es equivalente a $\{ t \mid \psi_1(t) \wedge \neg\psi_2(t) \}$. Como $\mathrm{DOM}(\psi_1(t) \wedge \neg\psi_2(t)) = \mathrm{DOM}(\psi_1(t)) \cup \mathrm{DOM}(\psi_2(t))$, la expresión anterior es segura.

Caso 3: $E = E_1 \; E_2$. Sean E_1 y E_2 equivalentes a las expresiones seguras del Caso 1, y supongamos que E_1 y E_2 denotan relaciones con aridad k y m, respectivamente. Entonces E es equivalente a

$$\{ t^{(k+m)} \mid (\exists u)(\exists v)(\psi_1(u) \vee \psi_2(v) \\ \wedge t[1] = u[1] \wedge \ldots \wedge t[k] = u[k] \\ \wedge t[k+1] = v[1] \psi \ldots \wedge t[k+m] = v[m]) \}$$

Es fácil checar que la expresión anterior es segura, ya que $t[i]$ está restringida a valores que $u[i]$ puede tomar, si $i \leq k$, y a valores que $v[i-k]$ puede tomar si $k < i \leq k+m$.

Caso 4: $E = \pi_{i1,i2,\ldots,ik}(E_1)$. Sea E_1 equivalente a la expresión segura $\{ t \mid \psi_1(t) \}$. Entonces E es equivalente a

$$\{ t^{(k)} \mid (\exists u)(\psi_1(u) \wedge t[1] = u[i_1] \wedge \ldots \wedge t[k] = u[i_k]) \}$$

La seguridad de esta expresión se demuestra de la misma manera que en el Caso 3.

Caso 5: $E = \sigma_F(E_1)$. Sea E_1 equivalente a la expresión segura $\{ t \mid \psi_1(t) \}$. Entonces E es equivalente a $\{ t \mid \psi_1(t) \wedge F' \}$ donde F' es F con cada operador que denota el componente i remplazado por $t[i]$. Esta expresión es segura, porque cada componente de t está restringido a aquellos símbolos permitidos en los componentes de $\psi_1(t)$. ◇

2.2.2 Cálculo relacional de dominios

El cálculo relacional de dominios es construido de manera similar a la construcción del cálculo relacional de tuplas. Las diferencias esenciales son:

- No existen variables de tuplas en el cálculo de dominios, en cambio existen *variables de dominios* que representan componentes de tuplas.
- Un átomo puede ser de dos formas:
  - $R(x_1x_2\ldots x_k)$, donde R es una relación con aridad k y cada x_i es una constante o una variable de dominio, o
  - $x\,\theta\,y$, donde x y y son constantes o variables de dominio y θ es un operador aritmético de comparación.

  $R(x_1x_2\ldots x_k)$ establece que los valores de aquellas x_is que son variables deben de ser buscados tales que $x_1x_2\ldots x_k$ es una tupla en R. El significado del átomo $x\,\theta\,y$ es que x y y deben tener valores que hagan a $x\,\theta\,y$ verdadera (T).

- Las fórmulas en cálculo de dominios usan los conectivos $\wedge$, $\vee$, y $\neg$, como en el cálculo de tuplas. También usamos $(\exists x)$ y $(\forall x)$ para formar expresiones del cálculo de dominios, y x es una variable de dominio, no de tupla.

Las nociones de variables de dominio libres y ligadas y el alcance de una variable ligada son definidas en el cálculo de dominios exactamente igual que en el cálculo de tuplas. Una expresión del cálculo de dominios es de la forma $\{ x_1x_2...x_k \mid \psi(x_1,x_2,...,x_k) \}$, donde ψ es una fórmula cuyas únicas variables de dominio libres son las variables $x_1,x_2,...,x_k$.

En analogía con el cálculo de tuplas, definimos que una expresión del cálculo de dominios $\{ x_1x_2...x_k \mid \psi(x_1,x_2,...,x_k) \}$ es segura si

- $\psi(x_1,x_2,...,x_k)$ implica verdaderamente que x_i está en DOM(ψ).
- Si $(\exists u)(\omega(u))$ es una subfórmula de ψ, entonces $(\omega(u))$ es verdadera para cualquiera valores de las variables libres de ω (además de u) e implica que u está en DOM(ω).
- Si $(\forall u)(\omega(u))$ es una subfórmula de ψ, entonces $\omega(u)$ es falsa para cualquiera valores de las variables libres de ω (además de u) e implica que u está en DOM(ω).

Reduciendo el cálculo de tuplas al cálculo de dominios

La construcción de una expresión del cálculo de dominios equivalente a una expresión del cálculo de tuplas $\{ t \mid \psi(t) \}$ es directa. Si t tiene aridad k, hay que introducir k nuevas variables de dominio $t_1,t_2,...,t_k$ y reemplazarlas en la expresión por

$$\{ t_1t_2...t_k \mid \psi'(t_1,t_2,...,t_k) \}$$

donde ψ' es ψ con cualquier átomo $R(t)$ reemplazado por $R(t_1t_2...t_k)$, y cada ocurrencia libre de t [i] reemplazado por t_i. Observe que pueden haber ocurrencias ligadas de t dentro de ψ, si existiera un cuantificador $(\exists t)$ o $(\forall t)$; los usos de esta t se refieren a una variable de tupla "diferente" y no será reemplazada (es una situación análoga a las variables locales y globales que poseen el mismo identificador, en los lenguajes de programación con estructura de bloques).

A continuación, para cada cuantificador $(\exists u)$ o $(\forall u)$, si u tiene aridad m, hay que introducir m nuevas variables de dominio $u_1,u_2,...,u_m$ y dentro del alcance de esta cuantificación de u, reemplazar $u[i]$ por u_i y $R(u)$ por $R(u_1u_2...u_m)$. Reemplazar $(\exists u)$ por $(\exists u_1)...(\exists u_m)$ y $(\forall u)$ por $(\forall u_1)...(\forall u_m)$. El resultado es una expresión en el cálculo de dominios equivalente a la expresión original del cálculo de tuplas.

Debe quedar claro que los valores que pudiera tomar t_i son exactamente aquellos que pudiera tomar $t[i]$ en la expresión original. Por lo tanto si $\{ t \mid \psi(t) \}$ es segura, entonces es la expresión resultante en el cálculo de dominios. Con todo esto podemos establecer el siguiente teorema.

Teorema 2.2: Para cada expresión segura del cálculo relacional de tuplas existe una expresión segura equivalente del cálculo relacional de dominios.

Reduciendo el cálculo de dominios al álgebra relacional

La idea es tomar cualquier fórmula segura $\psi(x_1,x_2,...,x_k)$ del cálculo de dominios, con variables libres $x_1, x_2,..., x_k$, y construir, por inducción en el número de operadores en ψ, una expresión algebraica cuyo valor es $\{ x_1x_2...x_k \mid \psi(x_1,x_2,...,x_k) \}$. Para hacerlo, probaremos por inducción sobre el tamaño (número de operadores) de una subfórmula ω de ψ con variables libres $y_1, y_2,..., y_m$, tal que sea una expresión del álgebra relacional para

$$\mathrm{DOM}(\psi)^m \cap \{ y_1y_2...y_m \mid \omega(y_1,y_2,...,y_m) \}$$

donde D_m denota $D \times D \times ... \times D$, ($m$ veces).

Note que por conveniencia, restringiremos nuestra consideración al DOM(ψ) para todas las subfórmulas ω de ψ, aunque difícilmente nos hayamos restringido los varios DOM(ω)s, cada uno de los cuales debe ser un subconjunto de DOM(ψ).

Los siguientes lemas serán de utilidad para la demostración.

Lema 2.1: Si ψ es cualquier fórmula en el cálculo de dominios (o del cálculo de tuplas) entonces existe una expresión en el álgebra relacional que denota la relación unaria (el conjunto) DOM(ψ).

Demostración: Si R es una relación de aridad k, sea

$$E(R) = \pi_1(R) \cup \pi_2(R) \cup ... \cup \pi_k(R)$$

Entonces la expresión deseada es la unión de $E(R)$, para cada variable de la relación R que aparezca en ψ, y la relación constante $\{ a_1, a_2,..., a_n \}$, donde cada una de las a_is son todos los símbolos constantes que aparecen en ψ. ◊

Lema 2.2: Si ψ es cualquier fórmula en el cálculo de dominios (o del cálculo de tuplas) entonces existe una fórmula equivalente ψ' del cálculo de dominios (respectivamente del cálculo de tuplas) sin ocurrencias de $\wedge$ ó $\forall$. Si ψ es segura, entonces ψ' lo es.

Demostración: Reemplace cada subfórmula $\psi_1 \wedge \psi_2$ de ψ por $\neg(\neg\psi_1 \vee \neg\psi_2)$ (Ley de De Morgan). Después, reemplace cada subfórmula $(\forall u)(\psi_1(u))$ por $\neg(\exists u)(\neg\psi_1(u))$. Esta transformación también preserva la equivalencia y nos dice en esencia que ψ_1 es verdadera para toda u si y sólo si no existe una u para la cual ψ_1 sea falsa.

Sea ψ' la fórmula resultante. Seguramente ψ' es equivalente a ψ. Si ψ es segura, entonces para cada subfórmula $(\forall u)(\psi_1(u))$ sabemos que $\psi_1(u)$ es verdadera siempre que u tenga un valor fuera del conjunto DOM(ψ_1). De aquí que $\neg\psi_1(u)$ es falsa siempre que u está

fuera de DOM(ψ_1), que es igual a DOM($\neg\psi_1$). Así la subfórmula introducida $(\exists u)(\neg\psi_1(u)$ de ψ satisface la condición de seguridad. $\diamond$

Teorema 2.3: Para cada expresión segura del cálculo relacional de dominios existe una expresión equivalente en el álgebra relacional.

Demostración: Sea $\{x_1...x_k \mid \psi(x_1,...,x_k)\}$ una fórmula segura del cálculo de dominios. Por el Lema 2.2 asumimos que ψ sólo tiene los operadores $\vee$, $\neg$, y $\exists$. Por el Lema 2.1 podemos tomar a E como una expresión del álgebra relacional para el conjunto DOM(ψ), y como es usual, definamos a E^k como $E \times E \times ... \times E$ (k veces). Probaremos por inducción en el número de operandos en una subfórmula ω de ψ que si ω tiene variables de dominio libres $y_1,y_2,...,y_m$, entonces

$$\text{DOM}(\psi)^m \cap \{ y_1...y_m \mid \omega(y_1,...,y_m)\}$$

tiene una expresión equivalente en el álgebra relacional. Entonces, como un caso especial, cuando ω es la misma ψ, tenemos una expresión algebraica para

$$\text{DOM}(\psi)^k \cap \{ x_1...x_k \mid \psi(x_1,...,x_k)\}$$

Ya que ψ es segura, la intersección con DOM(ψ)^k no cambia la relación denotada, con lo que habremos demostrado el teorema. Vayamos a la prueba por inducción.

Base: Cero operadores en ω. Entonces ω es un átomo, el cual podemos tomar sin perder generalidad en una de las formas $x_1\ \theta\ x_2$, $x_1\ \theta\ x_1$, $x_1\ \theta\ a$, ó $R(x_{i1}x_{i2}...x_{il})$,donde θ es un operador aritmético de comparación, y a es una constante. Si el átomo es $x_1\ \theta\ x_2$, entonces la expresión algebraica deseada es $\sigma_{1\ \theta\ 2}(E \times E)$. Los átomos de las formas $x_1\ \theta\ x_1$ y $x_1\ \theta\ a$ son manejados de manera similar.

Finalmente, si el átomo es $R(x_{i1}\ x_{i2}\ ...\ x_{il})$ construiremos la expresión

$$\pi_{j\ 1,j\ 2,...,\ jk}\ (\sigma_F(R))$$

donde F es una fórmula que tiene términos $u = v$ siempre que x_{iu} y x_{iv} sean la misma variable, y $u < v$; todos los términos están conectados por el operador $\wedge$. La lista $j_1,j_2,...j_k$ es cualquier lista tal que $x_{ij\ 1} = x_1,..., x_{ijk} = x_k$. Por ejemplo, si ω es $R(x_2x_1x_2x_3)$, entonces nuestra expresión es $\pi_{2,1,4}\ (\sigma_{1=3}(R))$.

Inducción: Supongamos que ω tiene por lo menos un operador y que la hipótesis inductiva es verdadera para todas las subfórmulas de ψ que tengan menos operadores que ω.

Caso 1: $\omega(y_1,...,y_m) = \omega_1(u_1,...,u_n) \vee \omega_2(v_1,...,v_p)$, donde cada u_i es una distinta y_j y cada v_i en una diferente y_j (aunque algunas de las us y vs pueden ser la misma y_j). Sea E_1 una expresión algebraica para

$$\text{DOM}(\psi)^n \cap \{ u_1...u_n \mid \omega_1(u_1,...,u_n) \}$$

y E_2 una expresión algebraica para

$$\mathrm{DOM}(\psi)^p \cap \{ v_1 \ldots v_p \mid \omega_2(v_1,\ldots,v_p) \}$$

Definamos E'_1 por

$$E'_1 = \pi_{i_1,\ldots,\ i_m}(E_1 \times E^{n-n})$$

donde i_l es aquella q tal que $u_q = y_l$ si tal u_q existe, de otro modo i_l es un único entero entre $n + 1$ y m. Similarmente definimos

$$E'_2 = \pi_{j_1,\ldots,\ j_m}(E_2 \times E^{m-p})$$

donde j_l es aquella q tal que $v_q = y_l$ si tal v_q existe, de otro modo j_l es un único entero entre $p + 1$ y m. Entonces la expresión deseada es $E'_1 \cup E'_2$.

Por ejemplo, si $\omega(y_1,y_2,y_3,y_4)$ es

$$\omega_1(y_1,y_3,y_4)\ \omega_2(y_2,y_4)$$

entonces

$$E'_1 = \pi_{1,4,2,3}\ (E_1 \times E)$$

y

$$E'_2 = \pi_{3,1,4,2}\ (E_2 \times E \times E)$$

El hecho de que la fórmula $E'_1 \cup E'_2$ esté correcta parte del hecho que E'_1 denota a $\mathrm{DOM}(\psi)^m \cap \{ y_1 \ldots y_m \mid \omega_1(u_1,\ldots,u_n) \}$ y E'_2 denota a $\mathrm{DOM}(\psi)^m \cap \{ y_1 \ldots y_m \mid \omega_2(v_1,\ldots,v_p) \}$ (Recuerde que cada una de las us y vs es una de las ys). De aquí se sigue que $E'_1 \cup E2'$ denota $\mathrm{DOM}(\psi)^m \cap \{ y_1 \ldots y_m \mid \omega(y_1,\ldots,y_m) \}$.

Caso 2: $\omega(y_1,\ldots,y_m) = \neg\omega_1(y_1,\ldots,y_m)$. Sea E_1 una expresión algebraica para $\mathrm{DOM}(\psi)^m \cap \{ y_1 \ldots y_m \mid \omega_1(y_1,\ldots,y_m) \}$. Entonces $E^m - E_1$ es una expresión para $\mathrm{DOM}(\psi)^m - \{ y_1 \ldots y_m \mid \omega_1(y_1,\ldots,y_m) \}$, la cual es equivalente a

$$\mathrm{DOM}(\psi)^m \cap \{ y_1 \ldots y_m \mid \neg\omega_1(y_1,\ldots,y_m) \}$$

Observe que $\{ y_1 \ldots y_m \mid \neg\omega_1(y_1,\ldots,y_m) \}$ se ve como un conjunto infinito, pero podemos obtener de él lo que necesitamos intersectándolo con el conjunto finito $\mathrm{DOM}(\psi)^m$.

Caso 3: $\omega_1(y_1,\ldots,y_m) = (\exists y_{m+1})(\ \omega_1(y_1,\ldots,y_{m+1})\ \}$. Sea E_1 una expresión algebraica para $\mathrm{DOM}(\psi)^{m+1} \cap \{ y_1 \ldots y_{m+1} \mid \omega_1(y_1,\ldots,y_{m+1})\ \}$. Ya que ψ es segura, y por lo tanto ω lo es, $\omega_1(y_1,\ldots,y_{m+1})\ \}$ no será verdadera a menos que y_{m+1} esté en el conjunto $\mathrm{DOM}(\omega)$, el cual es un subconjunto de $\mathrm{DOM}(\psi)$. Por lo tanto $\pi_{1,2,\ldots,\ m}(E_1)$ denota la relación $\mathrm{DOM}(\psi)^m \cap \{ y_1 \ldots y_m \mid (\exists y_{m+1})\ (\omega_1(y_1,\ldots,y_{m+1})\ \}$, el cual completa la inducción y demuestra el teorema. ◇

2.3 Aplicación de la Lógica a las Bases de Datos

Antes de considerar la formalización de bases de datos en términos de lógica, debemos mencionar algunas suposiciones que rigen la evaluación de consultas (y las

restricciones de integridad) de bases de datos. Por un lado están las suposiciones que expresan cierta representación de hechos negativos. Por ejemplo, "Pablo no es el papá de Pedro" se enunciaría ~ Padre (Pablo, Pedro). Y por otro lado las suposiciones que hacen preciso el universo de referencia que las consultas implican. Debido a lo expuesto anteriormente existen tres suposiciones:

- La *suposición del mundo cerrado (Closed World Assumption)*, también llamada convención de información negativa, la cual establece que hechos no conocidos que sean ciertos se supone que son falsos (i.e.: ~ R (e_1,..., e_n) se supone que es cierta si y sólo si la tupla (e_1,..., e_n) no se encuentra en la relación R).
- La *suposición del nombre único*, la cual dice que individuos con nombres diferentes son diferentes.
- La *suposición de la cerradura del dominio*, que establece: no existen otros individuos más que aquellos que se encuentran en la base de datos.

Respuestas a consultas ∀ ó consultas que involucran negación son obtenidas usando las hipótesis anteriores Por ejemplo, la pregunta "¿Quién no es profesor de tiempo completo?" hecha a una base de datos cuyo estado actual es

Profesor_completo (Juan)
Profesor_completo (Pablo)
Profesor_adjunto (Andrés)
Profesor_adjunto (Pedro)

obtiene la respuesta {Pedro, Andrés}.

Evidentemente, la suposición de la cerradura del dominio restringe a los individuos a considerar al conjunto {Juan, Pablo, Andrés, Pedro}. Además, de acuerdo con la suposición del nombre único, se obtiene lo siguiente: Pedro ≠ Juan, Pedro ≠ Pablo. Consecuentemente, Pedro ∉ Profesor_completo, el cual, de acuerdo con la suposición del mundo cerrado, lleva a ~ Profesor_completo (Pedro). El segundo elemento de la respuesta, ~ Profesor_completo (Andrés), es obtenido del mismo modo.

Observamos que una manera de evitar llamadas a la suposición de la cerradura de dominio es considerar como consultas (y restricciones de integridad) aceptables sólo a expresiones que restrinjan su propio dominio de referencia. Este es el caso para cualquier expresión del álgebra relacional y de las así llamadas fórmulas lógicas de clase de definido (ó de rango restringido).

Aunque el proceso de evaluación de consultas en cualquier DBMS trabaja (implícitamente) bajo las hipótesis anteriores, estas suposiciones pueden ser explícitas y claramente comprensibles sólo a través de una formalización lógica de las bases de datos.

Una base de datos desde el punto de vista de la lógica se ve de diferentes maneras: como una interpretación (de una teoría de primer-orden) o como una teoría (de primer orden).

Desde el punto de vista de interpretaciones, las consultas (y las restricciones de integridad) son fórmulas que serán evaluadas en la interpretación usando la definición semántica de verdad. Del punto de vista de una teoría, las consultas y las restricciones de integridad son consideradas como teoremas que deben ser demostrados. Los puntos de vista de interpretación y de teoría formalizan los conceptos de base de datos convencionales y deductivas, respectivamente. Reiter [REIT84] y Kowalski [KOWA81] han investigado estos dos enfoques con profundidad. Reiter se refiere a los dos enfoques como la "vista de modelo teórico" y la "vista de demostración teórica", respectivamente. Mientras que Kowalski se refiere a ellas como la "vista de estructura relacional" y la "vista de base de datos lógica".

