03063
1
2ej

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

UNIDAD ACADEMICA DE LOS CICLOS PROFESIONAL
Y DE POSTGRADO DEL COLEGIO DE CIENCIAS
Y HUMANIDADES

INSTITUTO DE INVESTIGACIONES EN MATEMATICAS
APLICADAS Y EN SISTEMAS

FALLA DE ORIGEN

IMPLANTACION DE UN LENGUAJE ORIENTADO A REGLAS PARA LA PROGRAMACION DE SISTEMAS EXPERTOS

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS
DE LA COMPUTACION

P R E S E N T A:

NORMA ANGELICA ARANDA DE DIOS

MEXICO, 1991.

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

DERECHOS RESERVADOS © PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis está protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

UNIDAD ACADEMICA DE LOS CICLOS PROFESIONAL Y DE POSGRADO
DEL COLEGIO DE CIENCIAS Y HUMANIDADES

INSTITUTO DE INVESTIGACIONES EN MATEMATICAS
APLICADAS Y EN SISTEMAS

IMPLANTACION DE UN LENGUAJE ORIENTADO A REGLAS PARA LA PROGRAMACION DE SISTEMAS EXPERTOS

TESIS QUE PARA OBTENER EL GRADO DE

MAESTRO EN CIENCIAS DE LA COMPUTACION

PRESENTA

NORMA ANGELICA ARANDA DE DIOS

MEXICO, 1991.

INDICE

CAPITULO 1

INTRODUCCION

1.1 Identificación del problema 1

1.2 Inteligencia artificial y bases de datos deductivas 1

1.3 Objetivo de la tesis 5

1.4 Trabajos Relacionados 8

1.5 Organización de la tesis 9

CAPITULO 2

LOGICA Y BASES DE DATOS

2.1 Bases de Datos Relacionales 11

2.1.1 El modelo relacional 11
2.1.2 Álgebra relacional 13
2.1.3 Lenguaje relacional de consulta (SQL:Structured Query Language) 14

2.2 Lógica matemática 16

2.2.1 Lógica Proposicional 16
2.2.2 Lógica de primer orden 19

2.3 Bases de datos deductivas 24

2.3.1 Bases de datos deductivas definidas 24
2.3.2 Bases de datos deductivas indefinidas 26

CAPITULO 3.

MECANISMO DE INFERENCIA

3.1 El principio de resolución para la lógica proposicional........ 27

3.2 Sustitución y Unificación........ 28

3.3 El Principio de Resolución para la Lógica de Primer Orden........ 31

3.4 El Proceso de Deducción........ 32

CAPITULO 4.

DESCRIPCION DEL SISTEMA.

4.1 Arquitectura General del sistema........ 36

4.2 Las Cláusulas de Horn y el álgebra relacional........ 38

4.3 Descripción del lenguaje........ 42

4.4 Descripción del algoritmo de unificación........ 45

CAPITULO 5.

IMPLANTACION DEL SISTEMA

5.1 Implantación del algoritmo de unificación........ 53

5.2 Representación interna de las reglas........ 58

5.3 Implantación del parser........ 66

5.4 Generación del árbol de inferencia........ 67

5.5 Implantación de la sustitución........ 68

5.6 Evaluación del árbol de inferencia........ 70

5.7 Interfase con el manejador de bases de datos........ 73

CAPITULO 6.

EVALUACION DE RESULTADOS

6.1 Comparación entre las bases de datos deductivas y las bases de datos tradicionales. 75

6.2 Estrategias de diseño e implantación.......... 76

6.3 Características del software empleado.......... 78

6.4 Ampliaciones al sistema.......... 78

6.5 Conclusiones.......... 79

APENDICE A. Descripción en BNF de la gramática del lenguaje.......... 81

APENDICE B. Palabras reservadas del sistema.......... 84

APENDICE C. Mensajes de error.......... 86

APENDICE D. Archivos del sistema.......... 92

APENDICE E. Estructuras de datos.......... 96

APENDICE F. Tabla de transiciones del reconocedor del lenguaje.......... 103

APENDICE G. El precompilador para C de ORACLE (PRO*C).......... 108

BIBLIOGRAFIA.......... 114

REFERENCIAS.......... 115

TABLA DE FIGURAS

Figura 1.1 Arquitectura de un sistema experto ideal.......................... 5

Figura 1.2 Arquitectura del Sistema General.. 7

Figura 3.1 Árbol de derivación..35

Figura 4.1 Arquitectura del sistema...37

Figura 4.2 Descripción parcial de la gramática del lenguaje.................43

Figura 4.3 Programa escrito en el lenguaje orientado a reglas..............44

Figura 5.1 Representación interna de un sistema de multiecuaciones.54

Figura 5.2 Representación interna de las multiecuaciones...................55

Figura 5.3 Representación interna de los multitérminos......................56

Figura 5.4 Función de unificación...57

Figura 5.5 Representación interna de las funciones.............................58

Figura 5.6 Representación interna de la tabla de variables.................59

Figura 5.7 Representación interna de las constantes........................... 61

Figura 5.8 Representación interna de las reglas.................................... 62

Figura 5.9 Representación interna de las relaciones........................... 63

Figura 5.10 Representación interna de las restricciones de integridad. 64

Figura 5.11 Representación interna de una regla..................................65

Capítulo 1.

INTRODUCCIÓN.

1.1 Identificación del problema.

La tendencia de la tecnología actual de bases de datos consiste en facilitar el desarrollo de aplicaciones más poderosas. La introducción de aplicaciones como CAD, CAE , CAM, automatización de oficinas, sistemas de entrenamiento, cartografía, etc., que manipulan grandes volúmenes de datos, ha motivado el diseño de sistemas de bases de datos más poderosas y flexibles. Tal es el caso de las bases de datos deductivas, las cuales cuentan con mecanismos de inferencia que permiten deducir conocimientos nuevos de hechos almacenados explícitamente.

1.2 Inteligencia Artificial y Bases de Datos.

La inteligencia artificial se encarga de los métodos que le permiten a la computadora resolver tareas para cuya solución se requiere inteligencia cuando ésta se lleva a cabo por un ser humano[JüGü88].

A pesar de que tradicionalmente ha existido poca interacción entre las investigaciones en las áreas de bases de datos e inteligencia artificial, recientemente se ha demostrado que las técnicas desarrolladas en cada área pueden ser aplicables a la otra. Una área de la inteligencia artificial que ha tenido considerables adelantos es el desarrollo de sistemas expertos o sistemas basados en el conocimiento. Los sistemas expertos son conjuntos de programas de computadora para la solución de problemas en áreas específicas en las cuales el conocimiento de un experto es necesario.

1. Introducción.

Desde el punto de vista de la inteligencia artificial, el diseño de un sistema experto se puede considerar como la construcción de una base de conocimiento para la representación del dominio de discurso. El principal problema de la representación del conocimiento es desarrollar una notación lo suficientemente precisa. A tal notación se le denomina un esquema de representación, por medio de la cual se puede especificar una base de conocimiento. Existen varios esquemas de representación del conocimiento [MyLe84]:

• **Redes Semánticas.**- Representan el conocimiento en términos de una colección de objetos (nodos) y asociaciones binarias. De acuerdo a este punto de vista una base de conocimiento es una colección de objetos e interrelaciones definidas sobre ellos, las modificaciones a la base de conocimientos se efectuan por la inserción o borrado de objetos y la manipulación de las interrelaciones.

• **Esquemas Procedurales.**- En estos esquemas se considera que una base de conocimiento está constituida por una colección de agentes activos o procesos. Una ventaja de estos esquemas de representación consiste en permitir la especificación de interacciones descritas entre hechos eliminando busquedas innecesarias, sin embargo, una base de conocimiento procedural es dificil de comprender y modificar.

• **Esquemas basados en "Frames".**- Un "frame" es una estructura de datos compleja para representar una situación estereotipada. Un "frame" consiste de ranuras ("slots") para los objetos que juegan un papel ("role") en la situación estereotipada, así como relaciones entre estos "slots".

• **Esquemas Lógicos.**- Una base de conocimiento desde este punto de vista consiste de una colección de fórmulas lógicas. Los esquemas lógicos disponen de reglas de inferencia, las cuales se pueden emplear para definir procedimientos de prueba. Por medio de tales procedimientos se puede extraer información, realizar la verificación de integridad semántica y solucionar problemas. La simplicidad en la notación empleada facilita la comprensión de las bases de conocimiento, además cuentan con una semántica formal.

No existe en general un esquema superior a otro para resolver todos los problemas, aunque los esquemas lógicos, como se mencionó, tienen grandes ventajas para la construcción de sistemas expertos.

En la construcción de sistemas expertos se trata con dos variantes del conocimiento: hechos y reglas de inferencia. Desde el punto de vista del conjunto de hechos y del conjunto de reglas podemos considerar cuatro tipos de sistemas expertos[JuAr90]:

1.- Con un conjunto reducido de hechos y un conjunto reducido de reglas.
2.- Con un conjunto grande de hechos y un conjunto reducido de reglas.
3.- Con un conjunto reducido de hechos y un conjunto grande de reglas.
4.- Con un conjunto grande de hechos y un conjunto grande de reglas.

debido a que las bases de datos son diseñadas para manipular un conjunto grande de datos, para los casos dos y cuatro es conveniente el uso de las bases de datos deductivas(1) para la construcción de sistemas expertos.

El conocimiento necesario para los sistemas expertos, se puede representar en un sistema de bases de datos como[Baye85]:

1.- Los hechos se expresan como reglas con el antecedente vacío y son almacenados en relaciones de la base de datos denominadas relaciones base.

2.- Las reglas de inferencia describen como se pueden deducir conocimientos nuevos de hechos almacenados explícitamente, y son representadas como expresiones de consulta de la base de datos.

1.3 Arquitectura de un sistema experto ideal.

La arquitectura ideal de un sistema experto se muestra en la figura 1.1. Los principales componentes de la arquitectura ideal son: el procesador del lenguaje, la base de conocimiento, el módulo explicativo, el pizarrón, el intérprete, el despachador ("scheduler") y el módulo para el mantenimiento de la consistencia semántica [HaWa83].

(1) Una base de datos deductiva permite deducir conocimientos de hechos almacenados explícitamente en la base de datos.

El **procesador del lenguaje** realiza el intercambio de información entre el usuario y el sistema. Típicamente el procesador del lenguaje efectúa tanto el análisis léxico y el análisis sintáctico, como la interpretación de las consultas y los comandos. Por otro lado, el procesador del lenguaje genera una representación de la información generada por el sistema, para responder a las consultas, explicar resultados y solicitar datos al usuario.

En el **pizarrón** se almacenan las hipótesis intermedias y las decisiones que el sistema manipula. Cada sistema experto emplea algún tipo de representación para las decisiones intermedias. En la arquitectura ideal se cuenta con tres clases de decisiones: el plan, la agenda y los elementos de solución. Los elementos del plan describen las estrategias generales que el sistema empleará para resolver el problema. La agenda almacena las acciones potenciales a ser ejecutadas, generalmente corresponden a las reglas de la base de conocimiento que parecen relevantes a alguna decisión. Los elementos solución representan las hipótesis candidatas y las decisiones que el sistema ha generado hasta este momento junto con las dependencias que relacionan unas decisiones con otras.

El **despachador** mantiene el control de la agenda y determina cuales acciones deben ejecutarse. El despachador determina las acciones más adecuadas a ejecutarse evitando esfuerzos redundantes, para lo cual necesita asignar a cada elemento de la agenda una prioridad de acuerdo a su relación con el plan y los elementos de solución.

El **intérprete** ejecuta los elementos escogidos de la agenda, aplicando la regla correspondiente de la base de conocimiento, para lo cual valida las condiciones relevantes de la regla, liga variables de estas condiciones a los elementos de solución particulares del pizarrón y actualiza el pizarrón.

El **módulo explicativo** justifica las acciones del sistema al usuario. Describe al usuario como el sistema logró obtener las conclusiones. Para realizar lo anterior, el módulo explicativo requiere conocer las hipótesis intermedias empleadas para llegar a la conclusión.

El **módulo de mantenimiento de la consistencia semántica** se encarga de mantener una representación íntegra de los datos. Algunos sistemas expertos emplean un esquema de ajuste numérico para determinar el grado de certeza en cada decisión, estos esquemas procuran asegurar que se alcancen conclusiones plausibles y se eviten las inconsistentes.

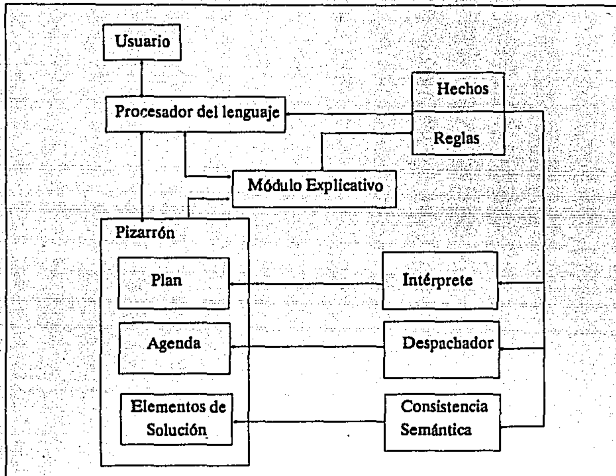


Figura 1.1 Arquitectura de un sistema experto ideal.

1.4. Objetivo de la tesis.

El sistema programado para esta tesis forma parte de una base de datos deductiva para la programación de sistemas expertos. La arquitectura del sistema general se muestra en la figura 1.2, y sus principales componentes son: la interfaz con el usuario, el lenguaje orientado a reglas, el parser, el monitor de transacciones, el subsistema de inferencia, la base de reglas, las restricciones de integridad semántica, el catálogo, el manejador de bases de datos relacional (RDBMS) y la base de datos[JuAr90].

El **lenguaje orientado a reglas** permite la generación de una base de datos convencional o la generación de un sistema deductivo. El lenguaje consiste de cuatro elementos: funciones, relaciones, reglas deductivas y restricciones de integridad.

El **parser** se encarga de efectuar el análisis léxico y el análisis sintáctico del programa, transformándolo a una representación interna.

El núcleo del sistema lo constituye el **subsistema de inferencia**, el cual es una extensión de un manejador de bases de datos relacional para soportar la deducción. Los componentes del subsistema de inferencia son:

a) **El Intérprete.** Es el componente más importante del subsistema de inferencia y se encarga de deducir conocimientos a partir de hechos de la base de datos y de las reglas almacenadas en la base de reglas deductivas.

b) **El Módulo Explicativo.** Es el encargado de justificar como se logró obtener las conclusiones, para lo cual identifica las reglas deductivas empleadas.

c) **Mecanismo de Actualización.** Se ha incluido un mecanismo que permita la modificación de las relaciones base, de las reglas almacenadas en la base de reglas deductivas y de las restricciones de integridad almacenadas en la base de restricciones de integridad.

d) **Subsistema de optimización.** Lleva a cabo transformaciones en las reglas de inferencia utilizadas para responder consultas de usuario, de manera que la respuesta sea lo más rapida posible.

e) **El Pizarrón.** Es el área del sistema empleada para almacenar resultados intermedios. Estos resultados intermedios son empleados por el módulo explicativo para justificar las conclusiones.

El **monitor** transfiere el control al submódulo adecuado dependiendo de la operación que el sistema realiza. El catálogo almacena información acerca de los objetos del sistema, es decir, funciones, relaciones, reglas y restricciones de integridad.

El objetivo de esta tesis consiste en obtener una versión limitada de una base de datos deductiva, lo cual se concentra fundamentalmente en el desarrollo del parser, del lenguaje orientado a reglas y del intérprete. En el intérprete se considera la manipulación de reglas normalizadas no recursivas.

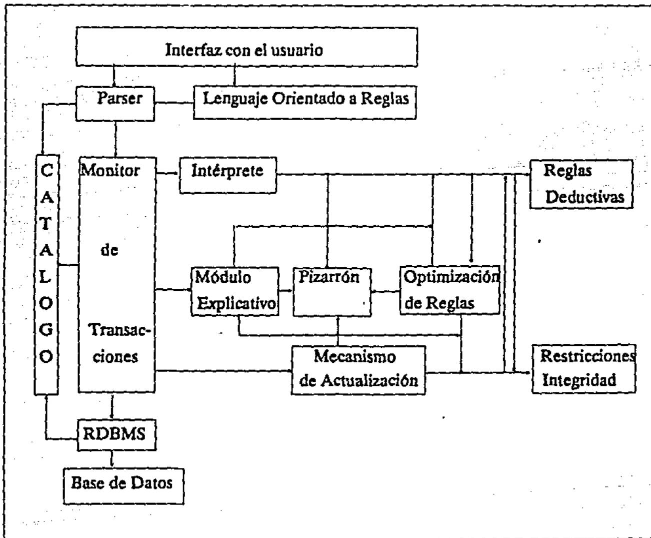


Figura 1.2 Arquitectura del sistema general.

1.5 Trabajos relacionados.

Un trabajo relacionado con esta tesis es el efectuado en la Universidad Tecnológica de Munich denominado "LOLA- A Logic Language for Deductive Database and its Implementation" (Un lenguaje lógico para Bases de Datos Deductivas y su Implantación) [FrSc90].

El lenguaje lógico LOLA se diseñó como un lenguaje de consulta para un sistema de bases de datos deductivas. LOLA integra programación lógica y el procesamiento relacional de consultas. La evaluación de consultas en lugar de efectuarla de arriba hacia abajo, la efectua de abajo hacia arriba comenzando en las relaciones base.

Una consulta se transforma a una expresión equivalente de un álgebra relacional extendida. Dado un conjunto de relaciones base, la expresión resultante calcula el conjunto de tuplos correspondientes que responden a la consulta. Las relaciones base pueden estar en memoria principal o en cualquier sistema de bases de datos relacional externo accesible por medio de SQL.

Los componentes del sistema LOLA son una interfaz con el usuario, el compilador, el optimizador y un sistema para ejecución ("run-time"):

• La interfaz con el usuario permite que el sistema acepte programas y consultas, después que la consulta es compilada por medio de esta interfaz se muestra la respuesta al usuario o se almacena como una relación.

• El compilador toma como entrada un programa o una consulta, y genera una expresión relacional equivalente.