Los términos "interpretación", "modelo", y "estructura relacional" están muy relacionados. Un modelo es una interpretación que hace verdaderos a todos los axiomas. La vista de "estructura relacional" significa que las consultas son evaluadas asumiendo que las entradas de la base de datos son verdaderas. Todos estos términos se relacionan con la definición semántica de verdad. Los tres términos "teoría", "demostración teórica" y "base de datos teórica" connotan que, para determinar respuestas a consultas, uno deriva datos de axiomas.

Tanto Kowalski como Reiter han demostrado que, aunque las bases de datos convencionales son generalmente consideradas desde el punto de vista de modelos, también pueden ser consideradas desde el punto de vista de teorías y puede así ser considerada como una base de datos lógica particular.

Veamos ahora una caracterización intuitiva de estos dos enfoques de una base de datos a través de la lógica.

Sea DB una instancia de una base de datos relacional. Entonces DB consiste de un conjunto de relaciones (i.e., una relación R para cada esquema de relación $R(A_1,..., A_n)$) y un conjunto de restricciones de integridad IC. Sea D la unión de todos los dominios subordinados de todos los atributos que existen en el esquema de la relación. Ahora definamos un lenguaje de primer orden L que consiste de un símbolo de predicado de n lugares REL para cada relación n-aria en DB y un conjunto de constantes, una por cada elemento en D; se asume que el lenguaje no tiene símbolos de función. DB puede ser visto como una interpretación de fórmulas del lenguaje como fue definido en la sección 1.2, y las fórmulas de L pueden ser evaluadas en esta interpretación del siguiente modo: el rango de variables sobre el dominio D, y $REL(e_1,...,e_n)$ es verdadero si y sólo si $(e_1,...,e_n) \in R$. El lenguaje puede ser extendido para incluir operadores aritméticos de comparación ($<$, $=$, $>$, $\leq$, $\geq$) como símbolos de predicado especiales, a los cuales se les asigna su interpretación usual.

Si las restricciones de integridad en IC son expresadas como fórmulas de L, entonces la base de datos DB será un estado válido de base de datos si y sólo si cada restricción en IC es verdadera en DB. Esto significa que si y sólo si DB es un modelo de IC. Observemos que, de acuerdo con la definición de una interpretación, la evaluación de fórmulas lógicas (en una interpretación) es hecha de acuerdo con las suposiciones del mundo cerrado, nombre único y cerradura del dominio establecidos al principio de esta sección.

Lo anterior constituye una descripción de una base de datos del punto de vista de modelo teórico. El punto de vista de demostración teórica de DB se obtiene construyendo una teoría T que admita a DB como único modelo. Entonces para cualquier fórmula bien formada w en L, w es derivable de T si y sólo si w es verdadera en DB.

El proceso de definir T consiste en hacer sus axiomas (propios) precisos, determinados como Aseveraciones, Axiomas de Particularización y Axiomas de Igualdad:

- *Aseveraciones*. Para cualquier relación R en DB y cualquier tupla $(e_1,...,e_n) \in R$, un axioma $REL(e_1,...,e_n) \in T$.
- *Axiomas de particularización*. Establecen explícitamente la hipótesis de evaluación que desde el punto de vista de modelo teórico son llevados mediante interpretación:
  - *Axiomas de terminación*. Existe un axioma para cualquier relación R en DB. Si $(e1^1,...,ei^n),...,(ep^1,...,ep^n)$ son todas las tuplas en R, esto es escrito como:

    $$\forall x_1...\forall x_n\ (R(x1, ..., x_n) \rightarrow (x_1 = e_{11} \wedge ... \wedge x_n = e_{1n}) \vee ... \vee (x_1 = e_{p1} \wedge ... \wedge x_n = e_{pn}))$$

    El axioma de terminación establece efectivamente que las únicas tuplas de valores que la relación R puede tener son $(e_{11},...,e_{1n}),...,(e_{p1},...,e_{pn})$.
  - *Axiomas de nombre único*. Si $e_1,...,e_q$ son todos los individuos en DB, los axiomas de nombre único son

    $(e_1 \neq e_2),...,(e_1 \neq e_q)$
    $(e_2 \neq e_3),..., (e_{q-1} \neq e_q)$.
  - *Axioma de la cerradura de dominio*. Este es:

    $$\forall x\ ((x = e_1) \vee (x = e_2) \vee ... \vee (x = e_q)).$$
- *Axiomas de igualdad*. Son necesarios pues los axiomas de particularización involucran el predicado de igualdad, requiriendo de la especificación de las propiedades:
  - *reflexiva*: $\forall x\ (x = x)$

- *simétrica*: $\forall x \forall y (x = y) \rightarrow (y = x)$
- *transitiva*: $\forall x \forall y \forall z (x = y) \wedge (y = z) \rightarrow (x = z)$
- *principio de substitución de términos iguales*:
  $\forall x1 \ldots \forall y_n (P(x_1,\ldots,x_n) \wedge (x_1 = y_1) \wedge \ldots \wedge (x_n = y_n) \rightarrow P(y_1,\ldots,y_n))$
  Cabe observar que el símbolo $\wedge$ es la proposición conjuntiva.

Veamos brevemente la razón de porqué T admite a DB como único modelo (como un isomorfismo). La única interpretación de T en la cual los axiomas de cerradura de dominio y de nombre único son todos satisfechos es aquella que sus individuos están en correspondencia uno-a-uno con elementos en D. Así todos los modelos posibles de T tienen el mismo modelo como DB (a manera de un isomorfismo). Para cualquier modelo M, ya que su dominio es ajustado, para ser diferente de DB, se le debe de asignar por lo menos a un predicado REL una relación R' diferente de R. Pero esto no es posible. Verdaderamente, si una tupla $(e_1, \ldots, e_n)$ pertenece a R pero no a R', entonces M no satisface una de las aseveraciones. Contrariamente, si $(e_1\ldots,e_n)$ pertenecen a R', pero no a R, entonces M no cumple con el axioma de terminación asociado a REL.

Como fue definido antes, T ofrece el punto de vista de demostración teórica de DB. De acuerdo a este punto de vista, DB satisface la restricción w en IC si y sólo si $T \mid\!- w$. Además la respuesta a una consulta formulada como $W(x_1,\ldots,x_p)$ -donde $x_1,\ldots,x_p$ son las variables libres en la fórmula W-consiste de aquellas p-tuplas $(e_1,\ldots,e_p)$tales que $T \mid\!- W(e_1,\ldots,e_p)$.

Es conveniente observar que, aún de acuerdo a esta vista las evaluaciones de consultas (y de restricciones de integridad) mediante técnicas de demostración; la DB sigue siendo una base de datos convencional (i.e., no deductiva). No existen otros hechos (positivos) que aquellos establecidos explícitamente en las aseveraciones que puedan ser derivados de T.

Hasta ahora podemos observar que el punto de vista de demostración teórica no tiene el propósito de ser usado directamente como base para la implantación de un DBMS. La complejidad combinatorial de los axiomas de particularización llevaría a sistemas ineficientes, pero, sin embargo Reiter [REIT84] enfatiza que el valor de este punto de vista es la generalización para las bases de datos considerando los siguientes puntos:

- Añadir algunos hechos disyuntivos o literales cuantificadas existencialmente entre las suposiciones y se obtiene una base de datos con valores nulos e información incompleta.

- Suprimir del conjunto de restricciones de integridad algunas de sus fórmulas y añadirlas a la teoría como axiomas y se obtiene una nueva teoría que es una base de datos deductiva.

Excepto para el trabajo en bases de datos deductivas, las aplicaciones de la lógica a las bases de datos se refieren principalmente, ya sea implícita o explícitamente, al punto de vista de modelos.

2.3.1 Vista de sintaxis y semántica de Lógica y el Lenguaje Relacional

Sintaxis

El lenguaje relacional L se obtiene adaptando la sintaxis del cálculo de predicados de primer orden al esquema de base de datos DB del siguiente modo:

- Las constantes individuales de L son finitas en número; el conjunto de constantes es la unión de los dominios de DB.
- No existen funciones (excepto para constantes individuales, vistas como funciones sin argumentos).
- El número de predicados de L es finito.
- Entre los predicados de L, existe un predicado binario distinguible, llamado igualdad (denotado $=$).
- Cada predicado en L aparte de la igualdad está asociado con un esquema de relación en DB y viceversa. Cada atributo en el esquema de relación corresponde con un argumento del predicado asociado. Tal predicado es llamado un predicado de relación. Para simplificar, el mismo nombre es dado tanto al predicado de relación como a la relación asociada.
- Las fórmulas del lenguaje relacional son construidas como aquellas del cálculo de predicados.

A partir del lenguaje relacional L se puede definir un lenguaje de consultas, considerando que una *consulta* se define como una expresión de la forma $\{ x \mid Q(x) \}$ donde $x = (x_1,...,x_n)$ es una n-tupla de variables llamadas *variables objetivo* o *variables de consulta*. $Q(x)$ es una fórmula del lenguaje L donde las únicas variables libres pertenecen a la n-tupla x. En particular, consultas con respuestas afirmativas o negativas exclusivamente son de la forma

$$\{ \mid Q \}$$

donde Q es una fórmula cerrada (sin variables libres).

Semántica

Como en el cálculo de predicados, la semántica del lenguaje relacional L involucra la definición de intrepretaciones y modelos de las fórmulas de L.

Una interpretación de L es una tupla $I = (D, I_C, I_V)$ donde

- D es un conjunto no vacío de constantes,
- I_C es una función que asocia un elemento de D con cada constante de L y un mapeo de D^n sobre {T, F} con cada predicado n-ario de L exceptuando la igualdad.
- I_V es una función que asocia un elemento de D con cada variable de L.

La igualdad es interpretada en D como identidad, como en el cálculo de predicados.

Cada fórmula de L tiene, en la manera usual, un valor de verdad en cada interpretación I de L. Si una fórmula es verdadera en una interpretación dada, entonces se dice que la interpretación es un modelo de la fórmula.

Una *interpretación de Herbrand* de un conjunto F de fórmulas es la asignación de un valor de verdad a cada fórmula de la base de Herbrand B_F. Por extensión, el subconjunto de atomos en B_F con valor de verdad T en la interpretación es también llamada la interpretación de Herbrand de F.

En la práctica, entre todas las interpretaciones de L, las más interesantes son las interpretaciones de Herbrand que están en correspondencia directa con extensiones de bases de datos cuyo esquema DB está asociado con L. En particular, como vimos anteriormente, la consulta $\{x \mid Q(x)\}$ caracteriza un conjunto de tuplas c de constantes para cuyo $Q(c)$ evalúa verdadero en una cierta interpretación de L o, equivalentemente, para cuyo $Q(c)$ es un teorema en una cierta teoría de L.

2.3.2 Lógica y Restricciones de Integridad

El modelo relacional posee gran independencia de datos permitiendo al usuario pensar en términos de conjuntos y relaciones en lugar de arreglos, apuntadores, bits, etc. Pero en la actualidad está muy restringido por la incapacidad de almacenar datos implícitos. Se usan reglas generales en forma de restricciones de integridad pero no sirven del todo para definir una relación.

Es sencillo pensar en ejemplos donde se vea la conveniencia del uso de reglas generales para definir una relación. Por ejemplo, el hecho de que los alumnos de sexto semestre de la carrera de Matemáticas Aplicadas y Computación deben de haber acreditado los dos cursos de Comprensión de Lectura del idioma inglés puede ser representado más concisamente mediante una regla general en lugar de almacenar los datos explícitamente, para cada estudiante de sexto semestre, el hecho de que debió de haber cursado Comprensión de Lectura. Cada uno de los hechos específicos pueden ser deducidos a partir de la regla general.

Las reglas generales también son útiles para evitar redundancias respecto a actualizaciones en los datos. Considérese, por ejemplo, una relación que pueda ser

definida en términos de otras dos relaciones. Es mejor establecer ésto a través de reglas generales que calculando y almacenando la relación explícitamente. Por ejemplo, si una relación llamada "padres" es definida de un modo general mediante las relaciones "padre" y "madre", una actualización en cualquiera de estas dos relaciones no implica una actualización en la relación "padres".

2.4 Deducción y Base de Datos Deductiva

La necesidad de capacidad deductiva ha sido reconocida por diseñadores de sistemas relacionales, y se han realizado esfuerzos para añadir un componente deductivo capaz de manejar información en general. Desde este enfoque, la lógica deberá ser usada como componente deductivo, concebido como un módulo separado hecho para datos específicos. Las relaciones entre ambos componentes son comúnmente disjuntas. Aunque las relaciones que contienen reglas generales y algunas excepciones (restricciones de integridad) no son, como pudiera pensarse, del todo extrañas. Un ejemplo de esta situación es el siguiente: para representar la relación de los estudiantes de quinto semestre y la condición de estar cursando la materia de Ecuaciones Diferenciales y suponiendo que Cortés es un alumno de sexto semestre que no ha cumplido con el requisito, la relación binaria "cursa" (el estudiante x cursa la materia y) puede ser representada a través de los siguientes predicados:

cursa (x,ecuaciones_diferenciales) ← semestre(x,5)
cursa(Cortés,ecuaciones_diferenciales) ←

Además las reglas generales pueden ser recursivas. Por ejemplo, para la relación de "padres" debemos definir la relación "ancestro" del siguiente modo:

ancestro(x,y) ← padres(x,y)
ancestro(x,y) ← padres(x,z), ancestro(z,y).

Otro aspecto interesante en la descripción de información a través de la lógica surge del hecho de que la forma clausular del cálculo de predicados de primer orden, como fue descrito en el capítulo anterior, se ha demostrado que tiene interpretación operacional y declarativa, esto es, un conjunto de cláusulas puede ser visto como un programa lógico que puede ser interpretado (ejecutado) mediante un demostrador de teoremas basado en resolución.

Una *base de datos deductiva* (DDB) es una base de datos en la cual hechos nuevos pueden ser derivados de hechos que fueron introducidos de manera explícita. Desde el punto de vista de demostración teórica estas bases de datos son consideradas como una teoría especial de primer orden. Formalmente, consideremos una base de datos consistente de un conjunto finito de constantes $\{c_1 \ldots c_n\}$, y un conjunto de cláusulas de primer orden sin símbolos de función. Las funciones son excluidas para tener respuestas finitas y explícitas de consultas. Inicialmente la teoría excluye valores que aparecen en una base de datos cuando se tienen enunciados como $(\exists x)\ P(a,x)$. Esto

significa que "*a*" en el predicado *P* tiene un valor asociado, pero precisamente ese valor es desconocido. Como vimos en el capítulo anterior, cuando se skolemiza la fórmula $(\exists x)\ P(a,x)$ y se escribe en forma clausular, se obtiene la cláusula $P(a,w)$, donde w es una constante Skolem (i.e., una constante cuyo valor es de otro modo irrestricto).

La forma general de cláusulas que representarán hechos y reglas deductivas es

$$P_1 \wedge P_2 \wedge \ldots \wedge P_k \rightarrow R_1 \vee \ldots \vee R_q.$$

Es equivalente a la cláusula

$$\neg P_1 \vee \ldots \vee \neg P_k \vee R_1 \vee \ldots \vee R_q.$$

La conjunción de las P_i se refiere al lado izquierdo de la cláusula y la disyunción de las R_j al lado derecho.

Ya que las cláusulas consideradas son libres de funciones, los términos que son los argumentos de los P_i y los R_j son constantes o variables. Siempre que cualquier variable que aparezca en el lado derecho de una cláusula aparece también en el lado izquierdo, se dice que la cláusula tiene rango restringido. Podemos considerar varios tipos de cláusulas dependiendo de los valores de k y q, como en [MINK83]:

Tipo 1: $k = 0$, $q = 1$. Las cláusulas tienen la forma

$$\rightarrow P(t_1, \ldots, t_m).$$

a) Si las t_i son constantes, $c_{i1}, \ldots, c_{im}$, entonces tenemos

$$\rightarrow P(c_{i1}, \ldots, c_{im}),$$

que representa una aseveración o un hecho en la base de datos. El conjunto de todas estas aseveraciones para el predicado *P* corresponde a una relación en la base de datos. La flecha que aparece a la izquierda es generalmente omitida.

b) Cuando algunas, o todas las t_i son variables, la cláusula corresponde a un enunciado general en la base de datos. Por ejemplo,

$$\rightarrow \text{ancestro}(\text{Adán}, x),$$

establece que Adán es un ancestro de todos los individuos en la base de datos (suponemos que la base de datos contiene únicamente seres humanos). Claro que este tipo de datos, que son cláusulas sin rango restringido, que asumen que todos los individuos en la base de datos son del mismo tipo aparece muy raramente.

Tipo 2: $k = 1$, $q = 0$. Son cláusulas de la forma

$$P(t_1, \ldots, t_m) \rightarrow.$$

a) Cuando todas las t_i son constantes, entonces tenemos

$$P(c_{i1}, \ldots, c_{im}) \rightarrow,$$

que establece un hecho negativo. Los enunciados negativos peculiares ya que las bases de datos relacionales no contienen datos negativos.

b) Cuando algunas de las t_i son variables hay dos casos, puede ser vista como una restricción de integridad (como una cláusula particular del Tipo 3, ver adelante) o como el significado "el valor no existe" para valores nulos.

Tipo 3: $k > 1$, $q = 0$. Son cláusulas de la forma

$$P_1 \wedge \ldots \wedge P_k \rightarrow .$$

Estos axiomas pueden ser vistos como restricciones de integridad. Esto es, los datos que serán añadidos a una base de datos deben satisfacer las leyes especificadas por la condición de integridad para ser integrados a la base de datos. Por ejemplo, uno puede especificar una ley de integridad que establezca que "ningún individuo puede ser padre y madre de otro individuo". Esto puede ser descrito como

$$\text{Padre}(x,y) \wedge \text{Madre}(x,y) \rightarrow .$$

Si ya existe en la base de datos Padre(Juan, Pedro), un intento por introducir Madre(Juan, Pedro) a la base de datos llevará a una violación de integridad. Esto no rige para otras clases de restricciones de integridad.

Tipo 4: $k \geq 1$, $q = 1$. Las cláusulas tienen la forma

$$P_1 \wedge P_2 \wedge \ldots \wedge P_k \rightarrow R_1.$$

La cláusula puede ser considerada tanto como una restricción de integridad como la definición del predicado R_1 en términos de los predicados $P_1, \ldots, P_k$ (dicha definición es una ley deductiva).

Tipo 5: $k = 0$, $q > 1$. Las cláusulas tienen la forma

$$\rightarrow R_1 \vee R_2 \vee \ldots \vee R_q$$

Si las x_i, donde $i = 1, \ldots, n$ son constantes, entonces tenemos una aseveración indefinida. Esto es, cualquier combinación de una o más R_i es verdadera, pero no sabemos específicamente cuales son verdaderas.

Tipo 6: $k \geq 1$, $q > 1$. Las cláusulas tienen la forma

$$P_1 \wedge P_2 \wedge \ldots \wedge P_k \rightarrow R_1 \vee R_2 \vee \ldots \vee R_q.$$

La cláusula puede ser interpretada tanto como una restricción de integridad o como la definición de datos indefinidos. Una restricción de integridad que establece que cada individuo puede tener a lo más 2 padres puede ser escrita como:

$$P(x_1, y_1) \wedge P(x_1, y_2) \wedge P(x_1, y_3) \rightarrow (y_1 = y_2) \vee (y_1 = y_3) \vee (y_2 = y_3).$$

Como regla general de deducción tendríamos

$$\text{Progenitor}(x, y) \rightarrow \text{Madre}(x, y) \vee \text{Padre}(x, y).$$

La regla general puede también ser interpretada como restricción de integridad.

Finalmente, una cláusula donde $k = 0$, $q = 0$ (la cláusula vacía) denota falsedad y no debe ser parte de una base de datos. Además, una cláusula será definida si su lado derecho consiste exactamente de un átomo (i.e., Tipo 1b o Tipo 4).

Todos los tipos de cláusulas antes definidos, excepto hechos concretos (Tipo 1a), son tratados como restricciones de integridad en bases de datos convencionales. En una base de datos deductiva algunos de ellos pueden ser tratados como leyes deductivas. Podemos distinguir dos clases de bases de datos: bases de datos definidas en las cuales no existen cláusulas de los Tipos 5 y 6 y bases de datos indefinidas en donde si aparecen estos tipos de cláusulas.

En la siguiente tabla aparece un resúmen de la clasificación de cláusulas hecha por de Minsky.

Tabla 2.2 Clasificación por tipos de cláusulas.

$$\sim P_1 \vee \ldots \vee \sim P_k \vee R_1 \vee \ldots \vee R_q.$$

k	q	Tipo	Forma
0	1	1a	$\rightarrow P(ci1, \ldots, cim)$ donde $ci1, \ldots, cim$ son constantes.
		1b	$\rightarrow P(t1, \ldots, tm)$ cuando algunas o todas las las t_i son variables.
1	0	2a	$P(ci1, \ldots, cim) \rightarrow$ donde todas las $ci1, \ldots, cim$ son constantes.
		2b	$P(t1, \ldots, tm) \rightarrow$ cuando algunas o todas las t_i son variables.
>1	0	3	$P_1 \wedge \ldots \wedge P_k \rightarrow$
≥ 1	1	4	$P_1 \wedge P_2 \wedge \ldots \wedge P_k \rightarrow R_1$
0	>1	5	$\rightarrow R_1 \vee R_2 \vee \ldots \vee R_q$
≥ 1	>1	6	$P_1 \wedge P_2 \wedge \ldots \wedge P_k \rightarrow R_1 \vee R_2 \vee \ldots \vee R_q.$

Al conocer el modelo relacional de bases de datos es admirable la similitud que existe entre los lenguajes relacionales (álgebra relacional, cálculo de tuplas y cálculo de dominios) y las lógicas proposicional y de primer orden, este parecido justifica la elección de la lógica como la estructura de trabajo para el desarrollo de sistemas manejadores de bases de datos. Podemos destacar dos razones, primero, cuando el lenguaje en que se programa un sistema es semejante al lenguaje con que se diseñan sus componentes se obtiene una reducción significativa en el tiempo ocupado para la implantación del sistema. Segundo, podemos aprovechar la capacidad de la lógica para realizar inferencias e integrarla en módulos con características deductivas en el DBMS, logrando productos cada vez más versátiles de acuerdo a las necesidades del entorno.

Capítulo 3

Programación Lógica y Base de Datos Relacional

Para iniciar este capítulo es conveniente destacar algunas de las bondades de la programación lógica que son aprovechadas en la implantación de las bases de datos.

- El no determinismo, cuando una consulta nos puede devolver más de un resultado.
- No se realiza distinción a priori entre las operaciones de entrada y las de salida; el papel que desempeñan los argumentos de entrada y salida de un procedimiento pueden cambiar de una consulta a otra.
- No se hace distinción entre el programa y los datos. La definición de un predicado normalmente consiste tanto de hechos explícitos como de reglas generales (procedimientos) para el proceso posterior de datos. Esta particularidad se comentará en la sección 3.1.

Estas características tienen implicaciones importantes para nuestro tema: una base de datos puede ser vista como un programa lógico, en el cual la obtención de información es hecha automáticamente a través del proceso de resolución. Las operaciones para obtener datos comunes en el modelo de base de datos relacional no se necesita programarlas dentro de Prolog. Ilustrémoslo con un ejemplo.

En Prolog una relación se especifica como un procedimiento hecho de cláusulas que corresponden a cada una de las tuples, sean

```
'EMP'(201,'José Esparza',1000000,3,3).
'EMP'(317,'Antonio Bustamante',1100000,7,2).
```

(se usan apóstrofes para prevenir que nombres con mayúsculas sean tratados como variables) dos tuplas de una base de datos.

Operaciones primitivas en relaciones pueden ser expresadas en términos de llamados a procedimientos. Por ejemplo, el procedimiento

```
s(Numemp,Nombre,Sueldo,Puesto,1):-
     'EMP'(Numemp,Nombre,Sueldo,Puesto,1),Sueldo>1000000.
```

puede ser usado para generar todas las tuplas de empleados que pertenezcan al área 1 y que ganan más de un millon; ésta es la operación de selección. El llamado a este procedimiento podría ser el siguiente:

```
?-s(E,N,S,P,A),write((E,N,S,P,A)),nl,fail.
```

El procedimiento p puede ser usado para implementar proyección:

```
p(Nombre,Numemp,Sueldo):-
     'EMP'(Numemp,Nombre,Salario,_,_).
```

La composición de la selección y de la proyección se vería del siguiente modo:

```
s_luego_p(Nombre,Numemp,Sueldo):-
     'EMP'(Numemp,Nombre,Sueldo,_,1),Sueldo>1000000.
```

O puede ser indicado directamente con la siguiente consulta:

```
?-'EMP'(E,N,S,_,1),S>1000000,write((N,E,S)),nl,fail.
```

Como último ejemplo, esta es la unión de las relaciones EMP y AREA sobre los números de áreas que coinciden:

```
j(Numemp,NombreE,Sueldo,Puesto,Area,Jefe):-
     'EMP'(Numemp,NombreE,Sueldo,Puesto,Area),'AREA'(Area,Jefe).
```

3.1 Base de Datos explícita en Prolog

Durante la evolución de la tecnología de bases de datos ha sido relevante el esfuerzo hecho para que el usuario no se preocupe de los detalles concernientes a la representación de los datos, y dejando que centre sus bríos en la información contenida en sus datos. Este concepto es generalmente conocido como independencia de los datos, y requiere de modelos basados en formalismos matemáticos que permitan al usuario ver el contenido de una base de datos sin importarle como estén almacenados físicamente los datos. En particular, el concepto de independencia de los datos ha sido una de las principales motivaciones para el desarrollo de bases de datos relacionales.

Desde el punto de vista declarativo, un programa lógico define que hechos pueden ser inferidos de las cláusulas de que está compuesto, y desde el punto de vista operacional, éstas dictaminan como el programa será ejecutado. En otras palabras, la descripción de un problema en términos de cláusulas de lógica de primer orden y el programa que resuelve dicho problema pueden ser realizadas al mismo tiempo. De hecho, un programa lógico eficiente también incluye características no lógicas para entrada, salida y control, pero pueden ser ignoradas desde el punto de vista de la función declarativa mientras no alteren los hechos que puedan ser inferidos del programa, tan solo se dedican a indicarle al interprete como ejecutar de una manera eficiente. En particular para nuestro problema, el programa lógico sirve para definir los datos y sus atributos como al mismo tiempo el procesarlos. En sistemas convencionales de base de datos estas dos funciones (de descripción y de consulta de una relación) son realizadas por componentes separadas, generalmente basadas en diferentes formalismos. Además, el hecho de que los aspectos operacionales tomados en cuenta automáticamente por el intérprete elimina la necesidad de un lenguaje de programación propio que resulta en un alto nivel de independencia de los datos, el usuario sólo necesita trabajar sobre la semántica declarativa de su base de datos.

Si intentamos modelar relaciones de bases de datos mediante predicados, sus argumentos deben pertenecer a dominios específicos, no sólo para respetar el concepto matemático de una relación, sino también debido a varias razones prácticas.

- Los predicados incluidos en el programa mejòran la eficiencia, pues son responsables de una reducción importante del espacio de búsqueda.
- Permiten la detección de ciertas anomalías semánticas en una consulta. Por ejemplo, actividades como hablar son asociadas al dominio de los humanos y así detectar errores como atribuir esta capacidad a algún animal. En casos en que la excepción confirme la regla es necesario aumentar el o los predicados que incluyan dicho caso en el universo en que se trabaje.
- Los predicados proveen un medio eficaz de desecho de consultas que sean sintácticamente aceptables pero semánticamente erróneas, pueden decidir en la situación en que exista incertidumbre en determinar quién sea el núcleo del sujeto y quién el modificador directo o indirecto del mismo. Ambigüedades que aparezcan con diferentes significados de palabras homónimas pueden en ocasiones ser resueltos mediante verificación del contexto en que se encuentren.

3.2 Negación como no probabilidad

Se debe de tener cuidado cuando se representa información negativa, pues bajo ciertas circunstancias, sus interpretaciones operativas y declarativas pueden no coincidir.Esencialmente, dos enfoques pueden ser analizados:

- Explícita, por ejemplo, con afirmaciones del tipo "¬ p".
- Implícita, estableciendo la convención de que todos aquellos hechos cuya veracidad no pueda ser comprobada se asuma de que sean falsos.

El segundo enfoque presupone un completo conocimiento del dominio que esta siendo representado. Esta suposición -conocida como la suposición cercana al mundo real- es suficientemente natural para muchos dominios de aplicación. En éstos la representación implícita es preferible puesto que la información negativa generalmente es más que la información positiva, y representarla explícitamente puede resultar redundante en lugar de simplemente establecerla por omisión.

En investigaciones hechas por Reiter [REIT78] se ha demostrado que la suposición cercana al mundo real puede llevar a inconsistencias para bases de datos que no sean de cláusulas de Horn, pero no sucede para aquellas que si sean hechas con cláusulas de Horn. Clark [CLAR78] ha investigado la relación que existe entre la implantación de la negación por omisión y su semánticas funcionales verdaderas, haciendo explícita la suposición cercana al mundo real en términos de definiciones "sí y sólo si" en vez de los procedimientos tradicionales "si"(if) de los programas lógicos. Dahl [DAHL80] ha encontrado que usando suposiciones cercanas al mundo real no garantizan del todo que la negación por omisión funcione adecuadamente pues las variables de la consulta dan una respuesta equivocada cuando no están instanciadas. Una posible solución a este problema es posponer dinámicamente la evaluación de algunos átomos en una consulta, de acuerdo a un criterio predefinido. Desde este enfoque, la ejecución de la fórmula no(p) puede ser bloqueada hasta que p contenga variables instanciadas.

3.3 Plural distributivo y colectivo

Como sabemos, el adjetivo plural indica la existencia de más de un elemento. Necesitamos dos clases diferentes de relaciones para distinguir la diferencia entre plural distributivo y plural colectivo. Ilustrémoslas con un ejemplo de cada una. La frase "A y B son paralelas" revela una relación colectiva de plural, dado que la propiedad de ser paralela se aplica al conjunto completo {A,B}, y no se considera a A o a B individualmente. En cambio el enunciado "Arturo y José corren" manifiesta una relación distributiva, ya que es verdadera cuando cada uno de ellos corren, es falsa cuando ambos no corren, e indefinida en los casos restantes, es decir, si uno de ellos corre y el otro no.

Este tercer valor lógico (indefinido) no es estrictamente necesario para distinguir estos dos plurales semánticamente diferentes, pero puede hacerse uso de ello para permitir respuestas sutiles a preguntas que conciernan a conjuntos.

3.4 Tratamiento de expresiones de conjuntos

Las relaciones pueden aplicarse con conjuntos si se permiten plurales colectivos. Esto es una consecuencia natural del trato con bases de datos, como se puede apreciar en el cálculo relacional: muchas de las preguntas que involucran a conjuntos cumplen con ciertas características.

3.5 Necesidad de más de dos valores

Tres valores lógicos podrían ser útiles, pues en lenguaje natural hay dos maneras en que un enunciado puede fallar al tomar una decisión: ya sea porque su negación decida, o porque alguna presuposición en el enunciado falle al tratar ser satisfecho. En el último caso, el enunciado puede quedar en un estado incierto en lugar de falso. A un sistema de consulta-respuesta se le puede incluir un tercer valor lógico (indefinido) a un enunciado que contenga presuposiciones falsas, debido a que si el enunciado es declarado como falso, el usuario puede interpretar erróneamente lo que manifieste dicha negación. Es, por ejemplo, el caso del enunciado "El empleado que gana mil millones vive en Guadalajara" con respecto a la base de datos donde no exista ningún empleado que gane mil millones. Al recibir la respuesta falso el usuario puede cometer el error de pensar que sí existe el empleado que gana mil millones pero que no vive en Guadalajara. Otro enfoque a este problema es el pragmático; que sin necesidad de añadir un tercer valor lógico, consiste en hacer notar al usuario que la base de datos no contiene a la información buscada. Ambos puntos de vista son factibles de ser incluidos en un programa lógico.

Tabla 3.1 Programación lógica en aplicaciones de base de datos.

Ventajas	Desventajas
Capacidad deductiva	Los predicados deben ser tecleados
Reglas generales	Es necesario un tratamiento especial a la negación
Recursividad	
Modularidad inherente	Las relaciones colectivas y distributivas deben de ser distinguidas
No determinismo	
Doble rol (I/O) de cualquier argumento	Las relaciones deben aplicarse en conjuntos
No distinción entre programa y datos	
Doble interpretación (operacional/declarativa)	No son suficientes dos valores lógicos

Todas las consideraciones hechas en este capítulo, resumidas en la Tabla 3.1, nos llevan a la conclusión de que la programación lógica tiene cualidades que pueden ser explotadas al ser usada con las bases de datos, siendo vista como un formalismo único que comprenda varios aspectos de un sistema de base de datos, incluyendo interfases de lenguaje natural. Analizando el objetivo planteado en este trabajo lo podemos enunciar en términos de un sistema hecho en Prolog que pueda manejar conjuntos, predicados tecleados, tres valores lógicos, relaciones distributivas y colectivas, y en el cual puedan ser definidas bases de datos cercanas al mundo real.

Capítulo 4

Lenguaje Natural

En los años cincuentas y sesentas muchas personas trabajaron en el problema de la traducción mecánica de lenguajes naturales, como inglés, español, chino, ruso, etc. Antes de que estos lenguajes puedan ser traducidos, la estructura de los enunciados en los lenguajes debe ser comprendida. Entre otros, el lingüista norteamericano Noam Chomsky ha desarrollado varios modelos de estructura lingüística [CHOM69], obteniendo resultados aplicados a la traducción de programas de computadora. En la sección 4.1 veremos la clasificación de los modelos hecha por Chomsky y hablaremos de máquinas abstractas que generan y reconocen frases.

Una de las técnicas que ha gozado de popularidad son las redes de transición aumentada, desarrollada por William Woods en 1970 [WOOD70] donde aprovecha la técnica de autómatas para el reconocimiento de lenguaje natural. Conoceremos este método en la sección 4.2.

En la sección 4.3 se ilustran otras técnicas que hacen énfasis en el análisis semántico (significado de las frases) sobre del sintáctico (reglas de composición de enunciados).

4.1 Gramáticas.

Un lenguaje natural es la combinación de un diccionario que provee los significados de varias cadenas de símbolos (palabras) en el lenguaje y un conjunto de reglas que determinan cuales combinaciones de palabras y signos de puntuación forman enunciados del lenguaje. El conjunto de reglas es conocido como la gramática del lenguaje. La gramática del lenguaje puede ser usada para determinar la estructura del enunciado en el lenguaje. También puede usarse para probar si una secuencia de palabras forma una sentencia del lenguaje.

Es importante remarcar la diferencia que existe entre el significado y la estructura. El significado de las palabras y la interpretación de los significados basada en la estructura es la Semántica del lenguaje; la estructura por si misma es conocida como la Sintaxis del lenguaje.

Antes de definir un lenguaje necesitamos las siguientes definiciones:

- Un *símbolo* es un objeto mínimo distinguible que carece de significado.
- El *alfabeto* Σ es un conjunto de símbolos.
- Una *cadena* (o *sentencia* o *enunciado*) es una secuencia finita y ordenada de símbolos.
- La *longitud de la cadena* es la cantidad de símbolos en la cadena.
- Existe la cadena de longitud cero (*cadena vacía*) y es denotada por ε.

Un *lenguaje* L sobre un alfabeto Σ es un conjunto de cadenas de Σ. Las cadenas son construidas con símbolos mediante la operación de concatenación. La concatenación de dos símbolos x y y, escrita xy, es la secuencia ordenada de los símbolos. La concatenación de dos cadenas α y β, escrita $\alpha\beta$, es la cadena formada por la escritura de los símbolos de α y seguidos por los símbolos de β. Por lo tanto, si $\alpha = x_1x_2...x_n$ y $\beta = y_1y_2...y_m$, entonces $\alpha\beta = x_1x_2...x_ny_1y_2...y_m$.

Si tenemos la cadena $\alpha\beta\gamma$ constituida por la concatenación de las cadenas α, β y γ, α es llamada prefijo y γ sufijo de la cadena $\alpha\beta\gamma$, respectivamente.

Como ejemplos de lenguajes tenemos

$L1 = \{a|a+b\}$

$L2 = \{$Esta es una cadena.$|$PROLOG es un lenguaje de programación.$\}$

$L3 = \{a,b|a,c|a|b\}$.

Estos lenguajes no son similares a los naturales, pues entre otras cosas son muy pequeños, y los segundos tienen una gran cantidad de cadenas (de hecho, en la mayoría de los lenguajes naturales el número de cadenas es infinito); por ello es importante contar con técnicas que resuelvan este problema.

Podemos destacar 3 técnicas. La primera es aquella que exhibe un miembro genérico del conjunto infinito, seguido de restricciones en los valores de cualquier parámetro que pueda aparecer en la forma genérica, por ejemplo $\{a^n | n \geq 1\}$. Una segunda técnica es especificar un conjunto de reglas que definan un procedimiento para probar que una cadena dada pertenezca al lenguaje. La tercera es especificar un procedimiento para generar solamente aquellas cadenas que están en el lenguaje.

Existen dos tipos de símbolos usados en las gramáticas. Los símbolos terminales son los únicos que pueden aparecer en las cadenas del lenguaje. Los símbolos no-terminales son usados para denotar construcciones intermedias en las derivaciones. Por ejemplo, la gramática española usa conceptos como "núcleo del sujeto" y "objeto directo". Estas construcciones intermedias son usadas en la derivación de enunciados del español. Los símbolos terminales del español son las letras del alfabeto, signos de puntuación y el espacio en blanco. El conjunto de símbolos no-terminales es denotado por *N*; el conjunto de símbolos terminales por *T*. Los símbolos no-terminales serán denotados por letras mayúsculas.

Uno de los símbolos no-terminales representa una clase especial de cadenas, comunmente llamados "enunciados" que se les conoce como símbolos iniciales y son denotados por *S*.

El conjunto de reglas de producción es el corazón de la especificación del lenguaje. Cada regla tiene un lado izquierdo y un lado derecho, separados por una flecha (→) . Una regla especifica una opción disponible para la persona que intenta generar una cadena del lenguaje. Esta persona (o máquina) es conocida como el "generador del lenguaje".

¿Cómo construye el generador un enunciado? Primero, empieza con el símbolo inicial, ya que este símbolo denota un enunciado, busca una regla de producción que le diga cómo puede estar formado un enunciado. La nueva cadena producida puede contener símbolos no- terminales que deben ser reemplazados por otras cadenas de símbolos hasta que no aparezca ningún símbolo no-terminal. Durante este proceso el generador construye un número finito de cadenas. La cadena resultante de la aplicación más reciente de una regla es la cadena de trabajo para el generador. La secuencia de cadenas de trabajo usadas por el generador especifica una derivación de la cadena producida. Las reglas de producción especifican como el generador puede proceder de una cadena de trabajo de una derivación a la siguiente cadena de trabajo de la derivación. Siempre que la cadena en el lado izquierdo de una regla esté contenida en la cadena de trabajo, el generador puede reemplazarla con la cadena que aparece en el lado derecho de la misma regla. La substitución inversa no está permitida. Las reglas de producción pueden tener varios símbolos del lado izquierdo. La derivación está completa cuando ninguna regla pueda ser usada para modificar la cadena de trabajo. Si la cadena de trabajo final no contiene ningún símbolo no-terminal, ésta es un enunciado del lenguaje.

Las gramáticas (y los lenguajes) pueden ser clasificadas por las formas de las reglas de producción que son usadas en sus definiciones. Las reglas son clasificadas por las formas de las cadenas en cada lado de cada regla.