• El módulo de optimización puede realizar dos tipos de optimizaciones: a nivel fuente o actuando sobre el código intermedio representado por una gráfica de operadores.

• Los componentes del módulo de ejecución son: LISP como lenguaje anfitrión, el álgebra relacional extendida denominada R-LISP, una base de datos en memoria principal y una interfaz con SQL para sistemas manejadores de bases de datos relacionales externos.

Otro trabajo relacionado con esta tesis es el denominado "DATALOG"[Ullm88]. Datalog difiere de PROLOG en:

1. No permite símbolos de función como argumentos, es decir, Datalog permite sólo variables y símbolos de constantes como argumentos de un predicado.

2. La interpretación de los programas de Datalog utilizan el punto de vista del modelo teórico, mientras que Prolog cuenta con una interpretación computacional.

El modelo de datos matemático que fundamenta Datalog es esencialmente el modelo relacional, aunque las relaciones no hacen referencias a sus argumentos por su nombre, sino por su posición. Un predicado se define sólo de manera extensional o de manera intensional, pero no ambas.

Los sistemas descritos se orientan sólo al desarrollo de bases de datos deductivas de propósito general, mientras que nuestro sistema pretende desarrollar una herramienta para la programación de sistemas expertos, la cual utilize una base de datos deductiva. En este trabajo, como ya se mencionó anteriormente, nos limitamos al desarrollo e implantación parcial de la base de datos deductiva.

1.6. Organización de la tesis.

Dado que las bases de datos deductivas son una extensión de las bases de datos tradicionales, en el capítulo dos se describen tanto las bases de datos relacionales como la lógica de primer orden.

El proceso deductivo se fundamenta en el principio de Resolución, el cual es un procedimiento de prueba por refutación. En el capítulo tercero se describe el principio de Resolución para la lógica proposicional y para la lógica de primer orden, los conceptos de sustitución y unificación, así como diversos métodos para realizar la deducción en las bases de datos deductivas.

En el capítulo cuarto se efectua la descripción de la aquitectura del sistema objeto de esta tesis, así como el lenguaje orientado a reglas y el algoritmo de unificación empleado.

En el capítulo cinco se describe la implantación de cada uno de los principales componentes del sistema: las estructuras de datos empleadas para representar el conocimiento, la implantación del algoritmo de unificación, el parser, el proceso de generación del árbol de inferencia y el proceso de evaluación del árbol de inferencia.

Para la evaluación de resultados descrita en el capitulo sexto, se efectúa primero una comparación entre las bases de datos deductivas y las bases de datos tradicionales. Las estrategias de diseño e implantación emplean algunos principios básicos de programación. El software seleccionado para implantar el sistema es el lenguaje C y el manejador de bases de datos relacional ORACLE, sobre el cual se montó el sistema.

Capítulo 2.

LÓGICA

Y

BASES DE DATOS.

2.1 Bases de Datos Relacionales.

2.1.1 El Modelo Relacional.

Una aplicación está constituida por una colección de objetos interrelacionados y actividades que los involucran. Desde el punto de vista del modelado de datos se puede caracterizar una aplicación por:

i) Objetos relevantes y sus propiedades.
ii) Operaciones sobre los objetos y sobre las interrelaciones.
iii) Restricciones de integridad semántica.

Un modelo de datos es una colección de conceptos que permiten representar tanto los objetos como las interrelaciones de los objetos de una aplicación. A la especificación de los objetos y sus interrelaciones de una aplicación se le denomina un esquema.

En el modelo relacional se representa de la misma forma a los objetos y a las interrelaciones. A esta propiedad se le denomina relativismo semántico, el cual se expresa denominando entidades tanto a los objetos como a las interrelaciones; dichas entidades se representan por medio de n-adas, donde un conjunto de entidades del mismo tipo se representa por medio de un conjunto de n-adas; tal conjunto de n-adas es en un sentido formal una relación.

Definición 2.1. R es una relación definida sobre los dominios $D_1,...,D_n$ si y solo si

$$R \subseteq D_1 \times ... \times D_n$$

donde $D_1 \times ... \times D_n$ es el producto cartesiano de los dominios.

A una colección de relaciones se le denomina una base de datos relacional y la descripción de las estructuras de las relaciones de una base de datos relacional se especifican por medio de un esquema relacional. Las relaciones de un esquema relacional se especifican por medio de atributos, los cuales toman valores de un conjunto denominado dominio.

Asociado al modelado de datos se dispone de lenguajes para la definición y manipulación de los objetos de una aplicación. Una base de datos cuenta en general con cuatro lenguajes.

1. **Lenguaje de definición de datos (DDL).** Permite la definición o descripción de los objetos de la base de datos, es decir, se emplea para describir el esquema conceptual de la base de datos.

2. **Lenguaje de manipulación de datos (DML).** Lleva a cabo las acciones sobre los objetos, las cuales pueden ser: insertar, borrar o modificar.

3. **Lenguaje de Consulta.** Permite el acceso a la base de datos por parte de los usuarios.

4. **Lenguaje de Control.** Corresponde al lenguaje empleado por el administrador de la base de datos para el control del sistema de bases de datos, así como permite a los usuarios definir privilegios sobre sus relaciones. Por ejemplo está la creación de cuentas a usuarios, asignación de privilegios dependiendo del tipo de usuario, preservar la seguridad de la base de datos, etc.

Nosotros estamos interesados fundamentalmente en el lenguaje de consulta. En el modelo relacional los lenguajes de consulta se pueden clasificar en dos tipos: los lenguajes algebraicos y los lenguajes basados en el cálculo relacional. Dado que en este trabajo se utiliza el álgebra relacional, omitiremos la descripción del cálculo relacional.

2.1.2 Álgebra Relacional.

El álgebra relacional se compone de dos grupos de operadores: los operadores tradicionales sobre conjuntos: unión, intersección, diferencia y producto cartesiano, y los operadores especiales: restricción, proyección, "join" y división. Los operadores tradicionales sobre conjuntos se definen de la manera usual, sólo que para el caso de la unión, intersección y diferencia, los operandos deben ser compatibles a la unión [1].

Definición 2.2. Sean R y S dos relaciones de orden k_1 y k_2 respectivamente, el producto cartesiano R X S es el conjunto de n-adas (r,s) de orden $k_1 + k_2$, donde r pertenece a R y s pertenece a S, es decir:

$$R \text{ X } S = \{ (r,s) \mid r \in R \text{ y } s \in S \}$$

Definición 2.3. La unión de dos relaciones compatibles a la unión R y S es el conjunto de n-adas que pertenecen a R, a S o a ambas, esto es:

$$R \cup S = \{ x \mid x \in R \text{ o } x \in S \}$$

Definición 2.4. La intersección de dos relaciones compatibles a la unión R y S es el conjunto de n-adas que pertenecen a R y a S, es decir:

$$R \cap S = \{ x \mid x \in R \text{ y } x \in S \}$$

Definición 2.5. La diferencia de dos relaciones compatibles a la unión R y S (en ese orden) es el conjunto de n-adas que pertenecen a R y no pertenecen a S, es decir:

$$R - S = \{ x \mid x \in R \text{ y } x \notin S \}$$

Definición 2.6. Sea $R(A_1, A_2, \dots, A_n)$ una relación de orden n, $X \subseteq \{A_1, A_2, \dots, A_n\}$ y $Y = \{A_1, A_2, \dots, A_n\} \setminus X$, permutando los atributos R puede representarse como R(X,Y), la proyección de R sobre los atributos X representada como R[X] está definida por :

$$R[X] = \{ x \mid \text{existe una y tal que } (x,y) \in R(X,Y) \}$$

(1) Se dice que dos relaciones son compatibles a la unión si el número de atributos y los dominios de los atributos correspondientes son los mismos.

Definición 2.7. Sea $R(A_1,A_2,...,A_n)$ una relación de orden n y P una condición lógica definida sobre el producto cartesiano de los dominios de los atributos, la restricción (selección) de la relación R con respecto a P (R[P]) se define como :

$$R[P] = \{ x \mid x \text{ está en R y } P(x) \text{ es verdadera} \}$$

Definición 2.8. Sean $R(A_1,...,A_n)$ y $S(B_1,...,B_n)$ dos relaciones y sean los conjuntos de atributos $X \subseteq \{ A_1,...,A_n\}$ y $Y \subseteq \{B_1,...,B_m\}$, los cuales tienen el mismo número de atributos y los dominios de los atributos correspondientes son los mismos. Si denotamos $Z = \{A_1,...,A_n\} \setminus X$ y $W = \{B_1,...,B_m\} \setminus Y$, permutando el orden de los atributos, podemos representar las relaciones R y S como R(Z,X) y S(Y,W). El "join" natural de R y S sobre los atributos x,y, denotado por R[X=Y]S, se define como:

$$R[X=Y]S = \{ (z,x,w) \mid (z,x) \in R, (y,w) \in S \text{ y } x = y \}$$

Definición 2.9. Sean R(X,Y) y S(Z) dos relaciones donde X,Y y Z son conjuntos de atributos. Asumiendo que Y y Z contienen el mismo número de atributos y los dominios de los atributos correspondientes son iguales; el resultado de la división de R(X,Y) entre S(Z), expresado como R[Y:Z]S, es el subconjunto máximo de la proyección R[X] tal que su producto cartesiano con S(Z) está contenido en R(X,Y) , es decir:

$$R(X,Y) = R[Y:Z]S \times S(Z) \cup Q(X,Y)$$

donde R[Y:Z]S es el cociente y Q(X,Y) es el residuo.

En términos matemáticos, el álgebra relacional constituye un sistema cerrado, dado que define operaciones sobre relaciones y como resultado genera relaciones. Debido a que el resultado de una operación del álgebra relacional es una relación, ésta puede a su vez someterse a operaciones algebraicas adicionales.

2.1.3. Lenguaje Relacional de Consulta(SQL: "Structured Query Language").

Aunque el álgebra relacional es un lenguaje formal simple, éste no es muy adecuado para usuarios casuales de la base de datos, por lo que se han desarrollado algunos lenguajes más amigables como SQL, el cual empleamos debido a que es aceptado como un estandar.

El concepto fundamental de SQL es el denominado bloque de consulta cuya forma básica es:

SELECT < lista de atributos >

FROM < lista de relaciones >

WHERE < condición >

El resultado de la ejecución de un bloque de consultas es una relación cuya estructura y contenido está determinada por este bloque. Los atributos de esta relación están especificados en la lista de atributos. Los atributos listados son seleccionados de las relaciones en la lista de relaciones de la cláusula **FROM** . Las primeras dos cláusulas (**SELECT** y **FROM**) definen la operación de proyección. La condición de la cláusula **WHERE** es una expresión lógica; ésta contiene atributos de las relaciones listadas en la cláusula **FROM** y determina que n-adas de las relaciones serán objeto de la proyección. La cláusula **WHERE** contiene las especificaciones de las operaciones de restricción y "join".

Como se mencionó anteriormente el álgebra relacional constituye un sistema cerrado, de aquí que los bloques de consulta pueden aparecer como operandos del conjunto de operaciones : unión, intersección y diferencia. Para expresar este conjunto de operaciones, SQL cuenta con tres cláusulas adicionales : **UNION, INTERSECT** y **MINUS** respectivamente.

En las bases de datos existen dos tipos de relaciones: relaciones base y relaciones virtuales. Las relaciones base existen explícitamente en la base de datos, mientras que las segundas se obtienen a partir de las relaciones base y se conocen como vistas. SQL es más que un lenguaje de consulta debido a que integra instrucciones de los cuatro lenguajes de las bases de datos. Entre las instrucciones del lenguaje de definición de datos están la creación o borrado de relaciones y vistas.. La creación de una relación base se efectua por medio de la instrucción:

CREATE TABLE < nombre de la relación >

(<atributo $_1$> <Tipo $_1$> [NOT NULL],

<atributo $_2$> <Tipo $_2$> [NOT NULL],

:

<atributo $_n$> <Tipo $_n$> [NOT NULL])

donde [NOT NULL] indica que el atributo no puede asumir un valor nulo.

La creación de una vista se efectúa con la cláusula CREATE VIEW, cuya sintaxis es:

CREATE VIEW <nombre vista> <lista de atributos > AS <consulta>

Además de contar con instrucciones para la definición de datos, SQL también incorpora instrucciones del lenguaje de manipulación de datos (DML), entre las más relevantes cabe mencionar la inserción, el borrado y la actualización cuyas sintaxis están dadas respectivamente por:

INSERT INTO <nombre relación> [(<atributo$_1$,..,atributo $_n$>)]
{VALUES(<valor$_1$>. . .<valor $_n$>) | <consulta> }

DELETE FROM <nombre relación>
[WHERE <condición>]

UPDATE <nombre relación>
SET { <nombre atributo> = <valor $_1$>...<valor $_n$> } |
(atributo $_1$,. . . ., atributo $_n$) = (subquery)
[WHERE <condición>]

2.2 Lógica Matemática.

2.2.1 Lógica Proposicional.

Se dice que la lógica es el estudio del razonamiento, expresado por medio de cadenas de entidades linguísticas. Las entidades linguísticas que aparecen en esta clase de razonamiento forman sentencias, es decir, entidades que expresan un pensamiento completo [Dale83]. Una sentencia que puede ser exclusivamente verdadera o falsa se conoce como una proposición. La lógica que trata con proposiciones se denomina lógica proposicional. En la lógica proposicional existen proposiciones mínimas (las cuales no pueden descomponerse) denominadas **átomos** o **fórmulas atómicas**. A partir de los átomos se pueden construir proposiciones compuestas empleando conectivos lógicos. Los conectivos lógicos básicos de la lógica proposicional son: conjunción ($\wedge$), disyunción ($\vee$), negación ($\neg$), implicación ($\rightarrow$) y equivalencia ($\leftrightarrow$).

Definición 2.11 El lenguaje de la lógica proposicional cuenta con un alfabeto formado por:

i) Símbolos de proposiciones: $P_1, P_2, \ldots, P_n$
ii) Conectivos lógicos: $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$
iii) Símbolos auxiliares: "(" , ")" .

En esta lógica las expresiones que representan proposiciones se denominan **fórmulas** o **fórmulas bien formadas.**

Definición 2.12. Fórmula bien formada:

i) Un átomo es una fórmula.
ii) Si φ es una fórmula, entonces $\neg\varphi$ es una fórmula.
iii) Si φ y ψ son fórmulas, entonces $\varphi * \psi$ es una fórmula, para $* \in \{\wedge, \vee, \neg\}$.
iv) Ninguna otra cosa es una fórmula.

En la lógica proposicional estamos interesados en conocer cuando una fórmula representa un razonamiento verdadero o falso. Al valor verdadero o falso que puede tomar una fórmula se le denomina su valor de verdad.

Definición 2.13 Sea ν una función que mapea una fórmula φ al conjunto de valores de verdad $\{0,1\}$, donde 1 significa verdadero y 0 falso, se dice que ν es una evaluación si [Dale83]:

$$\nu(\varphi \wedge \psi) = \min(\nu(\varphi), \nu(\psi))$$

$$\nu(\varphi \vee \psi) = \max(\nu(\varphi), \nu(\psi))$$

$$\nu(\varphi \rightarrow \psi) = 0 \leftrightarrow \nu(\varphi) = 1 \text{ y } \nu(\psi) = 0$$

$$\nu(\varphi \leftrightarrow \psi) = 1 \leftrightarrow \nu(\varphi) = \nu(\psi)$$

$$\nu(\neg\varphi) = 1 - \nu(\varphi)$$

Definición 2.14. Sea φ una fórmula y sean $P_1, \ldots, P_n$ átomos que aparecen en φ, una interpretación de φ es una asignación de valores de verdad a $P_1, \ldots, P_n$, en la cual a cada P_i se le asigna 0 o 1, pero no ambos.

Para una fórmula que contiene n átomos diferentes, se pueden tener 2^n asignaciones distintas de valores de verdad. Cada una de estas asignaciones se denomina una interpretación de la fórmula.

Definición 2.15. Se dice que una fórmula φ es verdadera bajo una interpretación si y solo si ella evalua a 1 en esa interpretación; en caso contrario se dice que φ es falsa bajo esa intepretación.

A una fórmula que es verdadera bajo todas las interpretaciones posibles se le denomina fórmula válida o tautología. Cuando una fórmula es falsa bajo todas las interpretaciones se denomina inconsistente o se dice que es una contradicción.

Si una fórmula φ es verdadera bajo una interpretación I, se dice que I satisface φ, en caso contrario I falsifica φ. Cuando una interpretación satisface una fórmula, se dice que la interpretación es un modelo de la fórmula.

Definición 2.16. Una literal es un átomo o la negación de un átomo.

En la aplicación de la lógica a la computación suele ser conveniente el transformar una fórmula de una forma dada a otra forma equivalente.

Definición 2.17. Se dice que dos fórmulas φ y ψ son equivalentes si y solo si los valores de verdad de φ y ψ son los mismos bajo cualquier interpretación de φ y ψ.

En los procesos deductivos es conveniente trabajar con formas normales. En la lógica proposicional se cuenta con dos formas normales : forma normal conjuntiva y forma normal disyuntiva.

Definición 2.18. Se dice que una fórmula φ está en forma normal conjuntiva si y solo si tiene la forma $\varphi_1 \wedge \ldots \wedge \varphi_n$, $n \geq 1$, donde cada una de las φ_i ($i = 1, \ldots, n$) es una disyunción de literales[1].

(1) Una disyunción de literales es una fórmula de la forma $L_1 \vee \ldots \vee L_n$.

Definición 2.19. Se dice que una fórmula φ está en forma normal disyuntiva si y solo si φ tiene la forma $\varphi_1 \vee \ldots \vee \varphi_n$, $n \geq 1$, donde cada $\varphi_i (i = 1,...,n)$, es una conjunción de literales [1] .

Desde el punto de vista deductivo nos interesa probar que cierta fórmula es consecuencia lógica de un conjunto de fórmulas.

Definición 2.20. Dadas las fórmulas $\varphi_1,...,\varphi_n$ y ψ, se dice que ψ es una consecuencia lógica de $\varphi_1,...,\varphi_n$, si y solo si para cualquier interpretación en la cual $\varphi_1 \wedge \ldots \wedge \varphi_n$ es verdadera, ψ también es verdadera. A $\varphi_1,...,\varphi_n$ se les denomina axiomas, premisas o postulados de ψ.

2.2.2 Lógica de primer orden.

La lógica proposicional es un modelo de deducción sencillo, de aqui que existan ideas que no se pueden expresar en ella, por lo cual se introduce la lógica de primer orden, la cual contiene algunas nociones lógicas adicionales como son: términos, predicados y cuantificadores.

Los símbolos del alfabeto de la lógica de primer orden que se pueden emplear para construir átomos son:

i) Símbolos de constantes
ii) Símbolos de variable
iii) Símbolos de función
iv) Símbolos de predicado.

Donde los símbolos de función y predicado pueden tener un número específico de argumentos. Si un símbolo de función (o predicado) tiene n argumentos se dice que es una función (o predicado) de orden n. Los símbolos de constantes se consideran como símbolos de función de orden 0.

El lenguaje de la lógica de primer orden trata básicamente con dos tipos de entidades sintácticas: los términos y las fórmulas.

---

(1) Una Conjunción de literales es una fórmula de la forma $L_1 \wedge \ldots \wedge L_n$.

Definición 2.21. Término:

i) Una constante es un término.
ii) Una variable es un término.
iii) Si f es un símbolo de función de orden n y $t_1,...t_n$ son términos, entonces $f(t_1,...,t_n)$ es un término.
iv) Ninguna otra cosa es un término.

Definición 2.22. Si P es un símbolo de predicado de orden n, y $t_1,..,t_n$ son términos, entonces $P(t_1,...,t_n)$ es un átomo.

Para construir fórmulas en la lógica de primer orden, además de contar con los conectivos lógicos definidos en la lógica proposicional, se introducen los cuantificadores universal ($\forall$) y existencial ($\exists$). Dependiendo del alcanze que los cuantificadores pueden tener sobre las variables en una fórmula, se definen dos tipos de variables: variables libres y variables ligadas. Una ocurrencia de una variable en una fórmula es ligada, si y solo si la ocurrencia está bajo el alcance de algún cuantificador que la incluye; en caso contrario se dice que la variable es libre. Por ejemplo, en la fórmula $(\forall z)P(z,x) \wedge (\forall x)\ R(x)$, la variable z es ligada, mientras que x es tanto libre como ligada debido a que la primera ocurrencia no está afectada por ningún cuantificador, mientras que en la segunda parte de la fórmula x se encuentra afectada por el cuantificador universal.

Definición 2.23. Fórmula bien formada:

i) Un átomo es una fórmula.
ii) Si φ es una fórmula, entonces $\neg\varphi$ es una fórmula.
iii) Si φ y ψ son fórmulas, entonces $\varphi * \psi$ es una fórmula, donde $* \in \{ \wedge, \vee, \neg \}$.
iv) Si φ es una fórmula y x es una variable libre en φ, entonces $(\forall x)\varphi$ y $(\exists x)\varphi$ son fórmulas.
v) Ninguna otra cosa es una fórmula.

Debido a que en la lógica de primer orden se incluyen variables, para definir una interpretación de una fórmula es necesario especificar el dominio y una asignación para los símbolos de constantes, funciones y predicados que aparecen en la fórmula.

Definición 2.24. Una interpretación de una fórmula φ, consiste de un dominio no vacío D, y de una asignación de valores a los símbolos de constantes, funciones o predicados que aparecen en φ, de la siguiente manera:

i) A cada símbolo de constante c_i, se le asocia un elemento d_i en D.
ii) A cada símbolo de función de orden n, se le asocia un mapeo de D^n en D.
iii) A cada símbolo de predicado de orden n, se le asocia un mapeo de D^n al conjunto de valores de verdad $\{0,1\}$.

Definición 2.25. La evaluación de una fórmula es:

i) Sean φ y ψ fórmulas, los valores de verdad para las fórmulas $\neg\varphi$, $\varphi * \psi$, donde $* \in \{\wedge, \vee, \neg\}$ se evalúan como en la lógica proposicional.

ii) $(\forall x)\,\varphi$ es verdadera si φ evalúa a verdadera para cualquier d en D; en caso contrario φ es falsa.

iii) $(\exists x)\varphi$ es verdadera si φ es verdadera para al menos una d en D; en caso contrario φ es falsa.

Definición 2.26. Una fórmula φ es consistente si y solo si existe una interpretación I tal que φ es verdadera en I. Si una fórmula φ es verdadera en una interpetación I , se dice que la interpretación es un modelo de la fórmula.

Definición 2.27. Una fórmula ψ es una consecuencia lógica de las fórmulas $\varphi_1,\ldots,\varphi_n$ si y solo si para cualquier interpretación I en la cual $\varphi_1,\ldots,\varphi_n$ son verdaderas, ψ también es verdadera.

De manera similar como en la lógica proposicional se definieron dos formas normales, en la lógica de primer orden se introducen tres formas normales: forma normal prenex, forma normal de Skolem y forma clausal.

Definición 2.28. Se dice que una fórmula φ está en forma normal prenex si y solo si φ es de la forma:

$$(Q_1 x_1) \ldots (Q_n x_n) (M)$$

donde cada Q_i para $i = 1, \ldots, n$, es un cuantificador universal o un cuantificador existencial, cada x_i ($i = 1, \ldots, n$) es la variable cuantificada y M es una fórmula sin cuantificadores. $(Q_1 x_1) \ldots (Q_n x_n)$ se denomina el prefijo de φ y M se denomina la matriz de φ.

Una fórmula en forma normal prenex se puede transformar a una fórmula en forma normal de Skolem, para lo cual se eliminan todos los cuantificadores existenciales que aparezcan en la fórmula. Cada variable cuantificada existencialmente será reemplazada por funciones denominadas funciones de Skolem. Las funciones de Skolem son funciones arbitrarias cuyos argumentos son todas las variables cuantificadas universalmente que preceden la variable a ser reemplazada. Cuando una variable cuantificada existencialmente no tiene variables cuantificadas universalmente que la precedan se sustituye por una constante de Skolem.

Definición 2.29. Una fórmula en forma normal de Skolem es aquella donde todas las variables están cuantificadas universalmente.

Ejemplo: Dada la siguiente fórmula en forma normal prenex

$$\exists x \, \forall y \, \forall z \, \exists u \, \forall v \, \exists w \; P(x,y,z,u,v,w)$$

para transformarla a forma normal de Skolem es necesario eliminar los cuantificadores existenciales que afectan las variables x, u y w, para lo cual x se sustituye por una constante de Skolem arbitraria "a" debido a que no la precede ninguna variable cuantificada universalmente, u se sustituye por una función de Skolem arbitraria f cuyos argumentos son y y z, de igual manera w se sustituye por una función de Skolem arbitraria g con argumentos y,z y v, obteniendo la siguiente expresión para la forma normal de Skolem:

$$\forall y \, \forall z \, \forall v \; P(a, y, z, f(y,z), v, g(y,z,v))$$

Definición 2.30. Una cláusula es una disyunción de literales.

Debido a que el empleo de las cláusulas es muy común en la programación lógica, es conveniente adoptar una notación especial:

$$A_1 \vee \ldots \vee A_k \vee \neg B_1 \vee \ldots \vee \neg B_n$$

la cual se transforma a la siguiente expresión equivalente:

$$A_1 \vee \ldots \vee A_k \leftarrow B_1 \wedge \ldots \wedge B_n$$

donde el lado izquierdo se denomina consecuente y el lado derecho antecedente. Cuando $k = 1$ o $k = 0$, la cláusula se denomina **Cláusula de Horn.**

Definición 2.31. Una cláusula programa es una cláusula de la forma $A \leftarrow B_1, \ldots, B_n$, $n \geq 0$, la cual contiene una literal positiva A denominada cabeza y $B_1, \ldots, B_n$ se denominan el cuerpo de la cláusula programa.

Definición 2.32. Una cláusula unitaria es una cláusula de la forma $A \leftarrow$, es decir, una cláusula programa donde el cuerpo es vacío.

Definición 2.33. Un programa lógico es una secuencia finita de cláusulas programa.

En un programa lógico, al conjunto de todas las cláusulas programa con el mismo predicado P en la cabeza se denomina la definición de P.

Definición 2.34. Una cláusula meta es una cláusula de la forma $\leftarrow B_1, \ldots, B_n$, $n > 0$, es decir, una cláusula cuyo consecuente es vacío. Cada B_i $(i = 1, \ldots, n)$ se denomina una submeta de la cláusula meta.

La cláusula vacía es la cláusula con consecuente y antecedente vacío, esta cláusula representa una contradicción.

Definición 2.35. Se dice que una cláusula es de rango restringido si todas las variables que aparecen en el consecuente también aparecen en el antecedente de la cláusula.

Dado que la programación lógica con Cláusulas definidas no ofrece suficiente poder expresivo para el desarrollo de sistemas expertos, es necesario introducir la operación de negación[ApBl88]. El trabajo desarrollado utiliza Cláusulas definidas con negación.

2.3 Bases de Datos Deductivas.

Una base de datos deductiva (bdd) es una base de datos en la cual se pueden derivar nuevos hechos de hechos almacenados explícitamente en la base de datos. Una bdd consiste de hechos explícitos y de axiomas generales, es decir, de un conjunto de constantes $\{c_1,...,c_n\}$ y un conjunto finito de cláusulas de primer orden que no contienen símbolos de funciones. Las funciones se excluyen para obtener respuestas finitas y explícitas a las consultas. Las bdd se dividen en dos clases: bases de datos deductivas definidas y bases de datos deductivas indefinidas [GaMn84].

2.3.1. Bases de Datos Deductivas Definidas.

Una base de datos deductiva definida se define como una teoría particular de primer orden. Esta teoría se obtiene de la teoría dada por las bases de datos convencionales más una nueva clase de axiomas para las leyes deductivas. Formalmente una base de datos deductiva definida consiste en:

1. Una teoría cuyos axiomas propios son:

a) Axiomas de particularización.

- Axioma de cerradura del dominio.

  Establece que no existen más elementos que aquellos que están en la base de datos.

- Axioma de nombre único.

  Indica que elementos con nombres distintos son diferentes.

- Axiomas de igualdad.

  Especifican las propiedades de la igualdad:

  reflexividad: $\forall x\ (x = x)$

  simetría : $\forall x\ \forall y\ ((x = y) \rightarrow (y = x))$

  transitividad : $\forall x\ \forall y\ \forall z\ ((x = y) \wedge (y = z) \rightarrow (x = z))$.

•Axioma de completéz.
Se conforma de todos los hechos y de las leyes deductivas que involucran a dicho predicado.

b) Hechos elementales .
Conjunto de fórmulas definidas por medio de cláusulas de la forma:

$$R(c_1, \ldots, c_n) \leftarrow .$$

c) Leyes deductivas.
Corresponden al conjunto de cláusulas programas libres de funciones:

$$Q_1 \leftarrow P_1, \ldots, P_k.$$

2. Un conjunto de restricciones de integridad semántica. Permiten representar las propiedades estáticas que los objetos deben cumplir para poder pertenecer a la base de datos.

Las relaciones definidas por la unión de las leyes deductivas y los hechos elementales en una base de datos deductiva se denominan **relaciones derivadas.** Estas relaciones constituyen una generalización de las relaciones definidas como vistas en las bases de datos convencionales. Una relación derivada corresponde a una vista cuando no existen hechos elementales asociados con ésta, y no aparecen leyes deductivas recursivas entre las leyes deductivas que implican esta relación.

Debido a la complejidad combinatoria de los axiomas de particularización, la implantación de un Sistema Manejador de Bases de Datos Deductivo sería ineficiente, lo cual se soluciona eliminando los axiomas de particularización de la siguiente manera:

i) El axioma de cerradura del dominio se puede evitar si se trabaja con cláusulas de rango restringido.

ii) Los axiomas de nombre único y completéz se excluyen si se establece que la negación se interprete como falla [GaMn84].

Considerando fórmulas de rango restringido desde un punto de vista operacional una base de datos deductiva definida consiste en:

1. Un conjunto de axiomas formado por hechos elementales y leyes deductivas.
2. Un conjunto de restricciones de integridad.
3. Una metaregla: La negación como falla.

Comparando ambas definiciones tenemos que las dos son equivalentes en el sentido que proporcionan la misma respuesta a las consultas, pero no son equivalentes desde el punto de vista lógico, porque la primera se establece en una lógica monótona [(1)], mientras que la definición operacional conduce a una lógica no monótona.

2.3.2 Bases de Datos Deductivas Indefinidas.

Una base de datos deductiva indefinida difiere de una base de datos deductiva definida en el hecho de incorporar cláusulas indefinidas[GaMn84].

Una base de datos deductiva indefinida consiste de:

1. Un conjunto de axiomas que contiene hechos elementales y un conjunto de cláusulas definidas o indefinidas libres de funciones.
2. Un conjunto de restricciones de integridad.
3. Una metaregla: La negación como falla generalizada.

El concepto de la negación como falla se extiende a la negación como falla generalizada, la cual consiste de un conjunto E de todas las cláusulas positivas puras no probables, se asevera $\neg P(x)$ si y solo si $P(x) \vee C$ no es probable para cualquier C en E.

---

(1) Una lógica es monótona si dada una teoría T en la cual se puede probar una fórmula F, al agregar un axioma A a la teoría sigue permitiendo probar F.

Capítulo 3.

MECANISMO DE INFERENCIA.

3.1. El principio de resolución para la lógica proposicional.

El Principio de Resolución de Robinson para la lógica proposicional, se aplica a cualquier conjunto S de cláusulas para probar la insatisfacibilidad de S. El Principio de Resolución es un método de prueba por refutación. La idea esencial del principio de Resolución es verificar si S contiene la cláusula vacía o si la cláusula vacía puede derivarse de S.

Definición 3.1. Si A es un átomo, se dice que las literales A y $\neg A$ son complementarias, y el conjunto $\{A, \neg A\}$ se denomina un par complementario.

Definición 3.2. (Principio de Resolución) Para cualesquiera dos cláusulas C_1 y C_2, si existe una literal L_1 en C_1 que es complementaria a una literal L_2 en C_2, eliminando L_1 y L_2 de C_1 y C_2 respectivamente y construyendo la disyunción del resto de las cláusulas, se obtiene una cláusula denominada resolvente de C_1 y C_2.

Una propiedad importante de un resolvente consiste en que cualquier resolvente de dos cláusulas C_1 y C_2 es consecuencia lógica de C_1 y C_2. Si C_1 y C_2 son cláusulas unitarias, en caso de que exista un resolvente, éste es la cláusula vacía. Si un conjunto S de cláusulas no es satisfacible, se puede emplear el principio de resolución para generar la cláusula vacía a partir de S.

Definición 3.3. Dado un conjunto S de cláusulas, una resolución (deducción) de C en S es una secuencia finita de cláusulas $C_1, C_2, \ldots, C_k$, tales que $C_i(i = 1,\ldots,k)$ es una cláusula en S o un resolvente de cláusulas que preceden a C_i, y $C_k = C$. Una deducción de la cláusula vacía se denomina una refutación o una prueba de S.

3.2 Sustitución y Unificación.

La parte más importante de la aplicación del principio de resolución consiste en encontrar una literal en una cláusula que es complementaria a una literal en otra cláusula. En la lógica proposicional los átomos no tienen variables, sin embargo, en la lógica de primer orden se tiene que los átomos cuentan con términos que pueden ser símbolos de constantes, variables o funciones, por lo cual ya no es tan sencillo verificar si dos literales son complementarias.

Definición 3.4. Una sustitución es un conjunto finito de la forma $\{t_1/v_1, \ldots, t_n/v_n\}$, $n \geq 0$ donde cada v_i $(i = 1,\ldots,n)$ es una variable y cada t_i $(i = 1,\ldots,n)$ es un término diferente de v_i, tal que no existen dos elementos en el conjunto con la misma variable v_i después de la diagonal. Cuando $t_1,\ldots,t_n$ son símbolos de constantes, la sustitución se denomina sustitución base. La sustitución que no contiene elementos se denomina sustitución vacía (denotada por ϵ).

Definición 3.5. Dada la sustitución $\Theta = \{ t_1/v_1, \ldots, t_n/v_n \}$ y una expresión E, la expresión $E\Theta$ se obtiene reemplazando simultaneamente en E cada ocurrencia de v_i por t_i $(i = 1,\ldots,n)$ y se le denomina una instancia de E.

Definición 3.6. Sean dos sustituciones $\Theta = \{ t_1/x_1, \ldots, t_n/x_n \}$ y $\lambda = \{ u_1/y_1, \ldots, u_m/y_m \}$. La composición de Θ y λ, denotada por $\Theta \circ \lambda$, es la sustitución que se obtiene del conjunto

$$\{ t_1\lambda/x_1, \ldots, t_n\lambda/x_n, u_1/y_1, \ldots, u_m/y_m\}$$

borrando cualquier elemento $t_j\lambda/x_j$ para el cual $t_j\lambda = x_j$, y cualquier elemento u_i/y_i tal que y_i pertenece al conjunto $\{x_1, x_2, \ldots, x_n\}$.

Definición 3.7. Una substitución Θ se denomina un unificador para un conjunto $W = \{E_1,\ldots,E_k\}$, si y solo si $E_1\Theta = E_2\Theta = \ldots = E_k\Theta$. Si existe un unificador para W se dice que el conjunto es unificable.

Definición 3.8. Un unificador σ para un conjunto de expresiones $W = \{E_1,...,E_k\}$ es un unificador más general si y solo si para cada unificador Θ de W, existe una sustitución λ tal que $\Theta = \sigma \cdot \lambda$.

Al unificar dos o más expresiones se obtiene el conjunto de conflicto. Para un conjunto W no vacío de expresiones el conjunto de conflicto se genera revisando de izquierda a derecha término a término, en caso de que los términos comparados sean diferentes, se extrae de cada expresión de W dichos términos, por ejemplo, considérese el conjunto de expresiones:

$$W = \{ P(x,a,b),\ P(x,f(z),b),\ P(x,g(h(x),z),b) \}$$

comparando de izquierda a derecha tenemos que las expresiones tienen el mismo símbolo de raiz P y el primer término x de cada expresión es el mismo, el segundo término de cada expresión es desigual, por lo que forman parte del conjunto de conflicto, y verificando el tercer término de cada expresión es igual por lo que no se incluye en el conjunto de conflicto:

$$\{a,\ f(z),\ g(h(x),\ z) \}$$

La unificación de un conjunto de expresiones puede fallar por dos motivos: cuando existen dos términos (funciones o constantes) que no son variables, y son diferentes, o por la ocurrencia de una variable en la función que la sustituye.