La presente clasificación fue introducida por Chomsky [CHOM59]. En este esquema la única diferencia entre el lado izquierdo y el lado derecho puede ser que un

solo símbolo no-terminal es reemplazado por una cadena de símbolos terminales y/o no-terminales. En algunos casos sólo se permite el reemplazo si el símbolo no-terminal aparece en cierto contexto; en estos casos el contexto se especifica en la regla. El contexto debe ser el mismo en ambos lados de la regla. Entonces la forma general de una regla de producción es

$$\varphi A \psi \rightarrow \varphi \propto \psi$$

Esta regla especifica que A puede ser reemplazada por la cadena $\propto$ siempre que la A se encuentra en el contexto $\varphi A \psi$. Ocasionalmente es útil nombrar a los componentes de la regla. Llamamos a φ el contexto izquierdo, a ψ el contexto derecho, y a $\propto$ la cadena de reemplazo. Todos estos componentes pueden ser cadenas arbitrarias de símbolos terminales y/o no-terminales. Ellos pueden ser cadenas vacías (ε).

Tipo 0. Las reglas de forma irrestricta son reglas del tipo 0. Si un gramática posee esclusivamente reglas del tipo 0, es una gramática del tipo 0, y el lenguaje que especifica es un lenguaje del tipo 0. Ya que no existen restricciones en los componentes de las reglas de tipo 0, cualquier regla es una regla de tipo 0, y cualquier gramática de Chomsky especifica un lenguaje de tipo 0. Los lenguajes de tipo 0 son conocidos como *lenguajes recursivos.*

Tipo 1. Las reglas de tipo 1 son como las reglas del tipo 0, excepto que no está permitida la eliminación del símbolo no-terminal. La forma general de una regla del tipo 1 es

$$\varphi A \psi \rightarrow \varphi \propto \psi \quad \text{donde} \propto \neq \varepsilon$$

Cualquier gramática que contenga solamente reglas del tipo 1 es una gramática del tipo 1. Una gramática del tipo 1 especifica un lenguaje del tipo 1. Las reglas, gramáticas y lenguajes son denominados de *contexto sensitivo* o *dependiente del contexto.*

Tipo 2. Las reglas del tipo 2 son como las reglas del tipo 1, excepto que no puede existir ningún contexto. La forma general es

$$A \rightarrow \propto \quad \text{donde} \propto \neq \varepsilon$$

Las reglas del tipo 2 aparecen en las gramáticas de los lenguajes del tipo 2. Ya que no existe dependencia del contexto, los lenguajes del tipo 2 son conocidos como *lenguajes de contexto libre.*

Tipo 3. Las reglas del tipo 3 son como las reglas del tipo 2, pero la cadena de reemplazo está restringida a ser o un solo símbolo terminal o un solo símbolo terminal seguido de un solo símbolo no-terminal. La forma general de las reglas de este tipo es

$$A \rightarrow \propto \quad \text{donde} \propto = a \text{ o} \propto = aB$$

donde a es un símbolo terminal y B es un símbolo no-terminal. Las reglas del tipo 3 son usadas en gramáticas del tipo 3, las cuales especifican lenguajes del tipo 3. Los lenguajes del tipo 3 son llamados *lenguajes regulares.*

Un resumen de la clasificación de Chomsky se encuentra en la siguiente tabla.

Tabla 4.1. La jerarquía de Chomsky.

Tipo	Clase de Lenguaje	Restricciones en las reglas
0	Recursivamente ennumerable	Ninguna.
1	De contexto sensitivo	Sin reglas de reduccion de tamaño.
2	De contexto libre	El lado izquierdo debe ser un símbolo no terminal.
3	Regular	El lado izquierdo debe ser un solo símbolo no terminal y el lado derecho debe ser un símbolo terminal o un símbolo terminal seguido de un símbolo no terminal.

La 3a. categoría puede ser reconocida con procedimientos de análisis gramatical bastante eficientes. Los Lenguajes Regulares pueden ser reconocidos usando un Autómata Finito Determinístico (AFD). Formalmente, un AFD es una quíntupla (I, S, S_0, f, F) donde

I = conjunto finito de símbolos de entrada,

S = conjunto finito de estados,

$S_0 \in S$ = el estado inicial,

$f: I \times S \rightarrow S$ = la función de trancisión al siguiente estado, y

$F \subseteq S$ = el conjunto de estados finales.

Si el final de la cadena de caracteres de entrada al autómata es encontrado y la máquina está en un estado final, entonces la cadena de entrada es aceptada como un enunciado gramatical. La representación gráfica del autómata se hace mediante grafos dirigidos, donde los estados de la máquina -representados por nodos- están conectados por arcos, cada uno etiquetado con un símbolo de entrada cuya presencia como el siguiente caracter en la cadena de entrada permite que la transición asociada suceda.

A pesar que los AFD son eficientes, su utilidad en el reconocimiento del lenguaje natural está limitada por las restricciones de las gramáticas regulares. El empotramiento no está permitido en la definición de lenguajes regulares. Para los lenguajes naturales, el empotramiento es necesario para manejar cosas tales como cláusulas relativas. Aun para definir lenguajes de programación las reglas de empotramiento son necesarias para permitir el anidamiento de expresiones con paréntesis. Por eso los procedimientos de reconocimiento que son posibles para lenguajes regulares son usados principalmente en la fase del análisis léxico de un lenguaje, en el cual la cadena de símbolos de entrada es dividida en una cadena de tokens que representan objetos tales

tales como palabras. Para los lenguajes libres de contexto, particularmente los determinísticos, se han desarrollado algoritmos de análisis gramatical eficientes y son usados extensivamente en compiladores de lenguajes de programación. Desafortunadamente, aunque la mayoría de los lenguajes de programación pueden ser descritos por reglas libres de contexto, los lenguajes naturales no. En una gramática libre de contexto, cada regla tiene un elemento no terminal en su lado izquierdo. Esto significa que la expansión de ese símbolo no terminal no puede ser influenciada por ningún símbolo que se encuentre a su alrededor, de ahí su nombre. Para algún lenguaje natural es importante en muchos casos conocer el contexto en que se encuentran las palabras para poder determinar el significado de los enunciados.

4.2 Redes de Transición Aumentadas (ATN)

Para tener la suficiente capacidad formal de reconocer un lenguaje natural, encontramos que es necesario un ingenio con la capacidad de una máquina de Turing. Además, si queremos reconocer enunciados eficientemente, necesitamos la facilidad de incorporar una variedad de clases de conocimientos en el sistema de análisis gramatical. Las Redes de Transición Aumentadas (Augmented transition networks) o ATN [WOOD70] proveen un medio de hacer ésto. Un ATN es similar a un autómata finito determinístico en las clases de etiquetas de los arcos que definen transiciones entre estados, aunque en el primero éstas han sido aumentadas. Los arcos pueden ser etiquetados con una combinación arbitraria de lo siguiente:

- Palabras específicas.
- Llamadas a otras redes que reconozcan componentes significantes de un enunciado. Estas llamadas pueden ser recursivas respecto a la red en donde se encuentran.
- Procedimientos que realicen pruebas arbitrarias (posiblemente semánticas) tanto en la cadena de entrada como en los componentes del enunciado que ya han sido identificados.
- Procedimientos que construyan estructuras que formen parte del análisis gramatical final.

Debido a esta variedad de condiciones que pueden ser asociadas a los arcos, un ATN tiene la capacidad formal de una Máquina de Turing. Un ejemplo de un ATN se muestra en la Fig. 4.1.

En este ATN encontramos que en cada subgráfica el estado inicial es una categoría gramatical: **S** de enunciado, **NP** de frase nominal o sujeto, y **PP** de frase preposicional. Los estados finales o aceptantes son: en la primera subgráfica **e4** y **e5**, en la segunda **e7** y **e8**, y en la tercera es **e10**. Los símbolos del ATN son: **S** para sentencia ó enunciado, **NP** para frase nominal, **V** para verbo, **aux** para auxiliar como por ejemplo

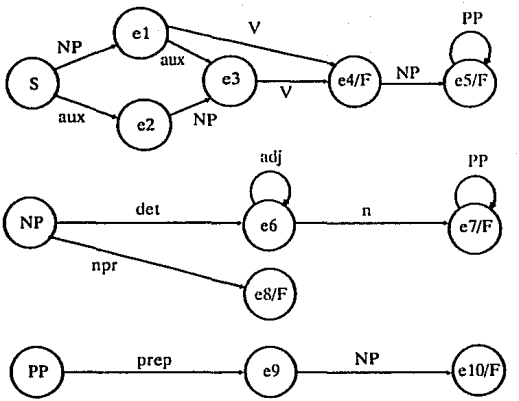


Figura 4.1 Una Red de Transición Aumentada para un subconjunto del idioma inglés.

"do", **det** para determinador (artículo), **PP** para frase preposicional, **npr** para frase pronominal, **prep** para preposición y **n** para nombre o sustantivo.

Es importante tener en mente que el formalismo de un ATN no contiene en si mismo la gramática de ningún lenguaje. Sólo es un mecanismo por el cual gramáticas pueden ser definidas y usadas. Existen principalmente tres modos de manejar las situaciones no determinísticas de un ATN:

- *Regreso hacia atrás* (Backtracking), implantado para almacenar, en cada punto de decisión, el contenido de todos los registros del sistema así como el apuntador del caracter actual de la cadena de entrada. Si sucede la vuelta

- Los símbolos fueron traducidos literalmente para conservar el significado que éstos poseen en el idioma inglés.

hacia atrás en ese punto, la cadena de entrada y los registros son restaurados en sus valores apropiados, para seguir por otro camino.

- *Componer* (Patch up), implantado para permitir a segmentos de código asociados a los arcos mover el contenido de un registro del sistema a otro, y así obtener una nueva interpretación de la cadena de entrada.
- *Esperar y ver*, implantado para permitir al registro de un sistema servir como buffer, de tal modo que se vaya almacenando en ese lugar lo ya analizado y esperar a tener suficiente información para elaborar una interpretación.

Aunque los ATN han probado ser útiles en sistemas de comprensión de lenguaje natural, tienen las siguientes desventajas:

- Pueden ser muy lentos al ejecutarse si es necesario usar muchas veces el backtracking. Ya que, dada una gramática muy grande, muchas sentencias son bastante ambiguas, en cuyos casos el backtracking no puede ser eludido.
- Aunque el conocimiento semántico puede ser usado para rechazar posibles rutas incorporándolo con pruebas en los arcos, no es fácil usar dicho conocimiento a ayudar a elegir la mejor de varias rutas posibles para examinarla primero.
- A menos que todas las palabras en el enunciado sean conocidas por el sistema y que toda la estructura concuerde con una ruta de la red, el proceso de análisis gramatical puede fallar. No existe habilidad para realizar comparaciones parciales.

4.3 Análisis Gramatical

En las dos secciones anteriores hemos hablado de las dos técnicas más populares en el área de Lenguaje Natural; pero por simplicidad las hemos evaluado únicamente desde el punto de vista del análisis sintáctico. El producir un análisis gramatical sintáctico de un enunciado es sólo el primer paso para entenderlo, requiriendo en algún momento, de una interpretación semántica del enunciado. Una manera de hacer ésto es generando una interpretación sintáctica completa y después con la estructura resultante a la mano usar un intérprete semántico. El problema principal de este enfoque es que comúnmente no es posible encontrar la interpretación sintáctica correcta sin considerar alguna información semántica, por ello es importante mezclar ambas en el proceso. La variedad de enfoques que han sido desarrollados para este problema pueden ser divididos en cuatro clases:

- Gramáticas semánticas. Combinan tanto conocimiento sintáctico como semántico en un solo conjunto de reglas.
- Gramáticas de casos. En ellas la estructura construida por el analizador gramatical contiene cierta información semántica.
- Filtrado semántico de análisis sintáctico. Existen dos técnicas:

- Filtrar análisis gramaticales parcialmente cuando se va encontrando. Esto se hace, por ejemplo, con pruebas semánticas en los arcos de un ATN.
- Esperar hasta completar el análisis gramatical y evaluarlo semánticamente.

- Restar el énfasis al análisis gramatical y manejar el proceso de comprensión con conocimiento semántico en vez de sintáctico.

4.3.1 Gramáticas Semánticas

Una gramática semántica es una gramática libre de contexto en la cual la elección de símbolos no terminales y de reglas de producción son regidos tanto por funciones sintácticas como semánticas. Muchos símbolos no terminales representan elementos semánticos, esto es, que su posición depende también del tema que se este tratando, como lugares, cosas, etc. Una gramática semántica puede ser usada por un sistema de análisis gramatical de la misma forma en que pudiera ser usada una gramática estrictamente sintáctica. Las principales ventajas son las siguientes:

- Cuando el análisis gramatical esta completo, el resultado puede ser usado inmediatamente sin otra etapa de proceso que pudiera ser requerida si una interpretación semántica no hubiese sido realizada durante el análisis.
- Muchas ambigüedades que pueden surgir durante un análisis gramatical estrictamente sintáctico son evitadas al tomar en cuenta criterios semánticos durante el análisis.

Consecuencias sintácticas que no afecten al análisis semántico pueden ser ignoradas. Sin embargo, existen también algunas desventajas en el uso de gramáticas semánticas

- El número de reglas requeridas puede llegar a ser demasiado grande ya que faltarían muchas generalizaciones sintácticas.
- Debido a lo anterior, el análisis gramatical puede ser muy lento y ocupar mucho espacio de memoria.

Se puede observar que las gramáticas semánticas pueden ser útiles para producir interfases de subconjuntos de lenguaje natural.

4.3.2 Gramáticas de Casos

Las gramáticas de casos ofrecen un diferente enfoque al problema de cómo pueden ser combinadas las interpretaciones semánticas y sintácticas. Las reglas de producción son escritas para describir regularidades sintácticas más que semánticas. Pero las estructuras que producen las reglas (árboles de análisis gramatical) corresponden a relaciones semánticas en lugar de ser estrictamente sintácticas. En estas representaciones, los roles semánticos se vuelven explícitos. Los casos usados por una

gramática de casos describen relaciones entre verbos y sus argumentos. Esto contrasta con la noción gramatical de casos superficiales que se ve, por ejemplo, en español, por la diferencia que existe entre el YO (caso nominal) y el MI o MIO (caso posesivo). Un caso gramatical, o superficial, dado puede indicar una variedad de casos semánticos, o de fondo.

No existe una convención clara en exactamente cuál es el conjunto correcto de casos de fondo que pudiera ser usado, pero algunos, los más obvios, aparecen en la siguiente tabla:

Tabla 4.2. Casos semánticos más comunes.

Caso	Descripción
Agente	Instigador de la acción (típicamente es animado).
Instrumento	Causa del evento o el objeto usado en causar el evento (típicamente inanimado).
Dativo	Entidad afectada por la acción (típicamente animado).
De lo hecho	Objeto o situación resultante del evento.
De lugar	Ubicación del evento.
Origen	Lugar de donde algo se mueve.
Destino	Lugar a donde se movió algo.
Beneficiario	Entidad por la cual el evento ocurrió (típicamente animado).
Tiempo	Momento en que el evento ocurrió.
Objeto	Entidad que es actuada o que tiene cambios, el caso más general.

El proceso del análisis gramatical dentro de una representación de casos depende mucho de la clasificación léxica asociada con cada verbo. Los lenguajes tienen reglas de mapeo desde las estructuras de casos internos hacia las formas externas sintácticas. Estas reglas pueden ser aplicadas en sentido inverso por un analizador gramatical para determinar la estructura de caso interno a partir de sintaxis superficial. El análisis gramatical usando gramática de casos es manejada por expectativa. Una vez que el verbo del enunciado ha sido identificado, puede ser usado para predecir cuál es la frase del sujeto del enunciado que aparecerá y para determinar la relación del sujeto con el resto del enunciado.

4.3.3 Dependencia Conceptual

Analizar gramaticalmente un enunciado mediante la representación de dependencia conceptual es similar al análisis con gramática de casos. En ambos sistemas, el proceso de análisis gramatical es fuertemente manejado por un conjunto de expectativas que están basadas en el verbo principal del enunciado. Pero debido a que la

representación del verbo en dependencia conceptual está en un nivel menor que en la gramática de casos, la dependencia conceptual ofrece un mayor grado de capacidad predictiva. El primer paso para mapear un enunciado en su representación de dependencia conceptual involucra un procesador sintáctico que extrae el núcleo del sujeto y el verbo. También determina la categoría sintáctica del verbo, esto es, transitivo, imperativo, etc. Entonces entra el procesador conceptual. Hace uso de un diccionario de verbos, el cual contiene una entrada por cada situación en la que el verbo pudiera aparecer. El siguiente ejemplo, tomado de [SCHA73] muestra las entradas del diccionario asociadas con el verbo querer. Estas tres entradas están asociadas con tres tipos de querer:

- Querer que suceda algo.
- Querer un objeto.
- Querer a una persona.

Una vez que la entrada correcta ha sido elegida, el procesador conceptual analiza el resto del enunciado buscando componentes que encajen en los espacios vacíos de la estructura del verbo. El procesador conceptual examina las interpretaciones posibles en el siguiente orden bien definido.

1º Objeto del caso instrumental.

2º Actor adicional del verbo principal.

3º Atributo del actor de la conceptualización.

Debido al manejo de gran cantidad de información semántica en el proceso de comprensión, los enunciados que pudiesen ser ambiguos para un analizador gramatical puramente sintáctico pueden ser asignados a una sola interpretación. Desafortunadamente, la cantidad de información semántica que se requiere para hacer este trabajo perfectamente es inmensa, pues todas las reglas sencillas tienen excepciones. Por ejemplo, tan solo el checar que un objeto sea o no animado no es suficiente para aceptarlo como un actor adicional del verbo transitivo. Es necesario conocimiento adicional. Más información puede ser insertada en la estructura de un procesador conceptual. Pero para producir interpretaciones semánticas correctas de enunciados se necesita información de todo el contexto en que aparece el enunciado.

En la Tabla 4.3 se muestra un resumen de las técnicas comentadas en este capítulo.

El reconocimiento del lenguaje natural es un área de investigación dentro de la inteligencia artificial. Tanto profesionales de cómputo como lingüistas se dedican a estudiar nuevos métodos para atacar este problema. Los éxitos alcanzados hasta hoy han sido implantaciones de subconjuntos de lenguajes naturales, integrados como interfases entre programas y los usuarios. La mejor actitud que podemos tomar al respecto es elegir las bondades de cada una de estas técnicas y mezclarlas de la manera

Tabla 4.3 Técnicas para el reconocimiento de lenguaje natural.

TECNICA	CARACTERISTICAS
ATN	Similar a un autómata, pero incorpora procedimientos (evaluaciones) como peso en sus arcos.
GRAMATICA SEMANTICA	Es una gramática libre de contexto, añade categorías semánticas específicas a las reglas sintácticas.
GRAMATICA DE CASOS	También es una gramática libre de contexto, pero incluye casos semánticos generales en las reglas.
DEPENDENCIA CONCEPTUAL	Independiente del análisis sintáctico, existe un procesador conceptual que analiza espectativas de un evento dado.

más conveniente para conseguir un programa funcional que se adapte al problema que se pretende resolver.

Capítulo 5

Sistema de Base de Datos basado en Lógica FGH

El objetivo de incluir en esta tesis un pequeño sistema de consulta de base de datos basado en lógica es de ilustrar las ideas expuestas a lo largo de los capítulos anteriores, mostrando en la práctica que la lógica es una buena alternativa para el desarrollo, tanto de sistemas manejadores de base de datos, como de las bases de datos mismas. El porqué el sistema fue bautizado como FGH es expuesto en la introducción de esta tesis.

Muchos de los intérpretes y compiladores del lenguaje de programación PROLOG incluyen una extensión notacional llamada DCG (gramática de definición de cláusulas). Esta facilita la programación de gramáticas libres de contexto. Una gramática transcrita del papel en notación DCG es directamente ejecutada por PROLOG como analizador sintáctico. Además la notación DCG permite incorporar la semántica del lenguaje en las reglas de producción de tal modo que el significado de un enunciado pueda ser manejado al mismo tiempo que la sintaxis. En la sección 5.1 conoceremos la notación DCG y la manera en que es traducida por PROLOG a cláusulas de Horn.

Dentro de una frase que pertenezca a un lenguaje como el español son importantes las dependencias que existen entre cada una de las palabras que la componen. Estas dependencias, llamadas contextuales, son reglas que determinan la formación correcta de expresiones de un lenguaje. Las dependencias contextuales que manejará el sistema son descritas en la sección 5.2.