El proceso de unificación se explicará en base al siguiente algoritmo dada su simplicidad, en lugar de explicarlo con el algoritmo empleado en esta tesis, debido a que es un algoritmo muy complejo y sería difícil explicarlo ahora[ChLe73].

Algoritmo de Unificación.

1.- $k = 0$; $W_k = W$ y $\sigma_k = \varepsilon$.

2.- Si W_k contiene sólo un elemento, termina y σ_k es un unificador más general para W; en caso contrario encontrar el conjunto de conflicto D_k de W_k.

3.- Si en el conjunto de conflicto existen elementos v_k y t_k tales que la variable v_k no aparece en t_k, ir al paso 4;
en caso contrario termina y W no es unificable.

4.- $\sigma_{k+1} = \sigma_k \{t_k / v_k\}$ y $W_{k+1} = W_k \{ t_k / v_k \}$

5.- $k = k + 1$ e ir al paso 2.

Ejemplo. Se desea encontrar el unificador más general para:

$$W = \{ P(y,b,f(z)),\ P(g(w),w,f(g(v))\ ) \}$$

siguiendo el algoritmo descrito, tenemos:

1.- $\sigma_0 = \epsilon$, $W_0 = W$. Debido a que W_0 tiene más de un elemento, σ_0 no es el unificador más general para W.

2.- $D_0 = \{y, g(w)\}$, donde $v_o = y$ no aparece en $t_0 = g(w)$.

3.- $\sigma_1 = \sigma_0 \circ \{t_0/v_0\} = \epsilon \circ \{g(w)/y\} = \{g(w)/y\}$

$W_1 = W_0\{t_0/v_0\}$

$= \{ P(y,b,f(z)),\ P(g(w),w,f(g(v)) \} \{g(w)/y\}$

$= \{ P(g(w),b,f(z)), P(g(w),w,f(g(v))\}$.

4.- Dado que W_1 contiene más de un elemento

$D_1 = \{b,w\}$

donde $v_1 = w$ y $t_1 = b$.

5.- $\sigma_2 = \sigma_1 \circ \{t_1/v_1\} = \{g(w)/y\} \circ \{b/w\} = \{ g(w)/y, b/w\}$

$W_2 = W_1\{t_1/v_1\}$

$= \{P(g(w),b,f(z)), P(g(w),w,f(g(v))\} \{b/w\}$

$= \{P(g(w),b,f(z)), P(g(w),b,f(g(v))\}$.

6.- W_2 sigue conteniendo más de un elemento, por lo que calculamos $D_2 = \{z,g(v)\}$ y tenemos $v_2 = z$ y $t_2 = g(v)$.

7.- $\sigma_3 = \sigma_2 \circ \{t_2/v_2\} = \{g(w)/y,b/w\} \circ \{g(v)/z\} = \{g(w)/y,b/w,g(v)/z\}$

$W_3 = W_2\ \{t_2/v_2\}$

$= \{P(g(w),b,f(z)), p(g(w),b,f(g(v))\} \{g(v)/z\}$

$= \{P(g(w),b,f(z))\}$

8.- Debido a que w3 contiene un sólo elemento, terminamos el algoritmo y el unificador más general es:

$$\sigma_3 = \{ g(w)/y,b/w,g(v)/z \}$$

3.3 El Principio de Resolución para la lógica de primer orden.

Debido a que la lógica proposicional es un formalismo sencillo existen muchas ideas que no pueden ser expresadas en ésta, mientras que la lógica de primer orden brinda un formalismo más poderoso.

Definición 3.9. Si dos o más literales (con el mismo signo) de una cláusula C tienen un unificador más general σ, $C\sigma$ se denomina un factor de C. Si $C\sigma$ es una cláusula unitaria se dice que es un factor unitario de C.

Ejemplo: Sea la cláusula $C = P(x,y) \vee P(u,f(v)) \vee \neg Q(y)$ donde las dos primeras literales tienen el unificador más general

$$\sigma = \{x/u, f(v)/y\},$$

por lo tanto

$$C\sigma = P(x,f(v)) \vee \neg Q(f(v))$$

es un factor de C.

Definición 3.10. Sean C_1 y C_2 dos cláusulas (denominadas cláusulas progenitoras) que no tienen variables en común, y sean L_1 y L_2 dos literales en C_1 y C_2 respectivamente. Si L_1 y $\neg L_2$ tienen un unificador más general σ, la cláusula $(C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma)$ se denomina un resolvente binario de C_1 y C_2, y las literales L_1 y L_2 se denominan las literales sobre las cuales se lleva a cabo la resolución.

Definición 3.11.- Un resolvente de cláusulas(progenitoras) C_1 y C_2 es uno de los siguientes resolventes binarios:

1. Un resolvente binario de C_1 y C_2.
2. Un resolvente binario de C_1 y un factor de C_2.
3. Un resolvente binario de un factor de C_1 y C_2.
4. Un resolvente binario de un factor de C_1 y un factor de C_2.

El principio de resolución es una regla de inferencia que genera resolventes de un conjunto de cláusulas. Este principio es completo, es decir, siempre genera la cláusula vacía de un conjunto no satisfacible de cláusulas.

3.4 El proceso de deducción.

En las bases de datos deductivas existen dos métodos para realizar el proceso de deducción: el método interpretativo y el método compilado [GaMn84]. En el primero se combina el uso de las leyes deductivas con la búsqueda en la base de datos, mientras que en el método compilado se retrasa el acceso a la base de datos hasta que se han concluido las transformaciones. Ambos métodos serán descritos por medio de un ejemplo.

Método Interpretativo.

Considérese la siguiente instancia de una base de datos.

Padre	
Papá	Hijo
Mario	Susana
Mario	Alejandra
Mario	Jorge

Madre	
Mamá	Hijo
Sara	Mario
Sara	Claudia
Sara	Antonieta

Esposo	
Esposo	Esposa
Armando	Sara

representada por las siguientes cláusulas:

C_1: Padre(Mario,Susana)←
C_2: Padre(Mario,Alejandra)←
C_3: Padre(Mario,Jorge)←
C_4: Madre(Sara,Mario)←
C_5: Madre(Sara,Claudia)←
C_6: Madre(Sara,Antonieta)←
C_7: Madre(Antonieta,Sara)←
C_8: Esposo(Armando,Sara)←

Sean las siguientes reglas de deducción:

R_1: Abuela(x,y) ← Madre(x,z) ∧ Madre(z,y)
R_2: Abuela(x,y) ← Madre(x,z) ∧ Padre(z,y)
R_3: Abuelo(x,y) ← Abuela(z,y) ∧ Esposo(x,z)

Supóngase que se desea conocer de quien es abuelo Armando, lo cual se representa con la cláusula:

Q:← Abuelo (Armando, y).

la cual unifica con R_3 obteniendo la sustitución {Armando / x} para generar el resolvente:

C_9: ←Abuela(z,y) ∧ Esposo(Armando,z)

resolviendo primero para Esposo(Armando,z) el cual unifica con C_8 se obtiene:

C_{10}: ←Abuela(Sara,y)

para resolver esta cláusula se puede aplicar R_1 o R_2, supóngase que se aplica primero R_2 obteniendo la cláusula:

C_{11}: ←Madre(Sara,z) ∧ Padre(z,y)

al unificarla con C_4 obtenemos el primer valor para z quedando por resolver:

C_{12}: ←Padre(Mario,y)

la cual unifica con C_1, para obtener y=Susana. De manera similar, unificando C_{12} con C_2 se obtiene un segundo valor para y, y = Alejandra, y unificando C_{12} con C_3 se obtiene y=Jorge.

Nuevamente tomando C_{11} para unificar con C_5 se obtiene otro valor para z, z=Claudia, generando el resolvente:

C_{13}: ←Padre(Claudia,y)

el cual no unifica con ninguna cláusula, de igual manera al unificar C_{11} con C_6 se obtiene para z el valor z = Antonieta, quedando:

C_{14}: ←Padre(Antonieta,y)

que no unifica con ninguna cláusula.

Si ahora para resolver C_{10} se aplica R_1 se obtiene:

C_{15}: ←Madre(Sara,z) ∧ Madre(z,y)

la cual puede unificar con C_4,C_5 y C_6, obteniendo un valor a la vez para z, z = Mario, z = Claudia, z = Antonieta, de los cuales se emplea z = Antonieta de acuerdo a la base de datos, obteniendo el resolvente:

C_{16}: ←Madre(Antonieta,y)

donde se obtiene un valor más para y, y = Paola.

Método Compilado.

Existen dos variantes del método compilado para efectuar la deducción: pseudo-compilación y compilación. La técnica de pseudo-compilación consiste en compilar sólo un camino a la vez por donde es probable encontrar la solución, y en caso de no obtener la solución por ese camino se compila otro camino. Por ejemplo, considérese la misma base de datos del ejemplo anterior con la misma consulta

Q:← Abuelo(Armando,y)

Unificando Q con R_3 y sustituyendo {Armando/x} se obtiene la expresión:

E_1: ←Abuela(z,y) ∧ Esposo(Armando,z).

para la literal Abuela(z,y) se puede emplear R_1 o R_2, si se utiliza R_1 y le aplicamos la sustitución {z/z1} se obtiene:

E_2: ← Madre(z,z1) ∧ Madre(z1,y) ∧ Esposo(Armando,z)

dado que la expresión E_2 sólo contiene relaciones extensionales, ésta se evalúa en la base de datos. De manera similar, si ahora se emplea R_2 y se aplica la misma sustitución se obtiene la expresión:

E_3: ← Madre(z,z1) ∧ Padre(z1,y) ∧Esposo(Armando,z).

La técnica de compilación consiste en compilar todos los caminos. Nuevamente considérese el mismo ejemplo con la consulta Q←Abuelo(Armando,y). Dado que existen dos caminos se obtiene la siguiente expresión:

E:(Madre(z,z1)∧Madre(z1,y) ∧Esposo(Armando,z)) ∨(Madre(z,z1) ∧Padre(z1,y) ∧Esposo(Armando,z))

cuyo árbol que representa esta expresión se muestra en la siguiente figura:

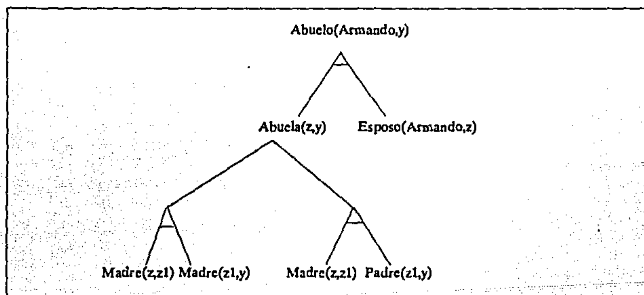


Figura 3.1. Arbol de derivación.

Capítulo 4.

DESCRIPCIÓN DEL SISTEMA.

4.1 Arquitectura General del Sistema.

En la figura 4.1 se muestra la arquitectura del sistema objeto de esta tesis, la cual corresponde a un subconjunto de la arquitectura que aparece en la figura 1.2. Los principales componentes del sistema son: lenguaje orientado a reglas, parser, monitor de transacciones e intérprete.

El lenguaje orientado a reglas consta de cuatro objetos: funciones, relaciones, reglas de deducción y restricciones; de los cuales para este trabajo sólo se consideran relaciones y reglas. El lenguaje diseñado permite generar una base de datos convencional, si el programa sólo incluye relaciones, o permite la generación de un sistema deductivo, si el programa incluye reglas deductivas además de relaciones.

El parser efectúa tanto el análisis léxico y el análisis sintáctico de un programa, como la generación de la representación interna de las reglas para el proceso de deducción y las estructuras necesarias para la definición de la base de datos extensional.

El monitor de transacciones está compuesto por dos submódulos: el monitor y la interfaz con el manejador de bases de datos relacional (RDBMSI). El monitor transfiere el control al intérprete para el proceso deductivo. La RDBMSI se emplea para crear las relaciones base y permite insertar, borrar o modificar información en las relaciones de la base de datos.

El intérprete es el núcleo del sistema, y constituye una extensión de un sistema manejador de bases de datos relacional para soportar la deducción. El intérprete se encarga de deducir conocimientos de hechos almacenados en la base de datos utilizando el método compilado

para la deducción [GaMn84].. Los componentes del intérprete son : el generador del árbol de inferencia, el submódulo de unificación y el submódulo de evaluación . El árbol se construye para una consulta empleando las reglas almacenadas en la base de reglas deductivas. El submódulo de unificación obtiene el unificador más general para dos átomos, en caso de existir. El submódulo de evaluación obtiene una expresión equivalente del árbol de inferencia en SQL, y efectúa la evaluación de la expresión obtenida.

El pizarrón es el área de trabajo empleada para almacenar resultados intermedios. En el pizarrón se almacena el árbol de inferencia construido por el intérprete, así omo las vistas generadas por el módulo de evaluación.

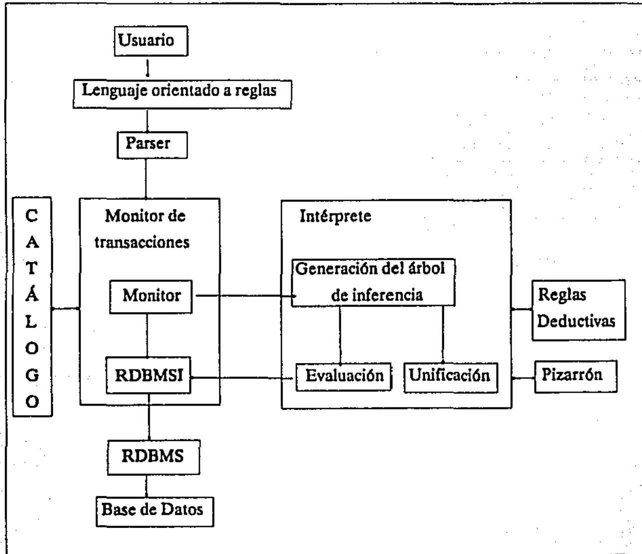


Figura 4.1. Arquitectura del sistema.

4.2 Las Cláusulas de Horn y el álgebra relacional.

Existen cláusulas de Horn para las cuales se puede encontrar una expresión equivalente en el álgebra relacional. Considere la siguiente cláusula de Horn:

$$Q \leftarrow P1 \wedge \ldots \wedge Pn$$

se define una relación asociada al cuerpo de dicha cláusula, la cual tendrá el esquema $(x_1,x_2,\ldots,x_n)$, donde las x_i $(i = 1,\ldots,n)$ son todas las variables que aparecen en el antecedente de la cláusula, y el conjunto de tuplos t_k de la forma $(c_{k1},c_{k2},\ldots,c_{kn})$ son las sustituciones c_{ki} por x_i tales que todos los predicados P_j $(j = 1,..,n)$ son verdaderos, considérese por un momento que todos los predicados P_i son relaciones, tal que P_i consiste de todos los tuplos $(c_1, c_2,\ldots,c_r)$ tal que $P_i(c_1,c_2,\ldots,c_r)$ es verdadero, la cláusula de Horn definida anteriormente puede representarse por medio de una lista de operaciones del álgebra relacional.

Considérese la siguiente cláusula:

$$Q(x,y) \leftarrow P(x,y,z)$$

la cual es equivalente a la operación de proyección P[x,y], representada en SQL como:

```
CREATE VIEW  Q(x,y)  AS
        SELECT  x,y
        FROM P
```

Supóngase una cláusula de la siguiente forma:

$$Q(x,w) \leftarrow P1(x,y) \wedge P2(y,w)$$

equivalente a la operación de join, y se representa en SQL por medio de la expresión:

```
CREATE VIEW  Q(x,w)  AS
        SELECT  x,w
        FROM  P1, P2
        WHERE P1.y = P2.y
```

Se deben considerar algunas restricciones sobre las variables de las reglas para evitar que las reglas generen operaciones sobre relaciones infinitas[Ullm88], considérese los siguientes casos:

i) Cuando aparecen variables en el consecuente de la regla y no ocurren en el antecedente. Por ejemplo, la cláusula:

$$Ama(x,y) \leftarrow Amante(y)$$

define un conjunto infinito de pares Ama(x,y) si la relación Amante es un conjunto infinito, debido a que el primer argumento de Ama varía sobre un conjunto infinito. Este problema se resuelve empleando cláusulas de rango restringido.

ii) Cuando una variable aparece únicamente en algún predicado relacional aritmético. Por ejemplo, sea la siguiente cláusula:

$$Mayor(x,z) \leftarrow x > z$$

la cual representa una relación infinita si x y z se definen sobre los enteros o cualquier conjunto infinito. Para solucionar este problema es necesario que se "limiten" las variables que aparecen en la regla.

Definición 4.1. Se dice que las variables de una regla están limitadas si:

- Cada variable x que aparece en una submeta $x = a$ o $a = x$, donde a es una constante, está limitada.
- Cada variable que aparece como argumento en un prediado ordinario del antecedente de la regla está limitada.
- La variable x está limitada si aparece en una submeta $x = z$ o $z = x$, y z es una variable limitada.

Es necesario introducir el operador de negación para poder expresar algunas ideas más complejas, sin embargo, ésto ocasiona algunos problemas dado que no se contaría con cláusulas de Horn. Un predicado negado se puede considerar como el complemento de una relación, aunque, el complemento de una relación no es un término bien definido. El álgebra relacional cuenta con el operador de diferencia y no incluye el operador de complemento debido a que se tendría que especificar la relación o el dominio de los valores posibles con respecto al cual se toma el complemento. Supóngase que se cuenta con cláusulas de la forma:

$$Q(x,y) \leftarrow P1(x,y) \land \neg P2(x,y)$$

las que se pueden representar en el álgebra relacional por medio de la operación de diferencia, es decir, P1 — P2, y en SQL se expresan como:

```
CREATE VIEW  Q(x,y)  AS
        SELECT  x,y
        FROM  P1
        MINUS
        SELECT  x,y
        FROM P2
```

La negación puede acarrear problemas si no se toman en cuenta algunas consideraciones sobre las variables de la regla que contiene predicados negados:

i) En el caso de que aparezcan variables sólo en submetas negadas, y estas variables no ocurren en el consecuente de la regla. Por ejemplo, considérese la regla [Ullm88]:

$$\text{Soltero}(x) \leftarrow \text{Masculino}(x) \wedge \neg \text{Casado}(x,z)$$

al calcular la relación efectuando el join de Masculino(x) con el "complemento" de Casado(x,z), se obtendría el conjunto de pares (x,z) tales que x es masculino y z no está casado con x, y al proyectar este conjunto sobre x se encuentra que "soltero" son todos las personas de sexo masculino que no están casadas con alguna persona en el universo, es decir, existe alguna z tal que z no está casado con x. Para evitar estas divergencias entre lo que intuitivamente se espera que una regla debería significar y lo que se obtendría al interpretar la negación como el complemento de la relación, es necesario no utilizar una variable en la submeta negada que no aparezca en otra submeta. Esta restricción se puede solucionar reescribiendo la regla para que no ocurran tales variables. Por ejemplo, reescribiendo la regla del ejemplo anterior se hace que los atributos de las relaciones involucradas sean los mismos:

$\text{Esposo}(x) \leftarrow \text{Casado}(x,z)$

$\text{Soltero}(x) \leftarrow \text{Masculino}(x) \wedge \neg \text{Esposo}(x)$

la cual expresada por medio de operaciones del álgebra relacional queda como:

```
CREATE VIEW Esposo(x) AS
        SELECT x
        FROM Casado
```

y

```
CREATE VIEW Soltero(x) AS
    SELECT x
    FROM Masculino
    MINUS
    SELECT x
    FROM Esposo
```

ii) Para el caso de tener menos variables en el predicado negado que en los predicados ordinarios, considérese el siguiente ejemplo:

$$\text{Compra}(x,z) \leftarrow \text{Gusta}(x,z) \wedge \neg \text{Frágil}(z)$$

la cual se interpreta como x compra z si a x le gusta z y z no es frágil. Esta regla se puede expresar en el álgebra relacional como el *join* de Gusta(x,z) y una nueva relación Irrompible(z), la cual es una relación infinita. La idea para solucionar el problema consiste en agregar una regla para obtener todos los objetos posibles que podrían agradar de la relación Gusta y emplear otra regla para concatenar estos objetos a la relación Frágil, obteniendo:

$\text{Atractivo}(z) \leftarrow \text{Gusta}(x,z)$

$\text{Inaceptable}(x,z) \leftarrow \text{Atractivo}(z) \wedge \text{Frágil}(x)$

$\text{Compra}(x,z) \leftarrow \text{Gusta}(x,z) \wedge \neg \text{Inaceptable}(x,z)$

Es posible tener varias cláusulas con el mismo consecuente, de la siguiente forma:

$$Q \leftarrow A_1$$
$$Q \leftarrow A_2$$
$$\vdots$$
$$Q \leftarrow A_n$$

donde cada Ai es un antecedente, a este conjunto de cláusulas se le denomina no normalizado, las cuales se pueden representar con una regla de la forma:

$$Q \leftarrow A_1 \vee A_2 \vee \ldots \vee A_n$$

la cual se puede representar por medio del operador de unión del álgebra relacional:

$$A_1 \cup A_2 \cup \ldots \cup A_n$$

4.3 Descripción del lenguaje orientado a reglas.

El lenguaje diseñado brinda las ventajas de los esquemas lógicos para el proceso de deducción y para la representación del conocimiento, así como emplea las facilidades de las bases de datos para manipular y almacenar grandes volúmenes de datos.

El lenguaje consiste de cuatro tipos de objetos: funciones, relaciones, reglas de deducción y restricciones de integridad. De estos cuatro elementos un programa debe incluir cuando menos relaciones, esto es, las funciones, las reglas y las restricciones de integridad son opcionales[1]. Si el programa contiene sólo relaciones (o relaciones y restricciones), el sistema representa una base de datos convencional; si además de las relaciones se incluyen reglas de deducción, representa un sistema deductivo. En la implantación solamente se trabajó con relaciones y reglas de deducción, quedando para otros trabajos la inclusión de funciones y restricciones de integridad.

El lenguaje está organizado en cuatro bloques como se muestra en la descripción parcial de la figura 4.2, esto es, cada elemento del lenguaje se define dentro de un bloque. El orden como deben aparecer declarados estos bloques es: funciones, relaciones, reglas deductivas y restricciones de integridad. Para identificar el inicio de cada bloque se emplea una palabra clave: FUNCTIONS_DEF para funciones, RELATIONS_DEF para relaciones, RULES_DEF para reglas y CONSTRAINTS_DEF para las restricciones. Por ejemplo, el programa de la figura 4.3 representa un sistema deductivo para préstamos de libros de una biblioteca.

La declaración de una relación comienza con la palabra clave RELATION, seguida por su nombre y sus atributos. De acuerdo al ejemplo son definidas las relaciones Libro, Ejemplar, Lector y Prestamo. Para cada atributo es necesario especificar el nombre y tipo, y dependiendo del tipo se indica una o dos longitudes, por ejemplo, la relación Libro contiene siete atributos, de los cuales cinco son de tipo caracter y dos son numéricos.

Las reglas deductivas se declaran comenzando con la palabra clave RULE, un nombre con el cual el usuario identifica la regla, el consecuente, el operador lógico <- y el antecedente.

(1) En el apéndice A se muestra la descripción en BNF de la gramática del lenguaje.

En nuestro ejemplo están definidas tres reglas identificadas por R1, R2 y R3. La regla R1 representa cuando un libro está disponible, un libro se encuentra disponible si está en existencia y no se ha prestado. Un libro ha sido prestado si existe el libro y se ha realizado un préstamo del mismo, representado en la regla R3, y un libro se encuentra en existencia si existe el libro y existen ejemplares del mismo, lo cual se representa por medio de la regla R2.

```
<DDB> ::= [<Functions>] <Relations> [<Rules>] [<Constraints>]
<Functions> ::= FUNCTIONS_DEF <Functions_def>
<Functions_def> ::= <Function_def> | <Function_def> <Functions_def>
<Function_def> ::= [<fn_type>] <fn_declarator> <fn_body>
<Relations> ::= RELATIONS_DEF <Relations_def>
<Relations_def> ::= <Relation_def> | <Relation_def> <Relations_def>
<Relation_def> ::=RELATION <Relation_name> ( Atributes ) [ DOC (<Comment>)]
<Relation_name> ::= <Identifier>
<Atributes> ::= <Atribute> | <Atribute> ; <Atributes>
<Atribute> ::= <Atr_name> : <Atr_type> [:<Atr_size1> ] [ :<Atr_size2> ]
                [ DOC ( <Comment> )]
<Constraints> ::= CONSTRAINTS_DEF <Const_definitions>
<Const_definitions> ::= <Const_definition> | <Const_definition> <Const_definitions>
<Const_definition> ::= CONSTRAINT <Const_name> :
                [ <Consequent> ] <- <Antecedent> [DOC( < Comment> )]
<Rules> ::= RULES_DEFINITION <Rules_def>
<Rules_def> ::= <Rule_def> | <Rule_def> <Rules_def>
<Rule_def>::=RULE<Rule_name>:<Consequent> <-<Antecedent>[DOC(<Comment>)]
<Consequent> ::= <Predicate>
<Predicate> ::= <User_Pred> | <System_Pred>
<User_Pred> ::= <Predicate_name> ( <Terms_list> )
<Antecedent> ::= <Literals> | <Lit_Ant>
<Literals> ::= <Literal> | <Literal><OR><Literal> | <Literal><AND><Literal>
```

Figura 4.2 Descripción parcial de la gramática del lenguaje

```
RELATIONS_DEF
        RELATION Libro
                ( colocacion : string : 25;
                  autor : string : 40;
                  titulo : string : 120;
                  editorial : string : 30;
                  edicion : integer : 2;
                  ano : integer : 6;
                  lugar : string : 20 )
        RELATION Ejemplar
                ( colocacion : string : 25;
                  num_ejem : integer : 2 )
        RELATION Lector
                ( num_reg : string : 10;
                  nombre : string : 40;
                  fecha_alta : integer : 6;
                  domicilio : string : 80;
                  telefono : string : 15 )
        RELATION Prestamo
              ( colocacion : string : 25;
                num_ejem : integer : 2;
                num_reg : string : 10;
                fecha_dev : integer : 6 )
RULES_DEF
  RULE R1: Disponible( colocacion, num_ejem, autor, titulo) <-
           Existencia(colocacion, num_ejem,autor, titulo, editorial, edicion, ano, lugar) AND
         NOT Prestado(colocacion, num_ejem, autor, titulo, editorial, edicion, ano, lugar)
  RULE R2: Existencia(colocacion, num_ejem, autor, titulo, editorial, edicion, ano, lugar) <-
           Libro(colocacion, autor, titulo, editorial, edicion, ano, lugar) AND
           Ejemplar( colocacion, num_ejem)
  RULE R3: Prestado( colocacion, num_ejem, autor, titulo, editorial, edicion, ano, lugar) <-
           Existencia(colocacion, num_ejem, autor, titulo, editorial, edicion, ano, lugar) AND
         Prestamo ( colocacion, num_ejem, num_reg, fecha_dev)
```

Figura 4.3 Programa escrito en el Lenguaje Orientado a reglas

El consecuente está dado por un predicado definido por el usuario de la forma $P(t_1,..,t_n)$, donde P es el nombre del predicado y $t_i, i = 1,..,n$ es una lista de términos. Por ejemplo, el consecuente de R1 es Disponible y tiene como argumentos cuatro términos variables. El antecedente es una lista de literales conectada por los operadores lógicos AND y OR. Como se definió previamente cada literal es un predicado o la negación de un predicado, por lo cual se incluye en el lenguaje el operador lógico NOT. Cada predicado del antecedente puede ser una relación base, una relación derivada o un predicado del sistema. Los predicados del sistema consisten de los predicados relacionales: menor (<), mayor (>), igual(=), etc.

4.4. Descripción del Algoritmo de Unificación.

El algoritmo de unificación que se emplea en el sistema fue desarrollado por Martelli y Montanari [MaMo82]. Este método trata el problema de la unificación como la solución de un sistema de ecuaciones.

$$\begin{aligned} s_1 &= t_1 \\ &\cdot \\ &\cdot \\ s_n &= t_n. \end{aligned}$$

donde un unificador es cualquier sustitución que hace todos los pares de términos s_i, t_i $(i = 1,...,n)$ iguales simultáneamente.

Con el propósito de encontrar el unificador más general es necesario contar con transformaciones que preserven los conjuntos de todos los unificadores. Existen dos transformaciones: reducción de términos y eliminación de variables .

1. Reducción de términos. Dada la ecuación

$$f(s_1, \dots, s_n) = f(t_1, \dots, t_n)$$

donde los símbolos de función son los mismos. El nuevo conjunto de ecuaciones se obtiene al reemplazar esta ecuación por el conjunto de ecuaciones:

$$\begin{aligned} s_1 &= t_1 \\ &\cdot \\ &\cdot \\ s_n &= t_n \end{aligned}$$

si $n = 0$, entonces f es un símbolo de constante, y la ecuación se elimina.

2. Eliminación de variables. Sea $x = t$ una ecuación de un conjunto de ecuaciones, donde x es una variable y t es un término. El nuevo conjunto de ecuaciones se obtiene al aplicar la sustitución $\Theta = \{ t/x \}$ a las ecuaciones restantes en el conjunto (sin eliminar $x = t$).

La unificación de un conjunto de expresiones puede fallar por dos motivos: 1) si existen dos términos (funciones o constantes) que no son unificables, 2) por la ocurrencia de una variable en el término que la sustituye.

Teorema 4.1. Sea S un conjunto de ecuaciones y $f(s_1,\ldots,s_n) = f'(t_1,\ldots,t_n)$ una ecuación en S, si $f \neq f'$, entonces no existe un unificador para S, en caso contrario, aplicando reducción de términos a la ecuación dada se obtiene el sistema de ecuaciones resultante S', el cual es equivalente a S.

Teorema 4.2. Sea S un conjunto de ecuaciones, al aplicar eliminación de variables a alguna ecuación $x = t$ de S, se obtiene un nuevo conjunto de ecuaciones S', donde si x aparece en t (t diferente de x), no existe un unificador para S; en caso contrario, S y S' son equivalentes, tengan o no unificador.

En el proceso de solución al manipular un conjunto de ecuaciones frecuentemente es necesario agruparlas, por lo que se introduce el concepto de multiecuación . Una multiecuación es la generalización de una ecuación, y tiene la forma $S = M$, donde S es un conjunto no vacío de variables y M es un multiconjunto[1] de términos, los cuales no son variables. Por ejemplo: $\{x1,x2\} = (t1,t2,t3)$; donde $S = \{x1,x2\}$ y $M = (t1,t2,t3)$.

Con el propósito de obtener conjuntos equivalentes de multiecuaciones se definen transformaciones de conjuntos de multiecuaciones, introduciendo las nociones de parte común y frontera de un multiconjunto de términos. La parte común de un multiconjunto de términos se genera tomando los términos que son comunes a ellos o los términos que generalizan a los otros términos[2]. Por ejemplo, para el siguiente multiconjunto de términos:

$$(f(x,h(a,g(x)),y),\ f(g(z),h(w,v),h(a,g(z))))$$

(1) Un multiconjunto es una familia de elementos donde existe un ordenamiento entre ellos, pero es posible que se repitan elementos.

(2) Una variable generaliza a una constante o a una función. Cuando se tienen dos variables es indistinto cual variable se seleccione.

Para generar la parte común se compara de izquierda a derecha, término a término, dado que ambos términos tienen el mismo símbolo de raíz, la parte común tendrá como símbolo de raíz a f. Al comparar el primer término que aparece como argumento de cada función se tiene la variable x y la función g(z), por lo que el primer argumento de la parte común es la variable x, la cual generaliza a g(z). Los siguientes argumentos son ambos símbolos de función por lo que el segundo término en la parte común es la función h(w,v), debido a que a su vez w generaliza la constante a y v generaliza la función g(x). De igual manera para el último término, la variable y generaliza a h(a,g(z)), obteniendo la expresión final para la parte común:

$$f(x, h(w, v), y)$$

En la frontera de un multiconjunto de términos se obtiene el conjunto de conflicto. La frontera se obtiene buscando para cada variable que aparece en la parte común los términos que fueron generalizados por dicha variable y se crean las multiecuaciones correspondientes. Por ejemplo, la frontera del conjunto de multiecuaciones anterior se obtiene generando para cada variable de la parte común la multiecuación correspondiente como sigue: para la variable x se genera la multiecuación $\{x\} = (g(z))$, debido a que x generalizó al término g(z), para w se genera la multiecuación $\{w\} = (a)$, dado que la constante a fue generalizada por w. De manera similar para los términos restantes, obtenemos:

$$\begin{aligned} \{ \{x\} &= (g(z)), \\ \{w\} &= (a), \\ \{v\} &= (g(x)), \\ \{y\} &= (h(a, g(z))) \} \end{aligned}$$

Reducción de Multiecuaciones.

Sea Z un conjunto de multiecuaciones que contiene una multiecuación $S = M$, tal que M es diferente del vacío y tiene una parte común C y una frontera F. El nuevo conjunto Z' de multiecuaciones se obtiene de la siguiente manera:

$$Z' = (Z - \{ S=M \}) \cup \{ S = (C) \} \cup F$$

Teorema 4.3. Sea $S = M$ (M diferente del vacío) una multiecuación de un conjunto Z de multiecuaciones, si no se puede obtener la parte común de M, o si alguna variable de su parte S pertenece a la parte S de alguna multiecuación de la frontera F de M, entonces no existe un

unificador para Z. En caso contrario, si se aplica reducción de multiecuaciones a la multiecuación S = M, se obtiene un conjunto Z' equivalente de multicuaciones.

Definición 4.2. Se dice que un conjunto Z de multiecuaciones es compacto si se cumple que :

$$\forall (S = M), (S' = M'),\ S \cap S' = \varnothing$$

Dado que la reducción de multiecuaciones no garantiza obtener un conjunto compacto de multiecuaciones se introduce la siguiente transformación:

Compactación.

Sea Z un conjunto de multiecuaciones no compacto y sea R una relación entre pares de multiecuaciones de Z, tal que (S = M) R (S' = M'), si y solo si $S \cap S' \neq \varnothing$, y sea R" la cerradura transitiva de R. La relación R" particiona Z en clases de equivalencia. Para obtener el conjunto compacto Z', se fusionan ("merge") todas las multiecuaciones que pertenecen a la misma clase de equivalencia, obteniéndose una multiecuación equivalente al conjunto de multiecuaciones de cada clase de equivalencia.

Para el proceso de solución se establece un sistema de multiecuaciones denominado R, el cual consta de dos partes: Una parte T que es una secuencia de multiecuaciones, denominada parte triangular o resuelta y una parte U que es un conjunto de multiecuaciones, denominada parte no resuelta. Un sistema de multiecuaciones R presenta las siguientes características:

i) Los conjuntos de variables que están en los lados izquierdos de todas las multiecuaciones de T y U, contienen todas las variables y son disjuntos;

ii) Todas las multiecuaciones de T tienen un solo término en su lado derecho.

iii) Cualquier variable perteneciente al lado izquierdo de alguna multiecuación de T, sólo puede ocurrir en el lado derecho de cualquier multiecuación que la precede.

Dado que el proceso de solución puede llevar por un camino muy largo, y el unificador obtenido puede no ser el unificador más general, es conveniente definir algún criterio para la selección de las multiecuaciones. Para elegir más eficientemente las multiecuaciones se asocia un contador a cada una de ellas. Las variables de la parte S de una multiecuación pueden

aparecer en las otras multiecuaciones. Dicho contador nos indica cuantas veces aparecen estas variables en las partes M de las otras multiecuaciones. Este contador nos permite establecer un ordenamiento parcial de las multiecuaciones.

El proceso de unificación consiste en seleccionar alguna multiecuación de la parte U que esté en el tope del ordenamiento parcial, de la cual se calcula la parte común y la frontera aplicando las transformaciones de multiecuaciones, ésto se repite mientras haya multiecuaciones en la parte U o mientras no ocurra una falla.