Además de checar que una frase cumpla con las reglas que dictaminan su construcción, es importante conocer el significado que contenga dicha frase. Los

métodos utilizados para extraer la semántica de las consultas del sistema FGH son ilustrados en la sección 5.3.

Finalmente en la sección 5.4 comentaré cómo fueron reunidos los elementos anteriores para formar el sistema de consulta de bases de datos en lenguaje natural basado en lógica FGH.

5.1 Reglas gramaticales en Prolog

Como vimos en el capítulo anterior, una gramática es un sistema formal que define conjuntos de secuencias de símbolos. Cada secuencia de símbolos puede ser abstracta, sin ningún significado, o por el contrario, puede ser un enunciado de un lenguaje de programación, o un programa completo, o un enunciado de un lenguaje natural como el español.

Una notación popular para la descripción de gramáticas es la BNF (Backus-Naur Form), la cual es usada comunmente en la definición de lenguajes de programación. Trabajemos con un ejemplo. Como sabemos, una gramatica contiene reglas de producción. He aquí una gramática BNF muy sencilla con tan solo dos reglas:

$$<s> ::= a\ b$$
$$<s> ::= a\ <s>\ b$$

La primer regla dice: siempre que aparezca el símbolo s en una cadena, puede ser reescrita con la secuencia ab. La segunda regla dice que s puede ser reescrita por la secuencia a, seguida de s, y seguida por b. En esta gramática s se encuentra siempre entre llaves '< >'. Esto indica que s es un símbolo no-terminal de la gramática. Por el otro lado, los símbolos a y b son símbolos terminales. Éstos no pueden ser reescritos, i.e., no pueden aparecer aislados en el lado izquierdo de una regla. En BNF, las dos reglas anteriores pueden ser descritas del siguiente modo:

$$<s> ::= a\ b \mid a\ <s>\ b$$

Solo que para fines de claridad en este capítulo usaremos la primera forma.

Una gramática puede ser usada para generar una cadena de símbolos, llamada un enunciado. El proceso de generación siempre comienza con un símbolo inicial no-terminal, s en nuestro caso. Entonces los símbolos en la secuencia actual son reemplazados por otras cadenas de acuerdo a las reglas de producción. El proceso termina cuando la secuencia actual no contiene ningún símbolo no-terminal. Con nuestra gramática, un ejemplo sería el siguiente. El símbolo inicial es:

$$s$$

Ahora con la segunda regla, s es reemplazado por:

$$a\ s\ b$$

Volviendo a utilizar la segunda regla obtendríamos:

$$a\,a\,s\,b\,b$$

Aplicando la primer regla la secuencia producida sería:

$$a\,a\,a\,b\,b\,b$$

Obviamente la gramática puede generar otras cadenas como *ab*, *aabb*, etc. En general, esta gramática genera cadenas de la forma $a^n b^n$ para $n = 1, 2, ..., \infty$. El conjunto de enunciados generados por la gramática es llamado el lenguaje definido por la gramática.

Esta gramática de ejemplo es muy simple y carente de significado. Sin embargo, podemos usar gramáticas para definir lenguajes más interesantes. Para un tratamiento serio y divertido a la vez de las gramáticas y de los sistemas formales de producción, vea [HOFS79].

Como mencionamos antes, una gramática genera enunciados. En la dirección opuesta, una gramática puede ser usada para reconocer un enunciado dado. Un reconocedor decide si el enunciado pertenece o no a un lenguaje, esto es, reconoce si el enunciado puede ser generado por la correspondiente gramática. El proceso de reconocimiento es escencialmente el inverso al de generación. En el reconocimiento, el proceso inicia con una cadena de símbolos dada, sobre la cual las reglas gramaticales son aplicadas en el sentido opuesto al de la generación: si la cadena actual contiene una subcadena, igual al lado derecho de alguna regla de la gramática, entonces esta subcadena es reescrita con el lado izquierdo de dicha regla. El proceso de reconocimiento termina exitosamente cuando toda la cadena dada ha sido reducida al símbolo inicial de la gramática. Si no hay manera de reducir la cadena dada al símbolo inicial, entonces el reconocedor rechaza al enunciado.

En el proceso de reconocimiento la cadena es separada en sus constituyentes; por lo tanto este proceso es llamado también análisis gramatical. Implantar una gramática normalmente significa escribir el programa de análisis para la gramática. En Prolog estos programas pueden ser escritos de una manera más sencilla usando la notación DCG. Una gramática escrita en DCG es ya un programa analizador sintáctico para dicha gramática. Para transformar una gramática escrita en BNF a DCG solo tenemos que cambiar algunas convenciones de notación. Por ejemplo la gramática anterior es escrita en DCG como:

```
s-->[a],[b].

s-->[a],s,[b].
```

Observe las diferencias entre la notación BNF y la DCG. "::=" es reemplazado por "-->". Los símbolos no-terminales no se encierran entre llaves, en cambio los

símbolos terminales van entre corchetes, constituyéndose como listas de Prolog. Además, cada símbolo va separado por coma y cada regla es terminada por un punto.

En implantaciones de Prolog que aceptan la notación DCG, nuestra gramática transformada puede ser usada inmediatamente como reconocedor de enunciados. Dicho reconocedor espera enunciados representados mediante diferencia de listas de símbolos terminales. Por lo tanto cada enunciado esta representado por dos listas: el enunciado es la diferencia entre las dos listas. Las dos listas no son únicas, por ejemplo la cadena *aabb* puede ser representada por las listas

```
[a,a,b,b] y [],
```

ó

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

ó

```
[a,a,b,b,0,1] y [0,1],
```

etc.

Tomando en cuenta esta representación de enunciados, nuestra gramática de ejemplo puede ser consultada para reconocer enunciados mediante preguntas como:

```
?-s([a,a,b,b],[]).                    %Reconoce la cadena aabb

yes

?-s([a,a,b],[]).

no
```

Hablemos ahora de cómo Prolog usa la notación DCG dada para contestar preguntas. Cuando Prolog consulta (incorpora a su ambiente) las reglas gramaticales las convierte automáticamente en cláusulas propias del Prolog. De esta manera, Prolog traduce las reglas gramaticales en un programa que reconoce enunciados generados por la gramática. Usando nuestro ejemplo, Prolog lo convierte en las siguientes cláusulas:

```
s([a,b|Resto],Resto).
s([a|Resto],Resto1):-
    s(Resto,[b|Resto1]).
```

Pero, ¿qué se consigue con esta conversión? Analicemos el procedimiento s. La relación s tiene dos argumentos - dos listas:

```
s(Diferencia,Resto)
```

que es verdadera si la diferencia de las listas `Diferencia` y `Resto` es una cadena aceptable. Ejemplos de relaciones tenemos las siguientes:

```
s([a,b],[])
```

ó

```
s([a,a,b,b,x,y,z],[x,y,z])
```

ó

```
s([a,b,a,b],[a,b]).
```

Puesto que una diferencia de listas nos marca donde termina una lista podemos ver que la segunda cláusula analiza cualquier cosa que se encuentre rodeada por una *a* al principio y una *b* al final. Además el proceso inverso de generación de cadenas es fácilmente realizado pues la diferencia de listas permite la concatenación de listas de una manera casi inmediata.

Podemos formular de manera más general la translación entre DCG y cláusulas de Prolog. Cada regla escrita en DCG es traducida a una cláusula de Prolog de acuerdo al siguiente esquema. Sea

```
n-->n1,n2,...,nn
```

una regla en DCG. Si `n1`, `n2`,..., `nn` son símbolos no-terminales entonces la regla es traducida a la siguiente cláusula:

```
n(Lista1,Resto):-
      n1(Lista1,Lista2),
      n2(Lista2,Lista3),
      ...
      nn(Listan,Resto).
```

Si algunos de los símbolos `n1`, `n2`,..., `nn` son terminales, (en las reglas de DCG aparecen entre corchetes) entonces se maneja de diferente modo. No aparece como objetivo en la cláusula, sino que es directamente insertado en la lista correspondiente. Por ejemplo consideremos

```
n-->n1,[t2],n3,[t4].
```

donde `n1` y `n3` son no-terminales y `t2` y `t4` son terminales. Esto es traducido a la siguiente cláusula:

```
n(Lista1,Resto):-
      n1(Lista1,[t2|Lista3]),
      n3(Lista3,[t4|Resto]).
```

Como vemos, la DCG es una notación muy útil para ahorrar tiempo en el desarrollo de sistemas de análisis gramatical. En la siguiente sección conoceremos otras facilidades que provee esta sintaxis pero al mismo tiempo trataremos ya el problema que nos atañe.

5.2 Dependencia Contextual

Como hemos venido exponiendo, la notación DCG nos permite definir de una manera cómoda gramáticas de lenguajes interesantes como lenguajes de programación ó subconjuntos de lenguaje natural. Pongamos el siguiente ejemplo:

```
[¿,Qué,alumnos,cursan,física,?]
```

En este enunciado podemos destacar las siguientes partes, el signo de interrogación que abre, el pronombre interrogativo Qué, una oración con sujeto, verbo y objeto directo, y el signo de interrogación que cierra la pregunta. Esto puede ser enunciado con la siguiente regla gramatical:

```
pregunta-->
     signo_interrogacion_abierto,
     pronombre_interrogativo,
     enunciado,
     signo_interrogación_cerrado.
```

y sus partes constituyentes serían las siguientes:

```
signo_interrogación_abierto-->[¿].

signo_interrogacion_cerrado-->[?].

pronombre_interrogativo-->[Qué].

enunciado-->
     sujeto,
     predicado.

sujeto-->
     sustantivo.

predicado-->
     verbo,
     objeto_directo.
```

```
sustantivo-->[alumnos].

verbo-->[cursan].

objeto_directo-->[física].
```

Salta a la vista el hecho de que se deben de añadir más símbolos terminales para que la gramática sea más útil. Entonces tendrían cabida reglas como

```
sustantivo-->[maestros].
sustantivo-->[maestro].
sustantivo-->[grupos].
sustantivo-->[grupo].
sustantivo-->[alumno].
```

Aparece ahora un grave problema, al incorporar estas reglas la gramática pondrá cualquier sujeto con cualquier predicado, como por ejemplo

```
[¿,Qué,grupo,cursan,física,?]
```

En muchos idiomas el sujeto y el predicado de una oración no son independientes, tiene que concordar su número. Ambos deben tener número singular o plural. A este fenómeno se le conoce como *dependencia contextual*. Una frase depende del contexto en que se ubique. Estas dependencias contextuales no pueden ser manejadas directamente por gramáticas BNF, pero sí es posible con gramáticas DCG, usando una extensión a las BNF: argumentos que se añaden a los símbolos no-terminales de la gramática. Entonces añadiremos el parámetro Número como un argumento al sujeto y predicado del siguiente modo:

```
sujeto(Numero)
predicado(Numero)
```

Entonces con esta adición es sencillo modificar nuestra gramática para que sea capaz de forzar a que el número sea el mismo entre el sujeto y el predicado

```
pregunta(Numero) -->
      signo_interrogacion_abierto,
      pronombre_interrogativo,
      enunciado(Numero),
      signo_interrogación_cerrado.

enunciado(Numero) -->
      sujeto(Numero),
      predicado(Numero).
```

```
sujeto(Numero)-->
     sustantivo(Numero).

predicado(Numero)-->
     verbo(Numero),
     objeto_directo(Numero1).

sustantivo(plural)-->[alumnos].
sustantivo(singular)-->[alumno].

verbo(plural)-->[cursan].
verbo(singular)-->[cursa].

objeto_directo(singular)-->[física].
...
```

Cuando estas reglas son leídas por Prolog y convertidas en cláusulas, los argumentos de los símbolos no-terminales son simplemente añadidos a las dos listas que teníamos como argumentos, con la convención de que las dos listas van al último (de izquierda a derecha). Entonces por ejemplo

```
enunciado(Numero)-->
     sujeto(Numero),
     predicado(Numero).
```

quedaría traducido de la siguiente forma

```
enunciado(Numero,Lista1,Resto):-
     sujeto(Numero,Lista1,Lista2),
     predicado(Numero,Lista2,Resto).
```

En general las dependencias contextuales que manejaremos son la voz del verbo, el género y el número; la persona no es necesario incluirla pues todos los elementos que pertenecen a este lenguaje están en la tercera persona. Dos son los géneros que pueden presentarse, masculino y femenino. Tomando por convención de que nuevos argumentos son añadidos a la izquierda de los argumentos actuales, nuestra gramática quedaría así

```
pregunta(Genero,Numero)-->
     signo_interrogacion_abierto,
     pronombre_interrogativo,
     enunciado(Genero,Numero),
     signo_interrogación_cerrado.

enunciado(Genero,Numero)-->
```

```
     sujeto(Genero,Numero),
     predicado(Genero,Numero).

sujeto(Genero,Numero)-->
     sustantivo(Genero,Numero).

predicado(Genero,Numero)-->
     verbo(Genero,Numero),
     objeto_directo(Genero1,Numero1).

sustantivo(masculino,plural)-->[alumnos].
sustantivo(femenino,plural)-->[alumnas].
sustantivo(masculino,singular)-->[alumno].
sustantivo(femenino,singular)-->[alumna].
...

verbo(Genero,plural)-->[cursan].
verbo(Genero,singular)-->[cursa].

objeto_directo(femenino,singular)-->[física].
...
```

Respecto a la voz es necesario hacer un análisis previo. El verbo tiene dos voces: activa y pasiva. La voz activa significa que el sujeto (actor) realiza la acción, por ejemplo

```
[¿,Qué,alumno,cursa,física,?]
```

Y la voz pasiva significa que el sujeto (receptor) recibe la acción, por ejemplo

```
[¿,Qué,materia,es,cursada,por,Juan,?]
```

pero aquí aparecen las complicaciones, pues aunque este enunciado no está comprendido en el lenguaje que hemos definido hasta ahora, se puede decir la misma idea (i.e., semánticamente equivalente) con el enunciado

```
[¿,Qué,materia,cursa,Juan,?]
```

Vemos que el mismo símbolo `cursa` desempeña dos funciones distintas dependiendo del contexto. Para denotar cada uno de los casos es necesario crear las categorías gramaticales actor, y receptor. Entonces el sujeto quedaría definido de la siguiente forma

```
sujeto(activa,Genero,Numero)-->
     actor(activa,Genero,Numero).
```

```
sujeto(pasiva,Genero,Numero)-->
     receptor(pasiva,Genero,Numero).
```

Para definir al predicado debemos primero hacer notar que el sustantivo que forma al sujeto es una palabra que define a una relación de la base de datos, alumno, maestro, ó grupo; en cambio el sustantivo que define al objeto directo del predicado es una instancia de alguna relación de la base de datos, como Juan, física, etc. Entonces el objeto directo del predicado estará definido por

```
objeto_directo(activa,Genero,Numero)-->
     instancia_receptor(activa,Genero,Numero).
objeto_directo(pasiva,Genero,Numero)-->
     instancia_actor(pasiva,Genero,Numero).
```

dependiendo de la voz que tome el verbo. O sea, si la voz del verbo es activa, entonces el objeto directo es una instancia del receptor; en cambio si la voz del verbo es pasiva, el objeto directo será una instancia del actor. Entonces las instancias serán definidas por una extracción a la base de datos para conformarlas en reglas DCG como

```
instancia_actor(pasiva,masculino,singular)-->
     [Juan].

instancia_receptor(activa,femenino,singular)-->
     [física].
```

En la práctica el género y el número de las instancias no depende contextualmente del verbo, por lo que no será necesario determinarlas (en nuestro ejemplo, claro). Entonces la gramática actualizada será:

```
pregunta(Voz,Genero,Numero)-->
     signo_interrogacion_abierto,
     pronombre_interrogativo,
     enunciado(Voz,Genero,Numero),
     signo_interrogación_cerrado.

enunciado(Voz,Genero,Numero)-->
     sujeto(Voz,Genero,Numero),
     predicado(Voz,Genero,Numero).

sujeto(activa,Genero,Numero)-->
     actor(activa,Genero,Numero).
sujeto(pasiva,Genero,Numero)-->
     receptor(pasiva,Genero,Numero).

predicado(Voz,Genero,Numero)-->
```

```
     verbo(Voz,Genero,Numero),
     objeto_directo(Voz,Genero1,Numero1).

actor(activa,masculino,plural)-->[alumnos].
actor(activa,femenino,plural)-->[alumnas].
actor(activa,masculino,singular)-->[alumno].
actor(activa,femenino,singular)-->[alumna].
...

receptor(pasiva,femenino,singular)-->[materia].
receptor(pasiva,femenino,plural)-->[materias].

verbo(Voz,Genero,plural)-->[cursan].
verbo(Voz,Genero,singular)-->[cursa].
verbo(activa,Genero,plural)-->[imparten].
verbo(activa,Genero,singular)-->[imparte].
...

objeto_directo(activa,Genero,Numero)-->
     instancia_receptor(activa,Genero,Numero).
objeto_directo(pasiva,Genero,Numero)-->
     instancia_actor(pasiva,Genero,Numero).

instancia_receptor(activa,Genero,Numero)-->
     [fisica].
...

instancia_actor(pasiva,Genero,Numero)-->
     [Juan].
```

De este modo nuestra gramática podrá analizar sintácticamente cadenas como

```
[¿,Qué,materias,cursa,Juan,?].
```

5.3 Semántica

5.3.1 Arboles de Análisis Gramatical (Parse Trees)

Ilustremos primero con un ejemplo el concepto de *árbol de análisis gramatical.* De acuerdo con un subconjunto de nuestra gramática, el enunciado

```
[alumnos,cursan,fisica]
```

es analizado sintácticamente como se muestra en la Figura 5.1. Algunas partes del enunciado son llamadas *frases* - aquellas partes que corresponden a símbolos no-terminales de la gramática. En nuestro ejemplo, `[cursan, fisica]` es una frase que corresponde al símbolo no-terminal `predicado`, y `[alumnos]` es otra frase que corresponde al símbolo no-terminal `sujeto`. Podemos observar en la figura como el árbol de análisis gramatical contiene como subárboles a los árboles de análisis gramatical de las frases.

Definamos que es un árbol de análisis gramatical. Un árbol de análisis gramatical de una frase es un árbol con las siguientes propiedades

- Todas las hojas del árbol están etiquetadas por símbolos terminales de la gramática.
- Todos los nodos internos del árbol son etiquetados por símbolos no-terminales; la raíz del árbol está etiquetada por el símbolo no-terminal que corresponde a la frase.
- La relación padre-hijo en el árbol está especificada por las reglas de la gramática. Por ejemplo, para la regla

```
enunciado-->
      sujeto,
      predicado.
```

`sujeto` y `predicado` son los hijos de `enunciado`.

En ocasiones es de utilidad tener el árbol de análisis gramatical representado de manera explícita en el programa para realizar algún proceso en él- por ejemplo, extraer el significado de la cadena. El árbol puede ser construido de manera sencilla usando

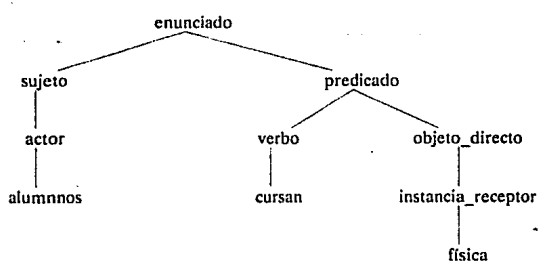


Figura 5.1 Arbol de análisis sintáctico.

argumentos de símbolos no-terminales en la notación DCG. El árbol de análisis gramatical puede ser representado convenientemente como un término de Prolog cuyo functor es la raíz del árbol y cuyos argumentos son los subárboles del árbol. Por ejemplo, el árbol de análisis gramatical del predicado `[cursan, fisica]` puede ser representado por

```
predicado(verbo(cursan),
     objeto_directo(instancia_receptor(fisica)))
```

Para generar el árbol, la gramática DCG puede ser modificada añadiendo a cada símbolo no-terminal su propio árbol de análisis gramatical como un argumento. Por ejemplo, el árbol de análisis gramatical del predicado en nuestro ejemplo sería

```
predicado(Verbo,Objeto_directo)
```

Aquí `Verbo` y `Objeto_directo` son los árboles de análisis gramatical de un verbo y un objeto directo respectivamente. Añadiendo estos árboles como argumentos a la regla gramatical del predicado obtendríamos

```
predicado(predicado(Verbo,Objeto_directo))-->
     verbo(Verbo),
     objeto_directo(Objeto_directo).
```

Esta regla puede ser descrita como un predicado de Prolog cuyo árbol de análisis gramatical es

```
predicado(Verbo,Objeto_directo)
```

que a su vez consiste de:

- un verbo cuyo árbol es `Verbo`, y
- un objeto directo cuyo árbol es `Objeto_directo`.