Algoritmo de Unificación.

1. Sea R un sistema de multiecuaciones formado de una parte U y una parte T; T = ∅;
2. Repite
   - 2.1 Selecciona una multiecuación S = M de U tal que las variables en S no ocurren en ningún otro lugar dentro de U.
     Si una multiecuación con esta propiedad no existe entonces termina con falla.
   - 2.2 Si M es el vacío
     Transfiere S= M de U al final de T
     en caso contrario
     - 2.2.1 Calcula la parte común C y la frontera F de M.
       Si M no tiene parte común, termina con falla.
     - 2.2.2 Transforma U aplicando Reducción de multiecuaciones y Compactación a la multiecuación seleccionada.
     - 2.2.3 Transfiere la multiecuación S = (C) de U al final de T.

   hasta que la parte U de R esté vacía.
3. Termina con éxito.

Para construir el sistema inicial de multiecuaciones, se genera una multiecuación con contador cero, la cual contiene en M los términos a unificarse y contiene en S una variable distinta a todas las variables de M. Por cada variable v_i de la multiecuación generada que aparece en M, se construye una multiecuación con v_i en S y parte M nula.

Ejemplo: Considérese las siguientes expresiones a unificar:

s = f(x1, g(x2,x3), x2, b)

t = f(g(h(a,x5), x2), x1, h(a,x4), x4)

obtenemos el sistema inicial:

U : {

[0] {x} = (f(x1, g(x2,x3), x2,b), f(g(h(a,x5), x2), x1, h(a,x4), x4)),

[2] {x1} = ∅,

[3] {x2} = ∅,

[1] {x3} = ∅,

[2] {x4} = ∅,

[1] {x5} = ∅

}

T : ()

donde los corchetes contienen el contador asociado a las variables de cada multiecuación. Se elige la primera multiecuación dado que su contador es cero y se calcula la parte común, obteniendose:

{x} = (f(x1,x1,x2,x4))

y la frontera

{x1} = (g(h (a, x5), x2))

{x1} = (g(x2, x3))

{x2} = (h(a, x4))

{x4} = (b)

la parte común se traslada a T y dado que el sistema resultante puede no ser compacto, se efectúa compactación. Durante la compactación por cada multiecuación de la frontera se decrementa el contador de la multiecuación correspondiente de la parte U donde aparecen las mismas variables en S. Para nuestro ejemplo debido a que x1 aparece en dos multi ecuaciones de la frontera, el contador asociado a la multiecuación {x1} = ∅ de U se decrementa en dos, procediendo de manera similar con el resto de las multiecuaciones se obtiene:

```
U: {
      [0] {x1} = ( g (h(a,x5), x2), g(x2,x3)),
      [2] {x2} = (h(a,x4)),
      [1] {x3} = ∅ ,
      [1] {x4} = ( b ),
      [1] {x5} = ∅
   }
T: ( {x} = ( f( x1, x1, x2, x4) ) )
```

Después de seleccionar la multiecuación con contador cero obtenemos:

```
U : {
      [0] {x2, x3} = ( h( a, x4), h(a, x5)),
      [1] {x4} = ( b ),
      [1] {x5} =∅
   }

T : (
      {x} = ( f(x1, x1, x2, x4)),
      {x1} = ( g(x2,x3))
   )
```

Nuevamente tomando la multiecuación en el tope del ordenamiento parcial con contador cero.

```
U : { [0] {x4,x5} = ( b )
    }
T : (
      {x} = ( f( x1, x1, x2, x4)) ,
      {x1} = ( g(x2,x3)),
      {x2,x3} = ( h(a,x4))
   )
```

Finalmente se toma la última multiecuación de U y se obtiene el sistema:

U : ∅

T : (

{x} = (f(x1, x1, x2,x4)),

{x1} = (g(x2,x3)),

{x2, x3} = (h(a,x4)),

{x4,x5} = (b)

)

Cabe mencionar que este algoritmo integra la verificación de ocurrencia en el proceso de unificación y no la deja al final como en otros algoritmos. La implantación de la verificación de ocurrencia se efectúa durante la selección de una multiecuación, revisando que no existan ciclos entre variables.

Definición 4.3. Dos términos son consistentes si al menos uno de ellos es variable o si tienen el mismo símbolo de raíz y sus argumentos también son consistentes.

Extendiendo la definición anterior para más de dos términos, tenemos que son consistentes si todos los pares de términos son consistentes. En este algoritmo se lleva a cabo la verificación de consistencia de términos introduciendo una estructura propia (denominada multitérminos) para representar los lados derechos de las multiecuaciones.

Definición 4.4. Un multitérmino puede ser el vacío o tener la forma f(E1,...,En) donde f es un símbolo de predicado o función y cada E_i (i = 1 , . . . , n) está constituida por una pareja $< S_i, M_i >$, donde S_i es un conjunto de variables y M_i es un multitérmino.

Ejemplo. Sea la siguiente expresión

P(x1, f(h(x2, "a"), x3))

dado que el primer argumento es variable, la primera pareja del multitérmino $<S_1,M_1>$, tiene M_1 nula y la variable x1 pertenece a S_1, quedando representada por $<\{x1\},\emptyset>$, el segundo argumento es una función, por lo que S_2 es nula y M_2 representa la función f(h(x2,"a"),x3), la expresión para el multitérmino queda:

$$P(< \{x1\}, \emptyset >, < \emptyset \ , f(< \emptyset \ , h(< \{x2\} , \emptyset >, < \emptyset \ ,\text{"a"} >) >, < \{x3\}, \emptyset >) >)$$

Capítulo 5.

IMPLANTACIÓN DEL SISTEMA.

5.1 Implantación del Algoritmo de Unificación.

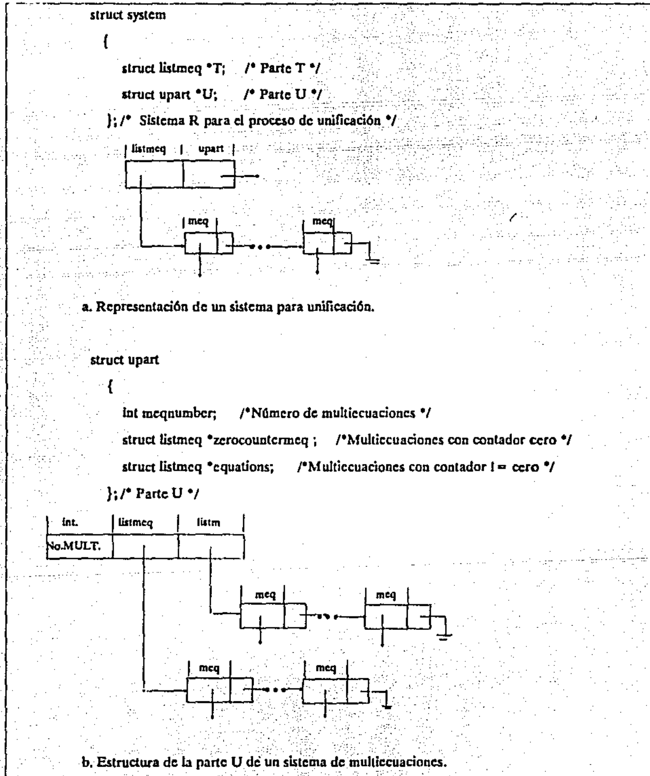
Como se mencionó en la sección 4.4, el algoritmo de unificación se basa en la solución de un sistema de multiecuaciones. El sistema R de multiecuaciones está constituido por dos partes: T y U (figura 5.1.a), donde la parte T es un apuntador a una lista de multiecuaciones y la parte U es un apuntador a una estructura que contiene la parte no resuelta del sistema.

La parte U de un sistema de multiecuaciones (ver figura 5.1.b) se conforma de tres elementos: meqnumber contiene el número de multiecuaciones que están en la parte U, zerocountermeq, es un apuntador a una lista de multiecuaciones cuyo contador asociado es cero, y equations contiene las multiecuaciones con contador diferente de cero. Inicialmente en la lista apuntada por zerocountermeq se tienen los términos a unificar. En equations al iniciar el sistema, están las multiecuaciones generadas por todas las variables que aparecen en los términos a unificar.

Como se muestra en la figura 5.2.a, la implantación de una multiecuación consta de cuatro elementos: counter, contiene el número de veces que aparecen las variables del lado izquierdo de la multiecuación en otras multiecuaciones; varnumber, almacena el número de variables que están en la parte S de la multiecuación; S es el apuntador a la lista de variables que pertenecen a la parte S, y M apunta al multitérmino que constituye el lado derecho de la multiecuación.

Inicialmente todas las multiecuaciones contienen en su parte S sólo una variable, por

consiguiente varnumber es igual a uno y conforme se efectúa el proceso de compactación puede ser que varnumber se incremente.



a. Representación de un sistema para unificación.

b. Estructura de la parte U de un sistema de multiecuaciones.

Figura 5.1: Representación interna de un sistema de multiecuaciones.

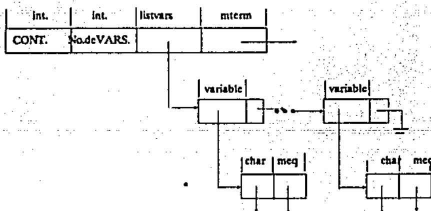
Las variables de la parte S de una multiecuación se representan por medio de la estructura de la figura 5.2.b, donde namevar es el apuntador a la variable y M es un apuntador a la multiecuación que contiene esta variable en su parte S.

```
struct meq
  {
    int counter;          /* Contador */
    int  varnumber;        /* Número de variables en su lado izquierdo */
    struct listvars *S ;  /* Lista de variables */
    struct mterm *M;  /* Multitérmino */
  }; /* Multiecuación */
```

a. Estructura para las multiecuaciones.

```
struct variable
  {
    char *namevar;    /* Apuntador al nombre */
    struct meq *M;   /* Multiecuación que la contiene */
  }; /* Variables */
```

b. Representación interna de las variables en la estructura.

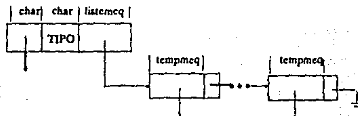


c. Representación gráfica de las multiecuaciones.

Figura 5.2. Representación interna de las multiecuaciones.

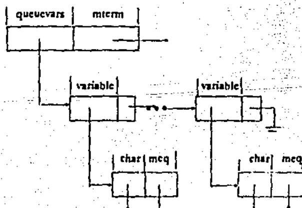
La estructura que representa un multitérmino consiste de un apuntador al nombre del multitérmino, el tipo y los argumentos del multitérmino (figura 5.3.a). A través de fsymb se apunta al nombre; debido a que empleando la misma estructura de multitérmino se representa una función, un predicado o una constante, en ftype se almacena el tipo del multitérmino; los argumentos de un multitérmino son pares, $< S_i, M_i >$ $(i = 1,...,n)$, como se describió anteriormente, los cuales son apuntados por args; cada par $< S_i, M_i >$ se representa por medio de una multiecuación especial, denominada multiecuación temporal, cuya estructura se muestra en la figura 5.3.b.

```
struct mterm
  {
    char *fsymb;          /* Apuntador al nombre del multitérmino */
    char ftype;           /* Tipo de multitérmino */
    struct listtemeq *args ; /* Lista de argumentos */
  }; /* Multitérmino */
```



a. Estructura de los multitérminos.

```
struct tempmeq
  {
    struct queuevars *S;   /* Lista de variables */
    struct mterm *M;       /* Multitérmino */
  }; /* Multiecuación Temporal */
```



b. Representación interna de las multiecuaciones temporales.

Figura 5.3. Representación interna de los multitérminos.

En la figura 5.4 se muestra la implantación codificada en lenguaje C, del algoritmo de unificación descrito en la sección 4.4., destacando las funciones Reduce y Compact, en las cuales se efectuan la reducción de multiecuaciones y la compactación de variables respectivamente, entregando como salida el sistema T de multiecuaciones conteniendo el unificador mas general.

```
struct listmeq *Unify(R,fail)
struct system *R;
short  *fail;
 {
    struct listtemeq *frontier;
    struct meq *mul;
  do
  {
     mul = Selmulteq(R->U);          /* Selecciona una multiec. */
    if ( mul->M != NULL )
    {
       frontier = NULL;                        /* Inicia la frontera */
       frontier = Reduce(mul->M,frontier);     /* Reduce */
       R->U = Compact (frontier, R->U, fail);  /* Compacta */
       if (*fail<= 0 )
       {
        errmsgmonit(1006);
       return(NULL);
       };
     }/* if */
    R->T = Newlstmeq(R->T,mul);            /* Pasa la multiec. a T */
  }
  while ( R->U->meqnumber != 0 );
  return ( R->T );
 }/* * * * Unify * * * */
```

Figura 5.4: Función de Unificación.

5.2 Representación interna de las reglas.

En esta sección se describen las tablas para almacenar los objetos del lenguaje, así como la representación interna de las reglas. La representación de las reglas se fundamenta en las estructuras de unificación. El contar con una estructura uniforme permite pasar los predicados directamente a unificación sin efectuar transformaciones.

Como se mencionó en la sección 4.2 que describe el lenguaje orientado a reglas, desde el punto de vista de la lógica el sistema tiene tres tipos de términos: funciones, variables y constantes, mientras que desde el punto de vista de las bases de datos deductivas cuenta con reglas de deducción, relaciones y restricciones de integridad semántica. Para cada uno de estos elementos del lenguaje el sistema asocia una tabla.

El sistema identifica los objetos del lenguaje por medio de la asignación de identificadores, tales identificadores se forman de un código consistente de un número entero positivo asignado de manera secuencial y de la tabla a la que pertenece el objeto, es decir, distintos tipos de objetos pueden tener el mismo código pero se distinguen por la tabla a la que pertenecen.

Los componentes de la tabla de funciones, como se muestra en la figura 5.5 son : el identificador asociado a la función (fnusername), el tipo (fntype) y el número de argumentos que contiene (fpars). En este proyecto se incluyen funciones con el propósito de verificar completamente el funcionamiento del algoritmo de unificación, además de ser necesarias para futuras ampliaciones que se realizarán a este trabajo.

```
struct functions
  { int fnsysname;                /* Código de la función */
    char fnusername[namesize];   /* Nombre del usuario*/
    int fntype;                  /* Tipo de función */
    int fpars;                   /* Número de argumentos */
  }; /* Tabla de Funciones */
```

int.	string	int	int.
ID. SISTEMA	NOMBRE DE FUNCION	TIPO	No. DE ARGS.

Figura 5.5: Representación interna de las Funciones.

La representación interna de las variables consta de dos elementos: varatrsysname almacena el código asignado a la variable, varatrusername es un arreglo de caracteres donde se guarda el nombre de la variable (ver Figura 5.6).

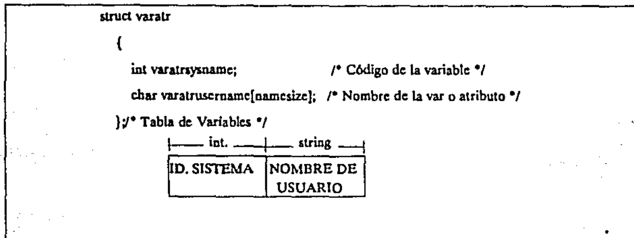


Figura 5.6: Representación interna de la tabla de variables.

Para almacenar las constantes existe una tabla principal cuya estructura se muestra en la figura 5.7.a. En esta tabla únicamente se almacena el código de la constante en ctesysname y el tipo de la constante en ctype. Debido a que el sistema manipula distintos tipos de constantes, se requieren distintas longitudes para almacenarlas, por lo que se emplean tablas independientes para cada tipo. La estructura de cada tabla contiene el código de la constante y un campo de valor. El campo de valor en cada tabla tiene el tipo asociado a la tabla.

La tabla que almacena las reglas de deducción consta de cinco componentes (figura 5.8.a.): el código asignado a la regla (rulesysid), el nombre que el usuario asocia a la regla (ruleusername), el nombre del consecuente (ruleconsname), el apuntador a la estructura interna de la regla (ruletree) y el número de argumentos del consecuente (rpars).

Como se describió en el capítulo anterior una regla consiste de un consecuente y un antecedente, para esta tesis el antecedente se representa en forma polaca como una lista de orden n, es decir, los operadores lógicos son de orden n, excepto el operador de negación el cual es de orden uno. Por lo anterior, la estructura empleada para representar una regla consiste de tres elementos: un apuntador a un predicado, un operador lógico y un apuntador a la lista de operandos del operador lógico, como se muestra en la figura 5.8.b, donde treemterm apunta al multitérmino que representa un predicado, treeop es el código del operador lógico y treelist es el apuntador a la lista de operandos.

Debido a que una relación R se puede representar como una regla cuyo antecedente es el vacío (R←), las relaciones también se almacenan en la tabla de reglas. Una diferencia de las relaciones con las reglas de deducción radica en que las relaciones se crean en la base de datos por lo que es necesario reconocer los atributos de cada relación. La estructura de la tabla de relaciones de la figura 5.9.a contiene dos campos: el código de la relación (rulesysid) y un apuntador a la lista de atributos (plstatr).

A diferencia de las variables, para los atributos de las relaciones es necesario definir el tipo debido a que se crean en la base de datos, por lo que además de almacenarse en la tabla de variables existe una tabla de atributos mostrada en la figura 5.9.b, donde se almacena el código del atributo en atrsysname, el tipo del atributo (string, caracter, real o entero) en atrtype, en atrsize1 se almacena una longitud del atributo y atrsize2 contiene una segunda longitud del atributo (se emplea para tipo real).

Como se mencionó al describir el lenguaje orientado a reglas, el consecuente de una restricción de integridad puede ser un predicado o puede ser nulo. La tabla donde se almacenan las restricciones de integridad (figura 5.10) consiste de: constsysname es el código que identifica la restricción, constusername contiene el nombre que el usuario asignó a la restricción, constconsname almacena el nombre del consecuente, en caso de que exista, constmterm es el apuntador a la estructura que representa la restricción y cpars contiene el número de atributos del consecuente. Esta estructura se puede alterar con próximas ampliaciones del sistema dado que actualmente no se procesan restricciones de integridad.

Ejemplo: Considérese que se tiene la siguiente regla

R: Q(x,y)←P1(x, f(z), z) AND P2(z, y, "A")

donde esta cláusula es una cláusula de rango restringido debido a que x,y aparecen en el antecedente; los nombres de los predicados, variables, funciones y constantes que aparecen en la regla se almacenan en las tablas respectivas, la estructura que almacena la regla consiste de un apuntador al multitérmino que representa el consecuente (Q(x,y)), el operador lógico AND y un apuntador a una lista de orden dos, en la cual el primer elemento de la lista representa al predicado P1(x,f(z),z) y el segundo elemento de la lista representa a P2(x,y,"A"). Como se observa en la figura 5.11 la estructura de multitérminos empleada permite almacenar de la misma forma los predicados, las funciones y las constantes.

```
struct constants
  { int ctesysname;            /* Código de la constante */
    char ctetype;              /* Tipo de constante */
  };/* tabla de Constantes */
  |---- int. ----|-- char --|
  | ID. SISTEMA  |   TIPO   |
```

a. Estructura de la tabla general de constantes.

```
struct ctestring
  { int  ctestrsysname;         /* Código de la constante string */
    char *ctestrvalue;          /* Valor*/
  };/* Tabla de Constantes Tipo "String" */
  |---- int. ----|-- char --|
  | ID. SISTEMA  |    •-----|---•
```

b. Estructura de la tabla para constantes caracteres.

```
struct cteinteger
  { int cteintsysname;          /* Código de la constante entera */
    int cteintvalue;            /* Valor */
  }; /* Tabla de Constantes tipo entero */
  |---- int. ----|-- int. --|
  | ID. SISTEMA  |  VALOR   |
```

c. Estructura de la tabla para constantes enteras.

```
struct ctereal
  { int cterealsysname;         /* Código de la constante real */
    float cterealvalue;         /* Valor */
  }; /* Tabla de Constantes tipo real */
  |---- int. ----|-- float --|
  | ID. SISTEMA  |  VALOR    |
```

d. Estructura de la tabla para constantes reales.

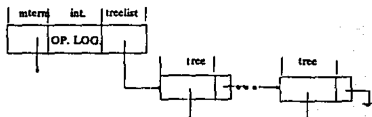
Figura 5.7: Representación interna de las constantes

```
struct rules
  { int rulesysid;                          /* Código de la regla */
    char ruleusername[namesize];            /* Nombre del usuario */
    char ruleconsname[namesize];            /* Nombre del consecuente */
    struct tree *ruletree;                  /* Apuntador a la estructura de la regla */
    int rpars;                              /* Número de argumentos */
  };/* Tabla de Reglas */
```

int.	string	string	tree	int.
ID. SISTEMA	NOMBRE REGLA	CONSECUENTE		No. DE ARGS.

a. Estructura de la tabla de reglas.

```
struct tree
  { struct mterm *treemterm;                /* Apuntador a un predicado */
    int treeop;                             /* Operador lógico */
    struct listree *treelist;               /* Operandos del operador lógico */
  };/* Estructura de una regla */
```



b. Estructura de una regla.

Figura 5.8: Representación interna de las reglas.

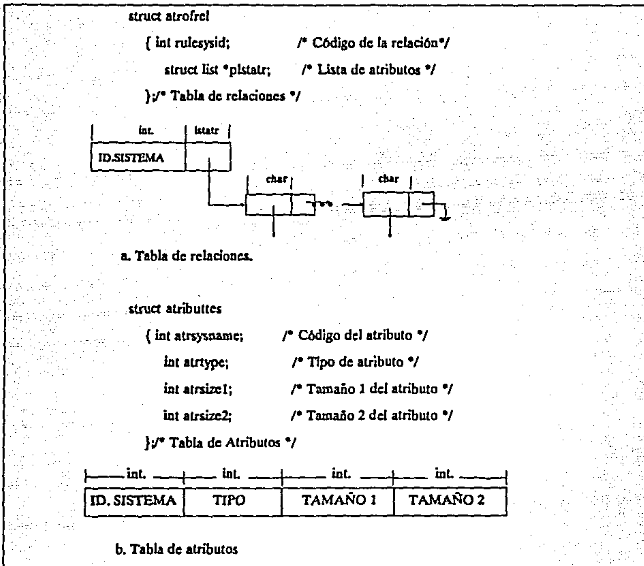


Figura 5.9: Representación interna de las relaciones.

```
struct constraints
  {
    int constsysname;                  /* Código de la restricción */
    char constusername[namesize];  /* Nombre del usuario */
    char constconsname[namesize];  /* Nombre del consecuente */
    struct tree *constmterm;           /* Apuntador a la estructura */
    int cpars;                          /* Número de argumentos */
  };/* Tabla de Restricciones de Integridad */
```

int.	string	string	tree	int.
ID. SISTEMA	NOMBRE RESTRICCION	CONSECUENTE		No. DE ARGS.

Figura 5.10: Representación interna de las Restricciones de Integridad Semántica.

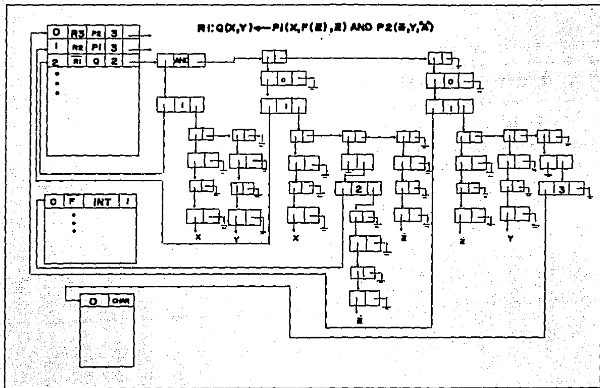


Figura 5.11 Representación interna de una regla.

5.3 Implantación del Parser

El parser efectúa tanto el análisis léxico y el análisis sintáctico del programa que recibe el sistema, como la generación de las estructuras internas que representan al programa.

El análisis léxico consiste en asignar un identificador a cada "token" del programa fuente si es un símbolo válido, y en caso contrario marca el error. El reconocedor de los "tokens" del lenguaje está almacenado en una tabla de transiciones donde por cada estado se tienen los caracteres que puede recibir dicho estado y los estados de transición correspondientes. En el apéndice F se describe la tabla de transiciones.

·Algoritmo del Analizador Léxico.

1. Inicio.
2. Mientras no se reconozca el token y no haya error
   2.1 Lee carácter.
   2.2 Verifica que el carácter corresponde a una transición.
   2.3 Si se encontró
         efectúa la transición
       en caso contrario
         marca error y retorna el código de error.
3. Retorna el token reconocido y su identificador.

El analizador sintáctico se encarga de reconocer sentencias válidas en la gramática de nuestro lenguaje, y genera las estructuras internas. El analizador sintáctico llama al analizador léxico para reconocer cada token , el analizador léxico retorna el identificador del token o un código de error en caso de ser un token inválido. Una vez reconocido el token se almacena en las estructuras de datos correspondientes.

El parser es de dos pasos con el objeto de verificar que cada predicado esté definido, ya sea en forma extensional o en forma intensional. En el primer paso, el parser analiza las relaciones y los consecuentes de las reglas. Las relaciones se almacenan como reglas con el antecedente vacío. Los nombres de los atributos de las relaciones se almacenan en la tabla de variables y el identificador del atributo, el tipo y longitud se almacenan en la tabla de atributos. Los

consecuentes de las reglas se almacenan en la tabla de reglas. En el segundo paso se efectúa el análisis del antecedente de cada regla. Para cada átomo del antecedente se verifica que exista, y se apunta a la regla donde aparece el átomo como consecuente de una regla. Las variables se almacenan en la tabla de variables.

Si existen dos o más reglas con el mismo predicado como consecuente, se dice que el sistema de reglas no está normalizado. Para los predicados de una regla no se almacena su nombre, existe un apuntador a su consecuente. Si existen reglas no normalizadas, el predicado apuntará a la primera regla donde aparece.

5.4. Generación del árbol de inferencia.

En la implantación del proceso de generación del árbol de inferencia se emplea el método compilado de la deducción[GaMn84] y el árbol se construye de arriba hacia abajo.

Una consulta puede estar constituida por uno o más predicados conectados por los operadores lógicos AND, OR y NOT. Dada una consulta se procesa por el parser para transformarla a notación polaca, representada por una lista de operadores y predicados. Los nodos se expanden tomando el primer elemento de la lista.

Para expander un átomo se lleva a cabo la unificación de éste con el consecuente de la regla correspondiente. La unificación puede tener éxito o fallar, si es exitosa se expande el átomo, si falla se tienen que tomar acciones dependiendo si el átomo es descendiente de un nodo AND , de un nodo OR o de un nodo NOT.

Si el nodo donde se produce la falla es descendiente de un nodo AND , ésto significa que el nodo AND también falla.

Si ocurre falla en un nodo descendiente de un nodo OR, se elimina el nodo que falló. Después de eliminar el nodo que falló se pueden tener los siguientes casos: si el nodo OR queda con exactamente un hijo, se elimina también el nodo OR y el nodo restante se agrega al padre del nodo OR, si el nodo OR queda con exactamente un hijo y el nodo OR es la raíz del árbol, se elimina el nodo OR y el nodo restante queda como raíz del árbol, y en caso de que el nodo OR quede con dos o más hijos no se realizan acciones adicionales.

Si falla un nodo descendiente de un nodo NOT, se interpreta que el nodo NOT evalúa a verdadero y por lo tanto se elimina el nodo NOT. En caso de que después de eliminar el nodo NOT, éste era hijo de un nodo AND y el nodo AND queda con un sólo hijo, se elimina el nodo AND y el nodo restante se agrega al nodo padre del nodo AND.

En la generación del árbol se emplean las mismas estructuras de datos utilizadas para representar a las reglas. Cuando se efectúa la expansión de un nodo, se realiza una copia de la regla correspondiente, si la unificación es exitosa, con el objeto de no alterar el sistema de reglas original.

Algoritmo para la Generación del arbol: Genera_árbol

1. Sea C una lista de átomos y operadores que representa una meta o submeta.
   Sea P el primer átomo de la lista.
   Sea R la regla donde el consecuente podría unificar con P.
2. Mientras C ≠ NULO
   2.1 Duplica la regla R.
   2.2 Crea la estructura inicial para unificación.
   2.3 Unifica P con el consecuente de R.
   2.4 Si no falla
      Efectua sustitución de variables.
      en caso contrario
      Propaga falla y termina
   2.5 Expande el nodo
   2.6 Elimina P de la lista C.
   2.7 Genera_árbol.
3. Termina y devuelve el apuntador a la raíz del árbol generado.

5.5. Implantación de la Sustitución.

El proceso de sustitución utiliza las multiecuaciones de la parte T construidas durante la unificación, las cuales representan al unificador más general. Los casos que se pueden presentar al efectuar la sustitución de variables en una regla son: la sustitución de variables por variables y la sustitución de variables por símbolos de constantes o símbolos de función.

Como se describió anteriormente una multiecuación consiste de una parte S y una parte M, por lo que en el caso de tener sustitución de variables por variables la parte M es nula, mientras que en el caso de sustitución de variables por símbolos de constante o símbolos de funcion se tiene en S las variables a ser sustituidas y en M el símbolo de constante o el símbolo de función por el cual se van a sustituir.

Para efectuar la sustitución separamos las variables de la parte S de cada multiecuación en dos listas de variables V_p y V_q. A V_q pertenecen las variables que aparecen en el consecuente de la regla con que se unificó y en V_p están las variables del nodo a expander. Para la sustitución de variables por variables se sustituye cada variable de V_q por la primera variable que aparece en V_p, si V_p tiene más de una variable, de la segunda variable en adelante se sustituyen por la primera variable de V_p, y si el nodo a expander tiene más de un hermano, de la segunda variable en adelante de V_p se sustituye por la primera variable de V_p en los hermanos del nodo a expander. En el caso de sustitución de variables por símbolos de constante o símbolos de función, se sustituyen las variables de V_q por el símbolo de constante o símbolo de función, y si V_p tiene más de una variable, se sustituyen las variables de V_p por el símbolo de constante o símbolo de función en los predicados hermanos del nodo a expander.

Algoritmo de Sustitución.

1. Sea T el unificador más general del nodo a expander.
2. Mientras T $\neq$ NULO
   2.1 Selecciona la primera multiecuación S = M de T.
   2.2 Si longitud (Si) < 1 termina con error.
   2.3 Genera dos listas de variables V_p y V_q:
   $V_p = \{v | v$ es una variable del predicado a expander$\}$
   $V_q = \{v | v$ es una variable de la regla con que se unificó$\}$
   2.4 Si M es nulo
      2.4.1 Sustituye cada variable de V_q por la primera variable que aparece en V_p
      2.4.2 Si longitud(V_p) > 1
      En los predicados que aparecen como hermanos del predicado a expander se sustituyen las variables restantes de V_p por la primera variable de V_p.
   2.5 Si M no es nulo
      2.5.1 Sustituye cada variable de V_q por la constante o función
      2.5.2 Si longitud(V_p) > 1

En los predicados que aparecen como hermanos del predicado a expander se sustituyen las variables de V_p por el símbolo de constante o función.

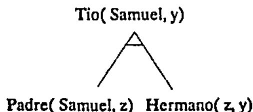
2.6 Elimina la primera multiecuación de T.

3. Termina

5.6. Evaluación del árbol de inferencia.

El método de evaluación empleado se basa en transformar el árbol de inferencia generado a una expresión equivalente del álgebra relacional. La expresión del álgebra relacional se construye por medio de la creación de vistas tanto para cada nodo intermedio del árbol como para la raíz, y una vez generadas las vistas se realiza la operación de selección sobre el nodo raíz.

Un nodo intermedio representa una submeta. Los nodos AND se representan por una vista incluyendo en la cláusula **FROM** los predicados hijos de dicho nodo, en la cláusula **WHERE** las condiciones de restricción y "join", y para la cláusula **SELECT** se recurre a las reglas originales con el objeto de consultar las relaciones base de acuerdo a los nombres originales de los atributos de la relación. Por ejemplo, considérese el siguiente subárbol:



el cual representa una submeta que involucra a la regla

$$Tio(x, y) \leftarrow Padre(x, z) \wedge Hermano(z, y)$$

la cual representa que el Tio de x es y. Considérese además que Padre y Hermano pertenecen a la base de datos extensional y están definidas como:

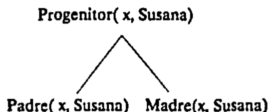
Padre(u, v)
Hermano(w, r)

la expresión equivalente en el álgebra relacional para el subárbol anterior es un join dado que el subárbol es un nodo AND y los predicados tienen variables en común. El nombre de la vista es Tio con atributos x,y, los cuales aparecen como atributos en la regla original. En la cláusula FROM se incluyen las relaciones Padre y Hermano sobre las cuales se efectua el join. Debido a que existe una constante en el subárbol la condición de restricción de la cláusula WHERE se obtiene buscando el nombre del primer atributo de Padre en la regla original, es decir, x = "Samuel", y para la condición del join se selecciona el segundo atributo de Padre, y el primer atributo de Hermano, quedando v = w. La expresión equivalente obtenida para evaluar el subárbol es:

```
CREATE VIEW Tio(x,y) AS
        SELECT x,y
        FROM Padre,Hermano
        WHERE x = "Samuel" AND
                v = w
```

Puede darse el caso de que algunos nodos tengan el mismo nombre del predicado, lo cual ocasionaría conflictos debido a que no pueden existir dos vistas con el mismo nombre. Para resolver este problema el sistema asigna un nombre temporal distinto a cada predicado duplicado. El control de las vistas creadas en la base de datos que lleva el sistema permite que al final de la evaluación del árbol se eliminen las vistas de la base de datos.

Los nodos OR se representan por medio de una vista formada por la unión de los bloques de consulta de cada uno de sus descendientes. Para cada bloque de consulta la cláusula **SELECT** contiene los nombres de las variables que aparecen en el predicado de acuerdo a como se denominan en la regla original, en la clausula **FROM** se declara el nombre del predicado y en la cláusula **WHERE** se indican las condiciones de restricción. Por ejemplo considérese el siguiente subárbol:



el cual involucra la siguiente regla:

Progenitor(x , y) ← Padre(x, y) ∨ Madre(x, y)

considérese además que Padre y Madre pertenecen a la base de datos extensional y están definidas por:
Padre(u,v)
Madre(w,z)

para obtener la expresión que representa este subárbol, se crea una vista con nombre Progenitor y atributos x,y, los cuales aparecen en la regla original. Para el primer nodo descendiente se construye la consulta

```
SELECT u,v
FROM Padre
WHERE y = "Susana"
```

donde u y v son los nombres de los atributos en la regla original, y para el segundo descendiente se obtiene:

```
SELECT w,z
FROM Madre
WHERE y = "Susana"
```

en ambos casos se tiene la condición de restricción y = "Susana", dado que el segundo atributo de cada átomo es un símbolo de constante. La expresión en SQL para el subárbol es:

```
CREATE VIEW Progenitor(x,y) AS
        SELECT u,v
        FROM Padre
        WHERE y = "Susana"
        UNION
        SELECT w,z
        FROM Madre
        WHERE y = "Susana"
```

5.7. Interfase con el manejador de base de datos.

La interfaz con el manejador de bases de datos es empleada para ejecutar las siguientes acciones:

i) Creación y borrado de relaciones.
ii) Creación y borrado de vistas.
iii) Consultas.
iv) Inserción y borrado de tuplos.
v) Para conectarse y desconectarse de la base de datos.

Desde un programa escrito en el lenguaje C se pueden ejecutar instrucciones de SQL, para lo cual ORACLE proporciona cuatro métodos los cuales difieren en el tipo de operaciones que se pueden ejecutar. El pre-compilador para C de ORACLE se denomina PRO*C. En el apéndice G se describe dicho precompilador. Los cuatro métodos para ejecutar cláusulas de SQL desde un programa en C son:

1) Ejecución Inmediata (Execute Immediate).

Ejecuta cualquier cláusula de SQL, excepto una consulta, sólo una vez y es estática. Cada vez que se desee ejecutar la sentencia de SQL se precompila y se ejecuta.

2) Preparación y Ejecución (Prepare and Execute).