De igual modo podemos modificar toda nuestra gramática. Para asegurarnos de que los argumentos esten correctos -tanto en número como en concepto- pondremos primero las dependencias contextuales y al final el árbol de análisis gramatical. He aquí parte de la gramática modificada:

```
enunciado(Voz,Genero,Numero,enunciado(Sujeto,Predicado))-->
     sujeto(Voz,Genero,Numero,Sujeto),
     predicado(Voz,Genero,Numero,Predicado).

sujeto(activa,Genero,Numero,sujeto(Actor))-->
     actor(activa,Genero,Numero,Actor).
sujeto(pasiva,Genero,Numero,sujeto(Receptor))-->
     receptor(pasiva,Genero,Numero,Receptor).
```

```
predicado(Voz,Genero,Numero,
     predicado(Verbo,Objeto_directo))-->
     verbo(Voz,Genero,Numero,Verbo),
     objeto_directo(Voz,Genero1,Numero1,Objeto_directo).

actor(activa,masculino,plural,actor(alumnos))-->
     [alumnos].
actor(activa,femenino,plural,actor(alumnas))-->
     [alumnas].
...

receptor(pasiva,femenino,singular,receptor(materia))-->
     [materia].
receptor(pasiva,femenino,plural,receptor(materias))-->
     [materias].

verbo(Voz,Genero,plural,verbo(cursan))-->
     [cursan].
verbo(Voz,Genero,singular,verbo(cursa)) -->
     [cursa].
...

objeto_directo(activa,Genero,Numero,
          objeto_directo(Instancia_receptor))-->
     instancia_receptor(activa,Genero,Numero,Instancia_receptor).
objeto_directo(pasiva,Genero,Numero,
          objeto_directo(Instancia_actor))-->
     instancia_actor(pasiva,Genero,Numero,Instancia_actor).

instancia_receptor(activa,Genero,Numero,
          instancia_receptor(física))-->[física].
...

instancia_actor(pasiva,Genero,Numero,
          instancia_actor(Juan))-->[Juan].
```

Al ser leída por Prolog esta gramática es automáticamente traducida en un programa de Prolog. La primera regla de las anteriores es traducida en la cláusula:

```
enunciado(Voz,Genero,Numero,
          enunciado(Sujeto,Predicado), Lista1,Resto) :-
     sujeto(Voz,Genero,Numero,Sujeto,Lista1,Lista2),
     predicado(Voz,Genero,Numero,Predicado,Lista2,Resto).
```

De acuerdo a esta sintaxis puede preguntársele a Prolog el árbol de análisis gramatical de un enunciado, por ejemplo:

```
?-enunciado(Voz,Genero,Numero,Arbol,
          [alumnos,cursan,fisica],[]).

Voz=pasiva
Genero=masculino
Numero=singular
Arbol=enunciado(sujeto(actor(alumnos)),predicado(verbo(cursan
),objeto_directo(instancia_receptor(fisica))))
```

Las gramáticas DCG son una herramienta adecuada para el tratamiento del significado en un lenguaje, en particular de lenguajes naturales. Los argumentos que son añadidos a los símbolos no-terminales de la gramática pueden ser usados para manejar el significado de los enunciados. Un enfoque muy simple para extraer el significado involucra dos etapas:

- Generar el árbol de análisis gramatical de una frase dada.
- Procesar el árbol para conocer el significado.

Por supuesto esto sólo puede ser práctico si la estructura sintáctica, representada por el árbol, reflejara también la estructura semántica; esto es, si las descomposiciones sintáctica y la semántica tuvieran estructuras similares. En tal caso, el significado de un enunciado estaría compuesto de los significados de las frases sintácticas que conforman el árbol de análisis gramatical del enunciado. Ejemplo de esta situación son los árboles que construye un compilador o intérprete de un lenguaje de programación de cada una de las expresiones aritméticas que pueden aparecer en el código a traducir. Desafortunadamente no es una alternativa aplicable a nuestro caso. Claro, la notación DCG nos permite incorporar el significado directamente en la gramática evitando así la construcción intermedia del árbol de análisis gramatical, lo cual veremos a continuación.

5.3.2 Significado en Lenguaje Natural

Definir el significado del lenguaje natural es un problema tan difícil que actualmente es un área de investigación. Una última solución al problema de formalizar toda la sintaxis y la semántica de un lenguaje como el español se ve muy lejana. Pero subconjuntos relativamente sencillos de los lenguajes naturales han sido formalizados exitosamente y consecuentemente implantados como programas útiles.

Al definir el significado de un lenguaje, la primera pregunta es: ¿cómo puede ser representado el significado? Por supuesto que existen muchas alternativas y una buena elección dependerá de la aplicación en particular. Entonces la pregunta importante es: ¿cuál significado del texto en lenguaje natural será usado? Una de las aplicaciones más

comunes es aquella que nos compete: consultar una base de datos mediante enunciados de lenguaje natural. En este caso la representación a obtener del significado será un lenguaje de consulta de base de datos.

La Lógica ha sido aceptada como un buen candidato para la representación del significado de enunciados de lenguaje natural. En comparación con formalismos de base de datos, la Lógica es más expresiva y, además de subsumar a dichos formalismos (como el caso del modelo relacional), también permite tratar con consecuencias semánticas más sutiles. Veamos como se puede usar la notación DCG para obtener interpretaciones semánticas de enunciados que pertenezcan a un subconjunto de lenguaje natural.

Para empezar, veamos como traduciríamos el significado una oración del lenguaje natural a la Lógica de primer orden. Consideremos la frase **"Juan cursa física"**. La manera más sencilla de representarla en lógica, como un término de Prolog, es

```
cursa(Juan,fisica)
```

Definamos ahora, mediante reglas de DCG, el significado de estas sencillas frases, dentro del contexto de nuestro sistema. Por sencillez comenzaremos con las reglas sin argumentos y paulatinamente iremos incorporando el significado en esas reglas. He aquí la gramática que cubre la sintaxis para nuestro ejemplo

```
pregunta-->
     signo_interrogacion_abierto,
     pronombre_interrogativo,
     enunciado,
     signo_interrogación_cerrado.

signo_interrogación_abierto-->[¿].

signo_interrogacion_cerrado-->[?].

pronombre_interrogativo-->[Qué].

enunciado-->
     sujeto,
     predicado.

sujeto-->
     receptor.

predicado-->
     verbo,
```

```
     objeto_directo.

objeto_directo-->
     instancia_actor.

receptor-->[materia].

verbo-->[cursa].

instancia_actor-->
     [Juan].
```

Incorporemos ahora el significado en estas reglas. Comenzaremos por las categorías más sencillas- `instancia_actor` y `verbo`- y luego continuaremos hacia las más complicadas. Nuestros ejemplos anteriores sugieren las siguientes definiciones

```
instancia_actor('Juan')-->
     [Juan].
```

Esto es, el significado del símbolo terminal **'Juan'** es `Juan` precisamente.

El significado de un verbo como **'cursa'** es un poco más complejo. Puede ser descrito como

```
cursa(X,Y)
```

donde `X` e `Y` son dos variables que sólo serán conocidas por el contexto, ya sea en el objeto directo de la oración o bien en la consulta de la base de datos, según sea el caso. De este modo, la regla DCG correspondiente quedaría así:

```
verbo(cursa(X,Y))-->
     [cursa].
```

Ahora el problema a resolver es ¿cómo podemos instanciar los significados del verbo y el objeto directo? Pues si el significado de la pregunta sería `cursa(Juan, Y)`, entonces debemos forzar a que el argumento `X` sea igual a `Juan`.

En el punto en que nos encontramos nos puede ser útil ver la Figura 5.2. Muestra como el significado de las frases se va acumulando para llegar al significado de todo el enunciado. Para lograr el efecto de propagación de los significados de las frases, podemos primero definir que el símbolo no-terminal `objeto_directo` reciba su significado de `instancia_actor`:

```
objeto_directo(Significado)-->
     instancia_actor(Significado).
```

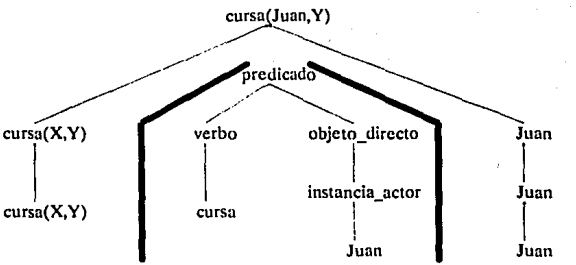


Figura 5.2 Propagación del significado en un árbol.

Del mismo modo continuaríamos con las demás reglas hasta llegar al símbolo inicial. Pero aún no hemos resuelto el problema de forzar una variable a tomar un valor del contexto, y en la regla del símbolo predicado la necesitamos. Para instanciarla explícitamente es necesario hacer al argumento `X` del término `cursa(X,Y)` "visible" desde afuera del término, como un argumento más de los símbolos no-terminales para que ésta sea accesible para ser instanciada. Como en nuestro ejemplo nos encontramos trabajando con la voz pasiva del verbo, sólo extraeremos al actor (`X`) del significado del verbo. Entonces volveremos a definir las reglas anteriores:

```
verbo(Actor,cursa(Actor,Receptor))-->
     [cursa].

objeto_directo(Actor)-->
     instancia_actor(Actor).
```

Ahora sí la regla predicado está lista para unir los significados de sus dos componentes en uno solo:

```
predicado(Significado)-->
     verbo(Actor,Significado)
     objeto_directo(Actor).
```

Esto forza al argumento `Actor` del significado del verbo que sea igual al significado del objeto directo.

Esta técnica de hacer visibles partes de significados es un truco algo común en incorporar significado en reglas DCG. La técnica funciona a grandes rasgos como sigue.

El significado de una frase es definido en un esquema o esqueleto - por ejemplo, `cursa(X,Y)`. Esto define la forma general del significado, pero deja parte del mismo sin instanciar, (aquí las variables `X` y `Y`). Tales variables sin instanciar sirven como ranuras que pueden ser llenadas después dependiendo del significado de otras partes del contexto. Este llenado de ranuras puede ser hecho por el mecanismo de emparejamiento de Prolog. Sin embargo, para facilitar esto las ranuras son visibles añadiéndolas como argumentos extra a los símbolos no-terminales. Adoptaremos la siguiente convención para mantener el orden de los argumentos: primero aparecerán las ranuras visibles del significado de la frase, seguidos por el significado mismo, por ejemplo, `verbo(Actor, Significado)`. A continuación aparece una muestra de la gramática.

```
pregunta(Significado)-->
     signo_interrogacion_abierto,
     pronombre_interrogativo,
     enunciado(Significado),
     signo_interrogación_cerrado.

signo_interrogación_abierto-->[¿].

signo_interrogacion_cerrado-->[?].

pronombre_interrogativo-->[Qué].

enunciado(Significado)-->
     sujeto,
     predicado(Significado).

sujeto-->receptor.

receptor-->[materias].

predicado(Significado)-->
     verbo(Actor,Significado),
     objeto_directo(Actor).

verbo(Actor,cursa(Actor,Receptor))-->[cursa].

objeto_directo(Actor)-->
     instancia_actor(Actor).

instancia_actor(Juan)-->[Juan].
```

Si realizaramos la siguiente consulta:

```
?-pregunta(Significado,[¿,Qué,materias,cursa,Juan,?],[]).
```

obtendríamos como respuesta

```
Significado=cursa(Juan,Receptor)
```

5.3.3 Consulta a Bases de Datos

Hasta este momento hemos obtenido el significado de una consulta en términos de lógica de predicados, lo que nos permitirá incorporarlo al ambiente de Prolog para efectuar la consulta de una manera casi inmediata.

Comunmente una base de datos relacional puede ser representada de una manera natural como un conjunto de hechos. En este caso cada alumno tendrá cinco componentes: número de cuenta, datos personales, clave Facultad o Escuela, clave estudios, y materias. Los datos personales consisten en el nombre o nombres, apellido paterno, apellido materno y fecha de nacimiento. Como la cantidad de materias cursadas y las que se están cursando varía de alumno en alumno, las materías son representadas por una lista que es capaz de contener cualquier cantidad de elementos. A su vez, cada materia es representada por una cláusula con los atributos nombre (que en este ejemplo fungirá como clave) y calificación ó el símbolo `cursando` cuando este inscrito pero no haya acreditado aún la materia. Entonces la representación en predicado de un alumno será:

```
alumno(8656184-6,
personales('Juan','Pérez','López',fecha(25,dic,1965)),
240,21,[materia('física',8),materia('química',cursando)])
```

Nuestra base de datos estará compuesta por una secuencia de hechos como éste que describan a todos los alumnos que sean de interés para nuestra aplicación.

Prolog es, de hecho, un lenguaje muy adecuado para obtener información de una base de datos. Un detalle importante de Prolog es que podemos referirnos a objetos sin necesidad de especificar en el momento todos los componentes de dicho objeto. Podemos meramente indicar la estructura de los objetos en que estamos interesados, y dejar sin especificar o parcialmente especificados a componentes del objeto. Entonces podemos referirnos a todos los alumnos que se llamen "Juan" mediante:

```
alumno(_,personales('Juan',_,_,_),_,_,_)
```

El caracter de subrayado (_) denota diferentes variables anónimas; i.e., no nos importan sus valores. Además, podemos referirnos a todos los alumnos que tienen 4 materias en su curricula como

```
alumno(_,_,_,_,[_,_,_,_])
```

Para encontrar los apellidos de todos los alumnos que tengan por lo menos 4 materias dirigiríamos al Prolog la siguiente consulta:

```
?-alumno(_,personales(_,Paterno,Materno,_),_,_,[_,_,_,_|_]).
```

El objeto de estos ejemplos es mostrar que podemos especificar objetos de interés mediante su estructura, no por su contenido. Sólo indicamos la estructura y dejamos sus argumentos como ranuras sin definir.

Podemos construir un conjunto de procedimientos que puedan ser de utilidad para hacer más cómoda la interacción con la base de datos. Tales procedimientos pueden ser parte de la interfase de usuario. Algunos procedimientos de utilidad para nuestra base de datos son:

```
datos_alumno(X):-                       % X son los datos personales
     alumno(_,X,_,_,_).                 % de un alumno

asignatura(X,Persona):-                 % X es una asignatura
     alumno(_,Persona,_,_,Materias),    % de Persona
     miembro(X,Materias).

miembro(X,[X|L]).                       % X es miembro de
miembro(X,[Y|L]):-                      % la lista L
     miembro(X,L).

fecha_nac(personales(_,_,_,Fecha),Fecha).

calif_materia_alumno(materia(_,Calificacion),Calificacion).
```

Podemos usar estas utilerías, por ejemplo, en las siguientes consultas:

- Busca los nombres y apellidos paternos de todos los alumnos que se encuentran en la base de datos:

```
?-datos_alumno(personales(Nombre,Paterno,_,_)).
```

- Busca a todos los alumnos nacidos en 1965:

```
?-datos_alumno(X),
      fecha_nac(X,fecha(_,_,1965)).
```

- Busca a todos los alumnos que su primer apellido sea 'Pérez' y que hayan nacido después de 1970:

```
?-datos_alumno(personales(Nombre,Paterno,_,fecha(_,_,A))),
      A<1970.
```

- Busca a los alumnos que hayan cursado ó están cursando física:

```
?-alumno(_,Persona,_,_,Materias),
      miembro('fisica',Materias).
```

- Busca la calificación de los alumnos que hayan cursado química:

```
?-datos_alumno(X),
      asignatura(X,materia('quimica',Calif)),
      not(Calif == cursado).
```

Para calcular el promedio de un alumno es necesario definir primero la suma de calificaciónes de una lista de materias como una relación de dos argumentos:

```
total(Lista_de_Materias,Suma_de_las_calificaciones).
```

La relación puede ser programada como:

```
total([],0).                              % Lista vacia de materias

total([Materia|Lista],Suma):-
     total(Lista,Resto),
     calif_materia_alumno(Materia,Calificacion),
     (not(Calificacion==cursando),
      Suma is Calificacion + Resto,
      !
      ;Suma is Resto
     ).
```

Podemos encontrar el valor de la suma con la siguiente consulta:

```
?-alumno(_,Persona,_,_,Materias),total(Materias,Suma).
```

Ahora necesitamos una relación que nos cuente las materias cursadas. Podemos definir la relación

```
cursadas(Lista_de_Materias,Cant_Mat_ya_cursadas)
```

del siguiente modo:

```
cursadas([],0).
cursadas([Materia|Resto],N),
     cursadas(Resto,N1),
     calif_materia_alumno(Materia,Calificacion),
     (not(Calif==cursando),
      N is 1 + N1,
      !
      ;N is N1
     ).
```

Ahora sí podemos obtener el promedio de un alumno con la consulta:

```
?-alumno(_,Persona,_,_,Materias),
     total(Materias,Suma),
     cursadas(Materias,N),
     Promedio is Suma / N.
```

Busquemos ahora un mayor refinamiento de nuestras utilerías.

5.3.4 Abstracción de Datos

La *abstracción de datos* puede ser vista como el proceso de organizar varias piezas de información en unidades naturales (quizás jerárquicamente), estructurando así la información en alguna forma conceptualmente semántica. Cada una de estas unidades de información debe ser accesada de manera sencilla dentro del programa. Idealmente, todos los detalles de la implantación de la estructura de datos debería ser invisible para el usuario de la estructura - el programador de aplicaciones podrá entonces concentrarse solamente en los objetos y las relaciones que existen entre ellos. El objetivo de este proceso es hacer posible el uso de la información sin que el programador tenga que pensar en los detalles de como la información es representada en la computadora.

Veamos como podemos conseguir este objetivo en Prolog. Consideremos nuestro ejemplo nuevamente. Cada alumno es una colección de piezas de información. Estas piezas están agrupadas en unidades naturales como (datos)`personales` y `materia`, de tal modo que puedan ser tratados a su vez como objetos individuales. Como vimos, cada alumno es representado como un objeto estructurado. Definamos ahora algunas relaciones que permitan que el usuario pueda accesar componentes individuales de un alumno sin conocer la estructura en que esta almacenado. Tales relaciones pueden ser llamadas *selectores* puesto que seleccionan componentes particulares. El nombre de la relación selector será el nombre del componente a ser seleccionado. La relación tendrá dos argumentos: el objeto que contiene al componente, y el componente mismo:

```
relacion_selector(Objeto,Componente_seleccionado)
```

Aquí hay varios selectores para la estructura `alumno`:

```
num_cta(alumno(Num_cta,_,_,_,_),Num_cta).

datos_pers(alumno(_,Datos_pers,_,_,_),Datos_pers).

materias(alumno(_,_,_,_,Lista_materias),Lista_materias).
```

También podemos definir selectores para materias específicas:

```
primera_materia(Alumno,Primera):-
     materias(Alumno,[Primera|_]).
```

```
segunda_materia(Alumno,Segunda):-
     materias(Alumno,[_,Segundo|_]).
...
```

Podemos generalizar esto al seleccionar a la n-ésima materia:

```
n_materia(N,Alumno,Materia):-
     materias(Alumno,Lista_materias),
     n_miembro(N,Lista_materias,Materia).

n_miembro(1,[X|L],X).                    % X es el 1er. miembro
n_miembro(N,[Y|L],X):-                   % o está en el resto.
     N1 is N - 1,
     n_miembro(N1,L,X).
```

Otro objeto interesante son los datos personales del alumno. Algunos de los selectores serían:

```
nombres(personales(Nombre,_,_,_),Nombre).

apellido_paterno(personales(_,Paterno,_,_),Paterno).

fecha_nac(personales(_,_,_,Fecha),Fecha).

dia_nac(fecha(Dia,_,_),Dia).

mes_nac(fecha(_,Mes,_),Mes).
```

¿Cómo podemos beneficiarnos de las relaciones selectores? Una vez definidas, podemos ahora olvidarnos de la forma específica en que la información estructurada es representada. Para crear y manipular esta información, sólo debemos de conocer los nombres de las relaciones selectores y usarlas en el resto del programa. En el caso de representaciones complicadas, es más sencillo este método a estar siempre refiriéndose a la representación de manera explícita. En nuestro ejemplo, el usuario no necesita saber que las materias son representadas por una lista. Por ejemplo, deseamos expresar que Juan Pérez es un alumno que nació en diciembre y que cursa Química y Física. Usando las relaciones selector anteriores se definiría así:

```
nombres(Persona,'Juan'),apellido_paterno(Persona,'Pérez'),
     mes_nac(Fecha,'diciembre'),fecha_nac(Persona,Fecha),
     datos_pers(Alumno,Persona),nombre_mat(Mat1,'física'),
     nombre_mat(Mat2,'química'),n_materia(1,Alumno,Mat1),
     n_materia(2,Alumno,Mat2).
```

Como resultado, las variables involucradas serán instanciadas como:

```
Persona=personales('Juan','Pérez',_,fecha(_,'diciembre',_))
Fecha=fecha(_,'diciembre',_)
Mat1=materia('fisica',_)
Mat2=materia('química',_)
Alumno=alumno(_,personales('Juan','Pérez',_,fecha(_,'diciembr
e',_)),_,_,[materia('fisica',_),materia('química',_)])
```

Cuando sea el caso de una consulta, como en `cursa('Juan','física')` debemos de indicarle también cuales son los selectores que debe de usar para poder comparar con los campos que están siendo pasados como parámetros, por ejemplo:

```
cursa(nombres,Nombres,nom_mat,Nombre_materia):-
      datos_pers(Alumno,Persona),
      nombres(Persona,Nombres),
      n_materia(N,Alumno,Materia)
      nombre_mat(Materia,Nombre_materia).
```

El uso de las relaciones selector también facilita la modificación de los programas de aplicación. Si quisieramos mejorar la eficiencia de un programa cambiando la representación de los datos, todo lo que tendríamos que hacer es cambiar las definiciones de las relaciones selector, y el resto del programa trabajaría con la nueva representación sin necesidad de hacerle cambio alguno.

5.4 Juntando las piezas

Procedamos ahora con la síntesis de los elementos que hemos analizado en este capítulo. Partiremos de la gramática que posee los accidentes del verbo como parámetros. Después a la izquierda de los mismos colocaremos los argumentos que devuelvan el significado de la consulta de entrada. Entonces tendremos lo siguiente:

```
pregunta(Significado)-->
      signo_interrogacion_abierto,
      pronombre_interrogativo,
      enunciado(Significado,Voz,Genero,Numero),
      signo_interrogación_cerrado.

signo_interrogación_abierto-->[¿].

signo_interrogacion_cerrado-->[?].

pronombre_interrogativo-->[Qué].
```

Como resultado, las variables involucradas serán instanciadas como:

```
Persona=personales('Juan','Pérez',_,fecha(_,'diciembre',_))
Fecha=fecha(_,'diciembre',_)
Mat1=materia('física',_)
Mat2=materia('química',_)
Alumno=alumno(_,personales('Juan','Pérez',_,fecha(_,'diciembr
e',_)),_,_,[materia('física',_),materia('química',_)])
```

Cuando sea el caso de una consulta, como en `cursa('Juan','física')` debemos de indicarle también cuales son los selectores que debe de usar para poder comparar con los campos que están siendo pasados como parámetros, por ejemplo:

```
cursa(nombres,Nombres,nom_mat,Nombre_materia):-
      datos_pers(Alumno,Persona),
      nombres(Persona,Nombres),
      n_materia(N,Alumno,Materia)
      nombre_mat(Materia,Nombre_materia).
```

El uso de las relaciones selector también facilita la modificación de los programas de aplicación. Si quisieramos mejorar la eficiencia de un programa cambiando la representación de los datos, todo lo que tendríamos que hacer es cambiar las definiciones de las relaciones selector, y el resto del programa trabajaría con la nueva representación sin necesidad de hacerle cambio alguno.

5.4 Juntando las piezas

Procedamos ahora con la síntesis de los elementos que hemos analizado en este capítulo. Partiremos de la gramática que posee los accidentes del verbo como parámetros. Después a la izquierda de los mismos colocaremos los argumentos que devuelvan el significado de la consulta de entrada. Entonces tendremos lo siguiente:

```
pregunta(Significado)-->
      signo_interrogacion_abierto,
      pronombre_interrogativo,
      enunciado(Significado,Voz,Genero,Numero),
      signo_interrogación_cerrado.

signo_interrogación_abierto-->[¿].

signo_interrogacion_cerrado-->[?].

pronombre_interrogativo-->[Qué].
```

```
enunciado(Significado,Voz,Genero,Numero)-->
     sujeto(Voz,Genero,Numero),
     predicado(Significado,Voz,Genero,Numero).

sujeto(pasiva,Genero,Numero)-->
     receptor(pasiva,Genero,Numero).

predicado(Significado,Voz,Genero,Numero)-->
     verbo(Actor,Significado,Voz,Genero,Numero),
     objeto_directo(Actor,Voz,Genero1,Numero1).

receptor(pasiva,femenino,singular)-->[materia].
receptor(pasiva,femenino,plural)-->[materias].

verbo(Campo_actor,Actor,
          cursa(Campo_actor,Actor,materias,Receptor),Voz,Genero,Numero
     -->[cursa].

objeto_directo(Campo_actor,Actor,pasiva,Genero,Numero)-->
     instancia_actor(Campo_actor,Actor,pasiva,Genero,Numero).

instancia_actor(nombres,'Juan',pasiva,Genero,Numero)-->
     [Juan].
```

Esta gramática nos permitirá realizar la consulta:

```
?-pregunta(S,['¿','Qué',materias,cursa,'Juan',?],[]).
S=cursa(nombres,'Juan',materias,Receptor)
```

que luego evaluaremos con la regla de consulta:

```
cursa(nombres,Nombres,materias,Materia):-
     datos_pers(Alumno,Persona),
     nombres(Persona,Nombres),
     materias(Alumno,Materias).
```

Este sistema nos ilustra brevemente de como podemos construir una serie de módulos de un DBMS para hacerlo más eficiente.

En el apéndice se encuentran los listados completos del sistema en lenguaje PROLOG.

Capítulo 6

Conclusiones

He intentado cubrir los tópicos fundamentales que en materia de lógica matemática aplicada a las bases de datos se han desarrollado. Contemplando esta tesis ahora que está terminada, veo que solo he escrito una introducción al tema. En la tecnología de bases de datos existen tres enfoques fundamentales: el relacional, el jerárquico y el de red; en la Lógica existe una gran cantidad de sistemas formales que están contenidos en varias corrientes de estudio. Tan solo he hablado del modelo relacional de base de datos y de las lógicas clásicas de proposiciones y de predicados que se encuentran dentro de la lógica de la escuela formalista de David Hilbert.

Durante los últimos años se ha trabajado mucho para relacionar conceptos de base de datos con las nociones de la lógica formal, por ende, también existen varios enfoques. Existe el enfoque de interpretación de una teoría, que considera al esquema como una teoría y a la base de datos como una estructura relacional que, para ser válida, debe ser un modelo de la teoría. Este enfoque puede utilizarse para formalizar sistemas de base de datos no deductivos convencionales.

El segundo enfoque considera que el esquema junto con la base de datos son una teoría y que el mundo real es la estructura. Este es el enfoque que se adopto para construir el sistema FGH, adecuado para el lenguaje PROLOG.

Este enfoque tiene la ventaja de permitir que tanto el esquema como la base de datos sean actualizados fácilmente. Sin embargo, no permite formalizar ciertos tipos de restricciones de integridad comúnmente requeridas en aplicaciones de base de datos.

En concreto, PROLOG está restringido a cláusulas de Horn y utiliza la metaregla "fallo (`fail` ó `!`) para demostrar como negación" para implantar la suposición del

mundo cerrado. Además las cláusulas que constan de literales negativas se consideran solamente como objetivos que hay que demostrar y no como aseveraciones (**assert**). Esto quiere decir que en PROLOG no es posible expresar hechos negativos como axiomas propios. En consecuencia, no es posible construir una teoría inconsistente en PROLOG y, por lo tanto, la noción de restricción de integridad no está respaldada.

Hay además dentro de las bases de datos deductivas un tercer enfoque, que descarta las suposiciones descritas en la sección 2.3, donde el esquema está considerado como una teoría y la base de datos como una estructura relacional incompleta, que es una subestructura de la estructura del mundo real que representa. Al parecer este enfoque se orienta hacia cierto tipo de sistemas de base de conocimientos (generalización de los sistemas expertos).

El primero y segundo enfoques se han utilizado ya extensamente para facilitar el diseño de algoritmos que verifiquen la integridad y la evaluación de consultas para ciertos tipos de sistemas de base de datos (un ejemplo es el sistema FGH). Se espera que, cuando el tercer enfoque esté más desarrollado, proporcione orientación para el diseño de algoritmos de una clase más amplia de sistemas de base de datos.

Al haber construido el sistema FGH he encontrado que el enfoque elegido permite desarrollar sistemas de base de datos con algunas capacidades deductivas. Dicha experiencia me lleva a concluir que esta es un área promisoria donde existe el compromiso de buscar nuevos horizontes: conocer otros tipos de lógicas dentro de la corriente formalista (lógicas no clásicas), estudiar la corriente intuicionista de la lógica, analizar los modelos de bases de datos buscando nuevas perspectivas, en general, buscar nuevas alternativas. Existen trabajos como [ADIB82] Y [REIT84] donde analizan con profundidad los caminos que se pueden seguir. Algunos son:

- *Optimizar la evaluación de consultas basada en conocimiento semántico*, el cual es necesario, en general, para el acceso interactivo a las bases de datos y especialmente en el contexto de un lenguaje natural.
- *Buscar criterios y métodos para la elección, de entre conjuntos de restricciones de integridad equivalentes, de un buen conjunto*, donde por "bueno" entendemos conjuntos de restricciones que sean fáciles de checar y de mantener.
- *Buscar criterios para decidir cuales reglas deben ser usadas como restricciones de integridad y cuales deben ser reglas de deducción.*
- *Buscar medios más eficientes para detectar violaciones de restricciones de integridad.*
- *Hacer más flexibles algunas de las condiciones para que una fórmula sea una restricción de integridad.*
- *Integrar lenguajes de manipulación de datos en lenguajes de programación.* Al hablar de bases de datos deductivas, usualmente uno considera al

componente deductivo como parte del DBMS. Sin embargo, es posible hacer interfase entre un DBMS (convencional ó deductivo) con un lenguaje de programación lógica, como PROLOG. Tal integración es más sencilla cuando el lenguaje del DBMS es basado en el cálculo de predicados; una integración completa se logra cuando el DBMS aparece como la parte del sistema (ó ambiente) de programación especializada en la manipulación de hechos.

El camino es largo y promisorio, hay que recorrerlo y disfrutarlo momento a momento, no solamente cuando se llega a las metas. Lo importante es que el camino tenga corazón. Sigamos el camino.

Anexo

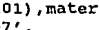
Archivo fgh.ari

```
carga:-
     cls,nl,nl,nl,nl,nl,tab(20),
     write('Favor de esperar mientras carga...'),
     consult('a:base'),
     consult('a:gram'),
     consult('a:q'),
     consult('a:interfaz'),
     cls.
```

Archivo base.ari

```
% Como se comentó en la sección 5.3.4,si la estructura de la
% base de datos es modificada, solo se deben de actualizar
% las relaciones selector para que el programa continue
% funcionando. El esquema de nuestra base de datos es el
% siguiente:
% asignatura(Clave,Nombre,Creditos)
% alumno(Num_cta,personales(Nombres,Paterno,Materno,
%     fecha(Día,Mes,Año)),Esc_fac,Carrera,Lista_materias).
% en donde cada materia esta compuesta por:
% materia(Clave,Calificacion)
% Las relaciones selector son las siguientes

clave_asignatura(asignatura(Clave,_,_),Clave).
nombre_asignatura(asignatura(_,Nombre,_),Nombre).
creditos_asignatura(asignatura(_,_,Creditos),Creditos).

no_cta_alumno(alumno(Num_cta,_,_,_,_),Num_cta).
datos_pers_alumno(alumno(_,Datos_pers,_,_,_),Datos_pers).
esc_fac_alumno(alumno(_,_,Esc_fac,_,_),Esc_fac).
carrera_alumno(alumno(_,_,_,Carrera,_),Carrera).
materias_alumno(alumno(_,_,_,_,Materias),Materias).

nombres_alumno(personales(Nombre,_,_,_),Nombre).
paterno_alumno(personales(_,Paterno,_,_),Paterno).
materno_alumno(personales(_,_,Materno,_),Materno).
fecha_nac_alumno(personales(_,_,_,Fecha),Fecha).
```

```
dia_nac_alumno(fecha(Dia,_,_),Dia).
mes_nac_alumno(fecha(_,Mes,_),Mes).
'año_nac_alumno'(fecha(_,_,A),A).

n_materia(N,Alumno,Materia):-
     materias_alumno(Alumno,Lista_materias),
     n_miembro(N,Lista_materias,Materia).

n_miembro(1,[X|L],X).
n_miembro(N,[Y|L],X):-
     N1 is N - 1,
     n_miembro(N1,L,X).

clave_materia_alumno(materia(Clave,_,_),Clave).
calif_materia_alumno(materia(_,Calif,_),Calif).
grupo_materia_alumno(materia(_,_,Grupo),Grupo).

% Para efectos de demostración del funcionamiento del
% sistema son necesarias algunas instancias (registros) en
% la base de datos. Estas aparecen a continuacion

asignatura(1100,'física',08).
asignatura(1101,'química',08).
asignatura(1102,literatura,08).
asignatura(1103,'álgebra',10).
asignatura(1104,'cálculo',10).

alumno('8656184-6',
personales('Juan','Pérez','López',fecha(25,12,1965)),240,21,
[materia(1100,8,101),materia(1101,7,101),materia(1102,9,101),
materia(1103,8,101),materia(1104,8,101)]).
alumno('8476190-7',
personales('Jaime','Colín','Morales',fecha(18,3,1968)),240,21
,[materia(1100,7,102),materia(1101,8,101),materia(1102,6,102)
,materia(1104,8,102)]).
alumno('8528492-0',
personales('Carmen','López','González',fecha(29,3,1967)),240,
21,[materia(1100,10,103),materia(1101,8,103),materia(1102,9,
103),materia(1103,9,103),materia(1104,8,103)]).
```



Archivo gram.arl

```
% Gramática del sistema

% Reglas generales

pregunta(Significado,Voz,Genero,Numero)-->
     signo_interrogacion_abierto,
     pronombre_interrogativo,
     enunciado(Significado,Voz,Genero,Numero),
     signo_interrogacion_cerrado.

signo_interrogacion_abierto-->['¿'].

pronombre_interrogativo-->['Qué'].

signo_interrogacion_cerrado-->['?'].

enunciado(Significado,Voz,Genero,Numero)-->
     sujeto(Campo,Instancia,Significado,Voz,Genero1,Numero1),
     predicado(Campo,Instancia,_,Voz,Genero,Numero).

% caso 1a: nombres tiene 8656184-6

sujeto(Campo,Instancia,Significado,pasiva,Genero,Numero)-->
     receptor(Campo,Instancia,Significado,Genero,Numero).

receptor(Campo,Instancia,nombres(Campo,Instancia,Var),
     masculino,plural)-->[nombres].

predicado(Campo,Instancia,Significado,Voz,Genero,Numero)-->
     verbo(_,_,_,Voz,Genero,Numero),
     objeto_directo(Campo,Instancia,_,Voz,Genero,Numero).

verbo(_,_,_,pasiva,_,singular)-->[tiene].

objeto_directo(Campo,Instancia,_,pasiva,Genero,Numero)-->
     instancia_actor(Campo,Instancia,_,pasiva,Genero,Numero).

instancia_actor(no_cta,'8656184-6',_,pasiva,
     masculino,singular)-->['8656184-6'].
instancia_actor(no_cta,'8476190-7',_,pasiva,
     masculino,singular)-->['8476190-7'].
instancia_actor(no_cta,'8528492-0',_,pasiva,
     masculino,singular)-->['8528492-0'].

% caso 1b: promedio tiene Juan
```

```
receptor(Campo,Instancia,promedio(Campo,Instancia,Var),
     masculino,singular)-->[promedio].

instancia_actor(nombres,'Juan',_,pasiva,masculino,singular)
     -->['Juan'].
instancia_actor(nombres,'Jaime',_,pasiva,masculino,singular)
     -->['Jaime'].
instancia_actor(nombres,'Carmen',_,pasiva,masculino,singular)
     -->['Carmen'].
instancia_actor(paterno,'Pérez',_,pasiva,masculino,singular)
     -->['Pérez'].
instancia_actor(paterno,'Colín',_,pasiva,masculino,singular)
     -->['Colín'].
instancia_actor(paterno,'López',_,pasiva,masculino,singular)
     -->['López'].
instancia_actor(materno,'López',_,pasiva,masculino,singular)
     --> ['López'].
instancia_actor(materno,'Morales',_,pasiva,
     masculino,singular)-->['Morales'].
instancia_actor(materno,'González',_,pasiva,
     masculino,singular)-->['González'].

% caso 1c: número de cuenta tiene Juan

sujeto(Campo,Instancia,Significado,pasiva,Genero,Numero)-->
     receptor(Campo,Instancia,Significado,Genero,Numero),
     modificador_mediato(Campo,Instancia,Significado).

modificador_mediato(Campo,Instancia,Significado)-->
     nexo_preposicion(Campo,Instancia,Significado),
     adnominal(Campo,Instancia,Significado).

receptor(Campo,Instancia,no_cta(Campo,Instancia,Var),
     masculino,singular)-->['número']

nexo_preposicion(Campo,Instancia,
     no_cta(Campo,Instancia,Var))-->[de].

adnominal(Campo,Instancia,
     no_cta(Campo,Instancia,Var))-->[cuenta].

% caso 1d: carrera tiene Juan

receptor(Campo,Instancia,carrera(Campo,Instancia,Var),
     femenino,singular)-->[carrera].
```

```
% caso 1e: apellido paterno tiene Juan

sujeto(Campo,Instancia,Significado,pasiva,Genero,Numero)-->
     receptor(Campo,Instancia,Significado,Genero,Numero),
     modificador_inmediato(Campo,Instancia,Significado,
          Genero,Numero).

modificador_inmediato(Campo,Instancia,Significado,Genero,
     Numero)-->adjetivo(Campo,Instancia,Significado,Genero,Numero)
.

receptor(Campo,Instancia,paterno(Campo,Instancia,Var),
     masculino,singular)-->[apellido].
receptor(Campo,Instancia,materno(Campo,Instancia,Var),
     masculino,singular)-->[apellido].

adjetivo(Campo,Instancia,paterno(Campo,Instancia,Var),
     masculino,singular)-->[paterno].

% caso 1e: apellido materno tiene Juan

adjetivo(Campo,Instancia,materno(Campo,Instancia,Var),
     masculino,singular)-->[materno].

% caso 2a: materias cursa Juan en el grupo 103

enunciado(Significado,Voz,Genero,Numero)-->
     sujeto(Campo1,Instancia1,Campo2,Instancia2,Significado,
          Voz,Genero1,Numero1),
     predicado(Campo1,Instancia1,Campo2,Instancia2,_,Voz,
          Genero,Numero).

sujeto(Campo1,Instancia1,Campo2,Instancia2,Significado,
          pasiva,Genero,Numero)-->
     receptor(Campo1,Instancia1,Campo2,Instancia2,Significado,
          Genero,Numero).

receptor(Campo,Instancia,materias(Campo,Instancia,Var),
     femenino,plural)-->[materias].

receptor(Campo1,Instancia1,Campo2,Instancia2,materias(
          Campo1,Instancia1,Var,Campo2,Instancia2),femenino,plural)
     -->[materias].

predicado(Campo1,Instancia1,Campo2,Instancia2,Significado,
          Voz,Genero,Numero)-->
     verbo(_,_,_,Voz,Genero,Numero),
```

```
     objeto_directo(Campo1,Instancia1,_,Voz,Genero,Numero),
     nexo_subordinante,
     frase_subordinada(Campo2,Instancia2,_).

verbo(_,_,_,pasiva,_,singular)-->[cursa].

nexo_subordinante-->[en].

frase_subordinada(Campo2,Instancia2,_)-->
     articulo,
     instancia_grupo(Campo2,Instancia2,_).

frase_subordinada(Campo2,Instancia2,_)-->
     articulo,
     nucleo_subordinada,
     instancia_grupo(Campo2,Instancia2,_).

articulo-->[el].

nucleo_subordinada-->[grupo].

instancia_grupo(grupo,'103',_)-->[103].
instancia_grupo(grupo,'102',_)-->[102].
instancia_grupo(grupo,'101',_)-->[101].

% caso 2b: alumnos cursan física en el grupo 103

sujeto(Campo,Instancia,Significado,activa,Genero,Numero)-->
     actor(Campo,Instancia,Significado,Genero,Numero).

sujeto(Campo1,Instancia1,Campo2,Instancia2,Significado,
          activa,Genero,Numero)-->
     actor(Campo1,Instancia1,Campo2,Instancia2,Significado,
          Genero,Numero).

actor(Campo,Instancia,alumnos(Campo,Instancia,Var),masculino,
     plural)-->[alumnos].

actor(Campo1,Instancia1,Campo2,Instancia2,
     alumnos(Campo1,Instancia1,Var,Campo2,Instancia2),
     masculino,plural)-->[alumnos].

verbo(_,_,_,activa,_,singular)-->[cursan].

objeto_directo(Campo,Instancia,_,activa,Genero,Numero)-->
     instancia_receptor(Campo,Instancia,_,activa,Genero,Numero).
```

```
instancia_receptor(nombre_asignatura,'física',_,activa,
     femenino,singular)-->['física'].
instancia_receptor(nombre_asignatura,'química',_,activa,
     femenino,singular)-->['química'].
instancia_receptor(nombre_asignatura,'literatura',_,activa,
     femenino,singular)-->['literatura'].
instancia_receptor(nombre_asignatura,'álgebra',_,activa,
     femenino,singular)-->['álgebra'].
instancia_receptor(nombre_asignatura,'cálculo',_,activa,
     femenino,singular)-->['cálculo'].
instancia_receptor(clave_asignatura,1100,_,activa,
     femenino,singular)-->[1100].
instancia_receptor(clave_asignatura,1101,_,activa,
     femenino,singular)-->[1101].
instancia_receptor(clave_asignatura,1102,_,activa,
     femenino,singular)-->[1102].
instancia_receptor(clave_asignatura,1103,_,activa,
     femenino,singular)-->[1103].
instancia_receptor(clave_asignatura,1104,_,activa,
     femenino,singular)-->[1104].

% caso 3: calificación tiene Juan en física

receptor(Campo1,Instancia1,Campo2,Instancia2,calif(Campo1,
          Instancia1,Var,Campo2,Instancia2),femenino,singular)
     -->['calificacin'].

frase_subordinada(Campo2,Instancia2,_)-->
     instancia_receptor(Campo2,Instancia2,_,_,_,_).
```

Archivo q.arl

```
% Consultas a la base de datos como cláusulas de Horn

% no_cta/3

respuesta(no_cta(_,_,R),R).

no_cta(nombres,Nombres,No_cta):-
     datos_pers_alumno(Alumno,Datos),
     nombres_alumno(Datos,Nombres),
     no_cta_alumno(Alumno,No_cta),
     Alumno.
```

```
no_cta(paterno,Paterno,No_cta):-
      datos_pers_alumno(Alumno,Datos),
      paterno_alumno(Datos,Paterno),
      no_cta_alumno(Alumno,No_cta),
      Alumno.

no_cta(materno,Materno,No_cta):-
      datos_pers_alumno(Alumno,Datos),
      materno_alumno(Datos,Materno),
      no_cta_alumno(Alumno,No_cta),
      Alumno.

% nombres/3

respuesta(nombres(_,_,R),R).

nombres(no_cta,No_cta,Nombres):-
      no_cta_alumno(Alumno,No_cta),
      datos_pers_alumno(Alumno,Datos),
      nombres_alumno(Datos,Nombres),
      Alumno.

nombres(paterno,Paterno,Nombres):-
      datos_pers_alumno(Alumno,Datos),
      paterno_alumno(Datos,Paterno),
      nombres_alumno(Datos,Nombres),
      Alumno.

nombres(materno,Materno,Nombres):-
      datos_pers_alumno(Alumno,Datos),
      materno_alumno(Datos,Materno),
      nombres_alumno(Datos,Nombres),
      Alumno.

% paterno/3

respuesta(paterno(_,_,R),R).

paterno(no_cta,No_cta,Paterno):-
      no_cta_alumno(Alumno,No_cta),
      datos_pers_alumno(Alumno,Datos),
      paterno_alumno(Datos,Paterno),
      Alumno.

paterno(nombres,Nombres,Paterno):-
      datos_pers_alumno(Alumno,Datos),
```

```
      nombres_alumno(Datos,Nombres),
      paterno_alumno(Datos,Paterno),
      Alumno.

paterno(materno,Materno,Paterno):-
      datos_pers_alumno(Alumno,Datos),
      materno_alumno(Datos,Materno),
      paterno_alumno(Datos,Paterno),
      Alumno.

% materno/3

respuesta(materno(_,_,R),R).

materno(no_cta,No_cta,Materno):-
      no_cta_alumno(Alumno,No_cta),
      datos_pers_alumno(Alumno,Datos),
      materno_alumno(Datos,Materno),
      Alumno.

materno(nombres,Nombres,Materno):-
      datos_pers_alumno(Alumno,Datos),
      nombres_alumno(Datos,Nombres),
      materno_alumno(Datos,Materno),
      Alumno.

materno(paterno,Paterno,Materno):-
      datos_pers_alumno(Alumno,Datos),
      paterno_alumno(Datos,Paterno),
      materno_alumno(Datos,Materno),
      Alumno.

% carrera/3

respuesta(carrera(_,_,R),R).

carrera(no_cta,No_cta,Carrera):-
      no_cta_alumno(Alumno,No_cta),
      carrera_alumno(Alumno,Carrera),
      Alumno.

carrera(nombres,Nombres,Carrera):-
      datos_pers_alumno(Alumno,Datos),
      nombres_alumno(Datos,Nombres),
      carrera_alumno(Alumno,Carrera),
      Alumno.
```

```
carrera(paterno,Paterno,Carrera):-
     datos_pers_alumno(Alumno,Datos),
     paterno_alumno(Datos,Paterno),
     carrera_alumno(Alumno,Carrera),
     Alumno.

carrera(materno,Materno,Carrera):-
     datos_pers_alumno(Alumno,Datos),
     materno_alumno(Datos,Materno),
     carrera_alumno(Alumno,Carrera),
     Alumno.

% promedio/3

respuesta(promedio(_,_,R),R).

total([],0).
total([Materia|Lista],Suma):-
     total(Lista,Resto),
     calif_materia_alumno(Materia,Calificacion),
     (not(Calificacion==cursando),
      Suma is Calificacion + Resto,
      !
      ; Suma is Resto
     ).

cursadas([],0).
cursadas([Materia|Resto],N):-
     cursadas(Resto,N1),
     calif_materia_alumno(Materia,Calificacion),
     (not(Calificacion==cursando),
      N is N1 + 1,
      !
      ; N is N1
     ).

prom(Materias,Promedio):-
     total(Materias,Suma),
     cursadas(Materias,N),
     Promedio is Suma / N.

promedio(no_cta,No_cta,Promedio):-
     no_cta_alumno(Alumno,No_cta),
     materias_alumno(Alumno,Materias),
```

```
      Alumno,
      prom(Materias,Promedio).

promedio(nombres,Nombres,Promedio):-
      datos_pers_alumno(Alumno,Datos),
      nombres_alumno(Datos,Nombres),
      materias_alumno(Alumno,Materias),
      Alumno,
      prom(Materias,Promedio).

promedio(paterno,Paterno,Promedio):-
      datos_pers_alumno(Alumno,Datos),
      paterno_alumno(Datos,Paterno),
      materias_alumno(Alumno,Materias),
      Alumno,
      prom(Materias,Promedio).

promedio(materno,Materno,Promedio):-
      datos_pers_alumno(Alumno,Datos),
      materno_alumno(Datos,Materno),
      materias_alumno(Alumno,Materias),
      Alumno,
      prom(Materias,Promedio).

% materias/3

respuesta(materias(_,_,R),R).

materias(no_cta,No_cta,Materias):-
      no_cta_alumno(Alumno,No_cta),
      materias_alumno(Alumno,Materias),
      Alumno.

materias(nombres,Nombres,Materias):-
      datos_pers_alumno(Alumno,Datos),
      nombres_alumno(Datos,Nombres),
      materias_alumno(Alumno,Materias),
      Alumno.

materias(paterno,Paterno,Materias):-
      datos_pers_alumno(Alumno,Datos),
      paterno_alumno(Datos,Paterno),
      materias_alumno(Alumno,Materias),
      Alumno.
```

```
materias(materno,Materno,Materias):-
     datos_pers_alumno(Alumno,Datos),
     materno_alumno(Datos,Materno),
     materias_alumno(Alumno,Materias),
     Alumno.

% materias/5

respuesta(materias(_,_,R,_,_),R).

materias(no_cta,No_cta,Materias_grupo,grupo,Grupo):-
     no_cta_alumno(Alumno,No_cta),
     materias_alumno(Alumno,Materias),
     Alumno,
     materias_grupo(Grupo,Materias,Materias_grupo).

materias(nombres,Nombres,Materias_grupo,grupo,Grupo):-
     datos_pers_alumno(Alumno,Datos),
     nombres_alumno(Datos,Nombres),
     materias_alumno(Alumno,Materias),
     Alumno,
     materias_grupo(Grupo,Materias,Materias_grupo).

materias(paterno,Paterno,Materias_grupo,grupo,Grupo):-
     datos_pers_alumno(Alumno,Datos),
     paterno_alumno(Datos,Paterno),
     materias_alumno(Alumno,Materias),
     Alumno,
     materias_grupo(Grupo,Materias,Materias_grupo).

materias(materno,Materno,Materias_grupo,grupo,Grupo):-
     datos_pers_alumno(Alumno,Datos),
     materno_alumno(Datos,Materno),
     materias_alumno(Alumno,Materias),
     Alumno,
     materias_grupo(Grupo,Materias,Materias_grupo).

materias_grupo(_,[],[]).
materias_grupo(Grupo,[Materia|Resto],[Materia|Materias]):-
     grupo_materia_alumno(Materia,Grupo),
     materias_grupo(Grupo,Resto,Materias),
     !.
materias_grupo(Grupo,[Materia|Resto],Materias):-
     materias_grupo(Grupo,Resto,Materias).
```

```
% alumnos/3

respuesta(alumnos(_,_,R),R).

miembro(X,[X|Xs]).
     miembro(X,[Y|Xs]):-
     miembro(X,Xs).

alumnos(clave_asignatura,Clave_asignatura,Alumno):-
     clave_materia_alumno(Materia,Clave_asignatura),
     materias_alumno(Alumno,Materias),
     Alumno,
     miembro(Materia,Materias).

alumnos(nombre_asignatura,Nombre_asignatura,Alumno):-
     nombre_asignatura(Asignatura,Nombre_asignatura),
     clave_asignatura(Asignatura,Clave_asignatura),
     Asignatura,
     clave_materia_alumno(Materia,Clave_asignatura),
     materias_alumno(Alumno,Materias),
     Alumno,
     miembro(Materia,Materias).

% alumnos/5

respuesta(alumnos(_,_,R,_,_),R).

alumnos(clave_asignatura,Clave_asignatura,
          Alumno,grupo,Grupo):-
     clave_materia_alumno(Materia,Clave_asignatura),
     grupo_materia_alumno(Materia,Grupo),
     materias_alumno(Alumno,Materias),
     Alumno,
     miembro(Materia,Materias).

alumnos(nombre_asignatura,Nombre_asignatura,
          Alumno,grupo,Grupo):-
     nombre_asignatura(Asignatura,Nombre_asignatura),
     clave_asignatura(Asignatura,Clave_asignatura),
     Asignatura,
     clave_materia_alumno(Materia,Clave_asignatura),
     grupo_materia_alumno(Materia,Grupo),
     materias_alumno(Alumno,Materias),
     Alumno,
     miembro(Materia,Materias).
```

Archivo interfaz.ari

```
% Interfase del sistema

eval(Pregunta,Significado):-
     pregunta(Significado,Voz,Genero,Numero,Pregunta,[]),
          Significado.

eval1(Pregunta,Respuesta):-
     eval(Pregunta,Significado),
     respuesta(Significado,Respuesta).

fgh:-
     cls,
     writeln('Por favor digite su consulta'),
     write(''),
     getsentence(Pregunta),
     nl,
     writeln(Pregunta),
     eval(Pregunta,Significado),
     writeln(Significado),
     respuesta(Significado,Respuesta),
     writeln(Respuesta).

% El siguiente código fue tomado de [STER86]

getsentence(Wordlist):-
     get0(Char),
     getrest(Char,Wordlist).

getrest(63,[?]):-!.
getrest(32,Wordlist):-
     getsentence(Wordlist).
getrest(Letter,[Word|Wordlist]):-
     getletters(Letter,Letters,Nextchar),
     name(Word,Letters),
     getrest(Nextchar, Wordlist).

getletters(63,[],63):-!.
getletters(32,[],32):-!.
getletters(Let,[Let|Letters],Nextchar):-
     get0(Char),
     getletters(Char,Letters,Nextchar).

writeln(A):-write(A),nl.
```

Bibliografía

[ADIB82] Adiba, M., et al, "Bases de donnes: Nouvelles perspectives," Rapport du groupe BD3 ADI-INRIA, París (1982)

[BOWE81] Bowen, D.L., "DECSystem-10 Prolog User's Manual," Departament of Artificial Inteligence, University of Edimburgh (1981).

[CHAN73] Chang, C. L. y Lee R. C., "Symbolic Logic and Mechanical Theorem Proving," Academic Press, New York (1973).

[CHOM59] Chomsky, Avram Noam, "On certain Formal Properties of Grammars," *Information and Control* núm. 2, págs. 137-167 (1959).

[CHOM69] Chomsky, Avram Noam, "Syntactic Structures," Mouton, The Hague (1969).

[CLAR78] Clark, K. S., "Negation as failure," en *Logic and Databases*, Gallaire, Hervé y Minker, Jack eds., Plenum Press, New York (1978).

[CLOC81] Clocksin, W. F. y Mellish, C. S. "Programming in Prolog," Springer-Verlag, Berlín (1981).

[CODD70] Codd, E. F. "A relational model for large shared data banks," *Comm. ACM*, vol. 13, núm. 6, págs. 377-387 (1970)

[COLM72] Colmerauer, Alain, Kanoui, H., Roussel, P. y Pasero, R., "Un système de communication homme-machine en français," Rapport preliminaire. Groupe d'Intelligence Artificielle, Université d'Aix-Marseille (1972).

[DAHL80] Dahl, Veronica, "A three-valued logic for natural language computer applications," *Proc. 10th Int'l Symp. Multiple Valued Logic*, Evanston, Ill., junio (1980).

[DAHL82] Dahl, Veronica, "On database system development through logic," *ACM Transactions on Database Systems* vol. 7, núm. 1, págs. 102-123, marzo (1982).

[ENDE78] Enderton, H. B., "A mathematical introduction to logic," Academic Press, New York (1972)

[GALL84] Gallaire, Hervé, Minker, Jack, y Nicolas, Jean Marie, "Logic and Databases: a Deductive Approach," *ACM Computer Surveys*, vol. 16, núm. 2, junio (1984).

[GALL78] Gallaire, Hervé y Minker, Jack eds., "Logic and Databases," Plenum Press, New York (1978).

[HOFS79] Hofstadter, Douglas R., "Gödel, Escher y Bach: una eterna trenza dorada," Consejo Nacional de Ciencia y Tecnología, México, D. F. (1979).

[KAIN72] Kain, Richard Y., "Automata theory: Machines and Languages," McGraw-Hill, New York (1972).

[KLUZ85] Kluzniak, Feliks y Stanislaw Szpakowicz, "Prolog for programmers," Academic Press, Londres (1979).

[KOWA72] Kowalski, Robert A., "The predicate calculus as a programming language," *Proc. of the Int'l Symposium and Summer School on Mathematical Foundations of Computer Science*, Jablona, Polonia (1972).

[KOWA74] Kowalski, Robert A., "Predicate logic as programming language," *Proc. of the IFIP Congress*, Amsterdam, págs. 569-574 (1974).

[KOWA79] Kowalski, Robert A., "Algorithm = Logic + Control," *Communications on the ACM*, núm. 22, págs. 424-431 (1979).

[KOWA81] Kowalski, Robert A., "Logic as a data base language," *Proc. of the Advanced Seminar on Theoretical Issues in Data Bases*, Cetravo, Italia (1981).

[KUHN67] Kuhns, J. L., "Answering questions by computers - A logical study," Rand Memo RM 5428 PR, Rand Corp., Santa Mónica, Calif. (1967)

[MAIE83] Maier, D., "The Theory of Relational Databases," Computer Science Press, Rockville, Maryland (1983)

[MART77] Martin, James, "Organización de las Bases de Datos," Prentice Hall Hispanoamericana, México, D. F. (1977).

[MEND78] Mendelson, E., "Introduction to mathematical logic," 2a. ed., Van Nostrand-Reinhold, New York, (1978)

[MINK83] Minker, Jack, "On deductive relational databases," *Proc. of the 5th Int'l Conference on Collective Phenomena*, J. L. Lebowitz de New York Academy of Science, págs. 181-200, julio (1983).

[PERE78] Pereira, L. M., Pereira, F. C. N. y Warren, D. H. D., "User's Guide to DECSystem-10 Prolog (provisional version)," Departament of Artificial Intelligence, University of Edimburgh (1978).

[REIT78] Reiter, Raymond, "On closed world databases," en *Logic and Databases*, Gallaire, Hervé y Minker, Jack, eds., Plenum Press, New York (1978).

[REIT84] Reiter, Raymond, "Towards a logical reconstruction of relational database theory," en *On conceptual Modeling*, Brodie, M., Mylopoulos, J., y Schmidt, J. W., eds. Springer-Verlag, Berlín (1984).

[RICH83] Rich, Elaine, "Artificial Inteligence," McGraw-Hill, Singapore (1983).

[ROBI65] Robinson, J. A., "A machine oriented logic based on the resolution principle," *J. ACM*, vol. 12, núm. 1, págs. 23-41, enero (1965)

[SCHA73] Schank, R. C., "Identification of Conceptualizations underlying Natural Language," en *Computer Models of Thought and Language*, Schank, R. C. y Colby, K. M. eds. Freeman, San Francisco (1973).

[STER86] Sterling, L., y Shapiro, E., "The Art of Prolog, Advanced Programming Techniques," MIT Press, Cambridge, Massachusetts (1982)

[ULLM82] Ullman, Jeffrey D., "Principles of Database Systems," 2a. ed., Computer Press, Rockville, Maryland (1982)

[WARR77] Warren, D. H. D., "Implementing Prolog-Compiling Predicate Logic Programs," DAI Report Nos. 39-40, University of Edimburgh (1977)

[WIRT76] Wirth, Nicklaus, "Algorithms + Data Structures = Programs," Prentice Hall, Englewood Cliffs, N.J. (1976).

[WOOD70] Woods, W. A., "Transition Network Grammars for Natural Language Analysis," *Communications of the ACM*, núm. 13, págs. 591-606 (1970)