En este método el enunciado de SQL se analiza(prepara) y se puede ejecutar varias veces, por lo cual las sentencias de SQL pueden contener variables que permitan ejecutar la misma sentencia para distintos valores de la variable, por ejemplo:

```
DELETE FROM EMP WHERE EMPID = :EMPNO
```

donde EMPNO es una variable. La limitación de este método radica en que el número de parámetros y sus tipos se deben conocer de antemano debido a que las variables que se emplean deben declararse previamente, y no permite consultas.

3) Prepara y busca (Prepare and Fetch).

Permite el uso de consultas pre-programadas, es decir, consultas cuya lista de atributos en la cláusula **SELECT** es conocida previamente.

4) Consultas dinámicas.

Un enunciado de SQL definido dinámicamente es una sentencia de SQL la cual no se conoce a tiempo de compilación y puede cambiar de ejecución a ejecución, por lo que el número y tipo de las variables no está predefinido por el programador.

De estos cuatro métodos se emplean dos en el sistema, dado que el método de ejecución inmediata es suficiente para la creación y borrado de relaciones, creación y borrado de vistas, inserción y borrado de tuplos, conectarse y desconectarse de la base de datos. Debido a que se manipulan consultas dinámicas se emplea el método cuatro para ejecutarlas. Utilizar el método tres implicaría tener todas las consultas posibles precompiladas, lo cual es impráctico por la explosión combinatoria en tiempo y espacio.

Capítulo 6.

EVALUACIÓN DE RESULTADOS

6.1 Comparación entre las bases de datos deductivas y las bases de datos tradicionales.

La forma clásica de los sistemas de base de datos fue diseñada para manejar una clase de aplicaciones importante pero limitada. Estas aplicaciones generalmente manipulan grandes volúmenes de datos, pero las operaciones realizadas sobre los datos son simples. Las operaciones predominantes son la inserción, borrado y extracción de tuplos, siendo la navegación entre un pequeño número de relaciones o archivos una de las cosas más complejas que se pudiesen efectuar.

Es necesario hacer notar las diferencias entre el DML y el lenguaje anfitrión ("host"). El DML cuenta con capacidades intrínsecas para accesar la base de datos eficientemente, pero el poder expresivo del DML es muy restringido. El lenguaje anfitrión es un lenguaje de propósito general. La dicotomía entre el DML y el lenguaje anfitrión generalmente se considera una ventaja en lugar de una desventaja de los sistemas de bases de datos. El DML permite optimizar consultas transformando los algoritmos que las expresan de una forma sorpresiva pero correcta. Las mismas consultas escritas en un lenguaje de propósito general no se podrian optimizar de igual manera.

Algunas aplicaciones clásicas donde es necesario combinar las capacidades del DML y del lenguaje anfitrión son: diseño de bases de datos de CAD, bases de datos gráficas y bases de datos de ingeniería de software, esto es, bases de datos que manipulan versiones múltiples de programas muy grandes. Estas aplicaciones se caracterizan por un acceso rápido y constante modificación de los datos, así como por la necesidad de incluir operaciones más poderosas sobre los datos.

Una observación relevante consiste en que los sistemas de bases de datos convencionales no permiten la definición de consultas recursivas, lo cual hace necesario emplear lenguajes anfitriones para realizar tales tareas.

Existen dos métodos para crear un lenguaje uniforme que integre las capacidades del DML para la manipulación de datos con las capacidades del lenguaje anfitrión:

1. El paradigma orientado a objetos, el cual emplea un lenguaje con capacidad de definir tipos de datos abstractos, o clases. El sistema permite al usuario utilizar estructuras de datos para accesos más rápidos.

2. El paradigma lógico, el cual emplea un lenguaje similar a las reglas lógicas de la forma si... entonces. Algunos predicados se consideran parte del esquema conceptual, mientras que otros se utilizan para definir vistas, o bien, pueden formar parte de un programa de aplicación.

Un lenguaje declarativo es un lenguaje en el cual se puede expresar lo que uno desea obtener sin especificar como lograr el resultado. Un lenguaje que no es declarativo es un lenguaje procedural. Los lenguajes declarativos suelen ser mas difíciles de implantar que los lenguajes procedurales debido a que se requiere optimizaciones extensivas si se desea una implantación eficiente.

6.2 Estrategias de diseño e implantación.

Para el diseño e implantación del sistema se emplearon algunos principios básicos de programación como son: modularidad, portabilidad y eficiencia, además de facilidad de reemplazar el RDBMS utilizado, verificación en la definición de predicados y uniformidad en las estructuras de datos.

Modularidad.

La elaboración de este trabajo consistió de una técnica de desarrollo incremental, lo cual fue factible debido a que se encuentra organizado en módulos lógicos. El sistema consiste de cuatro módulos principales: monitor, parser, lenguaje orientado a reglas e intérprete.

Portabilidad.

La portabilidad se refiere al hecho de hacer el sistema lo menos dependiente del equipo en el que fue implantado, siendo posible su implantación en otros equipos. En este trabajo de tesis se evitó el empleo de funciones no estándares del lenguaje de programación.

Eficiencia.

El objetivo de este primer prototipo no era obtener un sistema que fuera eficiente en el tiempo de respuesta al usuario final, sin embargo, si se planteó que el sistema esté programado de manera eficiente para que en ampliaciones posteriores se obtenga un sistema con un tiempo de respuesta adecuado, por lo cual, en todos los algoritmos programados se buscó que estos fueran lo mas eficientes posibles. Una excepción a ésto la constituye la estrategia de evaluación, la cual es relativamente pequeña y su sustitución por un método de evaluación más eficiente es relativamente simple.

Facilidad de reemplazar el RDBMS utilizado.

Con el objeto de poder cambiar el RDBMS se diseñó un módulo de interfaz con el manejador de bases de datos, el cual se puede reemplazar por un módulo que sea la interfaz de otro RDBMS, cabe aclarar que dado que se emplea SQL, el otro RDBMS también deberá utilizar SQL. Si existen diferencias en el dialecto empleado en el otro RDBMS habría que realizarlas en el dialecto actual.

Verificación en la definición de los predicados.

Otro criterio es la verificación de que todo predicado esté definido en forma intensional o en forma extensional, para lo cual se diseñó el parser en dos pasadas.

Uniformidad en las estructuras de datos.

Una estrategia de diseño fue contar con estructuras de datos uniformes para la representación de las reglas, generación del árbol de inferencia y unificación con el objeto de no tener que efectuar transformaciones.

6.3 Características del software empleado.

Debido a que el objetivo de este prototipo es desarrollar un sistema deductivo factible de utilizarse rápidamente, el sistema no se programó desde abajo, es decir, se utilizó una base de datos tradicional sobre la cual se pueda montar el sistema, lo cual nos lleva a contar con dos elementos de software para desarrollar la base de datos deductiva: un manejador de bases de datos relacional y un lenguaje de propósito general. Los criterios para la selección de este software son la eficiencia, la portabilidad del sistema y los estándares industriales.

El lenguaje de programación empleado es C debido a que es un lenguaje intermedio entre alto y bajo nivel, cuenta con las estructuras de control más relevantes para programar estructuradamente,y es portable, sin embargo, cada compilador tiene a su vez un conjunto de funciones fuera de los estándares. Los programas escritos en lenguaje C se basan en funciones lo cual permite obtener un sistema modular.

El manejador relacional de bases de datos seleccionado debe corresponder a los estándares, ser portable, relativamente eficiente, así como debe estar lo más apegado posible al álgebra relacional. SQL es el estandar más aceptado para bases de datos relacionales. Dado que ORACLE utiliza SQL y es uno de los RDBMS más eficientes actualmente en el mercado, se decidió utilizar este manejador de bases de datos.

Se consideró que el RDBMS pudiera tener como lenguaje anfitrión a C, lo cual satisface el RDBMS de ORACLE. Como se mencionó anteriormente PRO*C tiene cuatro métodos para ejecutar instrucciones de SQL desde un programa en C. Para esta tesis se emplearon el método de ejecución inmediata y el método de consultas dinámicas.

6.4 Ampliaciones al sistema.

El objetivo marcado para este trabajo fue desarrollar una primera versión que no incluyera todos los aspectos que comprende una base de datos deductiva, dado que ésto implicaría varios años hombre de trabajo y se consideró que con fines de la tesis era suficiente el desarrollo de un primer prototipo limitado que pudiera ampliarse con otros trabajos, es decir, de la arquitectura mostrada en la figura 1.2 falta por desarrollar e implantar los siguientes módulos: el módulo explicativo, el mecanismo de actualización, la optimización de reglas y el manejo de restricciones de integridad semántica.

ESTA TESIS NO DEBE
SALIR DE LA BIBLIOTECA

La arquitectura del sistema desarrollado está incompleta dado que el intérprete maneja exclusivamente reglas normalizadas, por lo que es necesario implantar la normalización automática, así como el manejo de reglas recursivas. Para llevar a cabo la normalización automática se requiere construir un módulo que tome como entrada el sistema de reglas no normalizado y genere el sistema normalizado, así este último puede ser manipulable por el sistema actual.

El sistema emplea tanto instrucciones del lenguaje de definición de datos como instrucciones del lenguaje de manipulación de datos, lo cual degrada el sistema. Esta es una ineficiencia del sistema dado que se generan vistas para representar los nodos intermedios y la raiz del árbol, después se genera una expresión de consulta sobre la raiz, por lo que es necesario la implantación de un método de evaluación que no genere vistas para los nodos intermedios.

Con el propósito de prueba del sistema se desarrolló una pequeña interfaz con el usuario la cual es muy primitiva, por lo que es conveniente implantar una interfaz con el usuario más amigable y poderosa.

Otra fuente de ineficiencia en una base de datos deductiva es que los hechos se encuentran almacenados en disco y los accesos a disco son generalmente más lentos que los accesos a memoria principal, por tal motivo para realizar busquedas más rápidamente es recomendable contar con índices. Esto implica tanto la modificación del lenguaje deductivo como el desarrollo y programación de un módulo para la creación y borrado de índices.

6.5 Conclusiones.

Aprovechando el formalismo de los esquemas lógicos, con las ventajas que las bases de datos brindan para manipular grandes volúmenes de datos, se realizó el diseño e implantación de una base de datos deductiva para la programación de sistemas expertos. La arquitectura del sistema constituye una extensión de un sistema manejador de bases de datos relacional para soportar la deducción.

Estado actual del sistema.

Actualmente se cuenta con una primera versión del sistema, la cual se encuentra disponible para su utilización, aunque tiene las limitaciones descritas en la sección anterior. Es importante

señalar que la primera parte que se programó fue el parser, por lo cual se cometieron algunos errores de diseño, aunque éste funciona actualmente, consideramos que sería conveniente la reprogramación de este módulo con el objeto de corregir los errores cometidos.

Portabilidad del sistema.

La implantación del sistema fue lo más portable posible del equipo donde se desarrolló, evitando el uso de funciones ajenas a los estándares. La implantación se realizó en el lenguaje de programación C [KeRi79], y se emplea el manejador relacional de bases de datos ORACLE.

Eficiencia.

En la implantación del sistema la manipulación de reglas no efectúa búsquedas cada vez que se tiene que unificar. El algoritmo de unificación seleccionado es un algoritmo casi lineal muy eficiente. Para incrementar la eficiencia del proceso de evaluación del árbol de inferencia es necesario considerar la creación de índices, efectuar transformaciones en el árbol y realizar heurísticos de las relaciones.

APENDICE A.

DESCRIPCIÓN EN BNF DE LA GRAMATICA DEL LENGUAJE.

<DDB> ::= [<Functions>] <Relations> [<Rules>] [<Constraints>]

<Functions> ::= FUNCTIONS_DEF <Functions_def>
<Functions_def> ::= <Function_def> | <Function_def> <Functions_def>
<Function_def> ::= [<fn_type>] <fn_declarator> <fn_body>
<fn_declarator> ::= <fn_name> ([<parameters>])
<parameters> ::= <identifier> | <identifier> , <parameters>
<fn_body> ::= <declars_list> { <fn_statements> }
<declars_list> ::= <declaration> | <declaration> ; <declars_list>
<fn_statements> ::= [<declars_list>] <statement_list>

<Relations> ::= RELATIONS_DEF <Relations_def>
<Relations_def> ::= <Relation_def> | <Relation_def> <Relations_def>
<Relation_def>::= RELATION <Relation_name>(<Atributes>)
[DOC(<Comment>)]
<Relation_name> ::= <Identifier>
<Atributes> ::= <Atribute> | <Atribute> ; <Atributes>
<Atribute> ::= <Atr_name> : <Atr_type> [:<Atr_size1>] [:<Atr_size2>]
[DOC (<Comment>)]
<Atr_name> ::= <Identifier>
<Identifier> ::= <letter> | <letter> <String>
<Atr_type> ::= STRING | CHAR | INTEGER | REAL | LONG | DOUBLE

```
<Atr_size1> ::= <Number>
<Atr_size2> ::= <Number>
<String> ::= <letter> | <digit> | - |_| <letter> <String> | <digit> <String> |
             - <String> | <String>
<Comment> ::= <print_char> | <print_char> < Comment>
<letter> ::= a|b|...|y|z|A|B|...|Y|Z
<digit> ::= 0| 1| 2|...8| 9

<Rules> ::= RULES_DEFINITION <Rules_def>
<Rules_def> ::= <Rule_def> | <Rule_def> <Rules_def>
<Rule_def> ::= RULE< Rule_name> : <Consequent> <- <Antecedent>
               [DOC(<Comment>)]
<Rule_name> ::= <String>

<Consequent> ::= <Predicate>
<Predicate> ::= <User_Pred> | <System_Pred>
<User_Pred> ::= <Predicate_name> ( <Terms_list> )
<Predicate_name> ::= <Identifier>
<Terms_list> ::= <Term> | <Term> , <Terms_list>
<Term> ::= <Constant> | <Variable> | <Function>
<Variable> ::= <letter> | <letter> <String>
<Constant> ::= "<String>" | <Numeric_cte>
<Numeric_cte> ::= <digit> | <digit> <Numeric_cte>
<Function> ::= <User_Function> | <System_Function>
<User_Function> ::= <Function_name> ( <Params_list> )
<Function_name> ::= <Identifier>
<Params_list> ::= <Variable> | < Function>
<System_Function> ::= <plus> | <minus> | <multiplication> | <division> |
                      <module>
<plus> ::= PLUS ( <Term>,<Term> ) | <Term> + <Term>
<minus> ::= MINUS ( <Term>,<Term> ) | <Term> - <Term>
<multiplication> ::= MUL ( <Term>, <Term> ) | <Term> * <Term>
```

```
<division> ::= DIV ( <Term>, <Term> ) | <Term> / <Term>
<module> ::= MOD ( <Term>, <Term> ) | <Term> % <Term>
<System_Pred> ::= <less> | <less_equal> | <greather> | <greather_equal> |
                    <equal>
<less> ::= LT( <Term>, <Term> ) | <Term> < <Term>
<less_equal> ::= LEQ ( <Term> , <Term> ) | <Term> <= <Term>
<greather> ::= GT ( <Term> , <Term> ) | <Term> > <Term>
<greather_equal> ::= GEQ( <Term> , <Term>) | <Term> >= <Term>
<equal> ::= EQ( <Term>,<Term> ) | <Term> = <Term>

<Antecedent> ::= <Literals> | <Lit_Ant>
<Literals> ::= <Literal> | <Literal> <OR> <Literal> | <Literal> <AND>
              <Literal>
<Lit_Ant> ::= <Ant> | <Literal> <OR> <Ant> | <Ant> <OR> <Literal> |
                        <Literal> <AND> <Ant> | <Ant> <AND> <Literal>
<Ant> ::= <NOT> ( <Antecedent> ) | <Antecedent> | ( <Antecedent> )
<Literal> ::= <Predicado> | <NOT> <Predicado> | ( <Literal> )
<AND> ::= AND
<OR> ::= OR
<NOT> ::= NOT

<Constraints> ::= CONSTRAINTS_DEF <Const_definitions>
<Const_definitions> ::= <Const_definition> |
                        <Const_definition> <Const_definitions>
<Const_definition> ::= CONSTRAINT <Const_name> : [ <Consequent> ] <-
                        <Antecedent> [ DOC (< Comment> ) ]
<Const_name> ::= <String>
```

APENDICE B

PALABRAS RESERVADAS DEL SISTEMA.

B.1 Palabras Claves.

FUNCTIONS_DEF
RELATIONS_DEF
RELATION
CONSTRAINTS_DEF
CONSTRAINT
RULES_DEF
RULE
STRING
CHAR
INTEGER
REAL
LONG
DOUBLE
DOC
<-

B.2 Funciones Aritméticas

PLUS
MINUS
MUL
DIV
MOD

B.3 Predicados Aritméticos

<
< =
>
> =
=
LT
LEQ
GT
GEQ
EQ

B.4 Operadores Lógicos.

AND
OR
NOT

APENDICE C

MENSAJES DE ERROR

C.1 Mensajes de Error del Parser.

NAC-0001. Símbolo Indefinido.

El nombre del identificador no fue declarado, o bien se encontró algún símbolo no aceptado en las palabras claves del lenguaje (ver apéndice B). Este error se pudo generar al ocurrir un error previo en alguna definición.

NAC-0002. Se espera identificador de la función.

El identificador que se esperaba correspondía al nombre de alguna función. Pudo haber ocurrido un error previo en la definición de la función.

NAC-0003. La función ya había sido definida.

Dos funciones distintas tienen asignado el mismo nombre. Verificar los nombres de las funciones.

NAC-0004. Se espera paréntesis "(".

El compilador encontró una expresión sin un paréntesis que abre en el lugar adecuado. Verificar la sintaxis donde ocurrió el error.

NAC-0005. Se espera paréntesis ")".

El compilador encontró una expresión sin un paréntesis que cierra donde se esperaba. Posiblemente el error fue ocasionado por un error previo.

NAC-0006. Se espera el caracter "{".

El compilador encontró una expresión sin una llave que inicie un bloque en el lugar apropiado. Verificar la sintaxis de inicio de un bloque donde ocurrió el error.

NAC-0007. Se espera el caracter "}".

El compilador encontró una expresión sin una llave que cierre un bloque en el lugar adecuado. El cuerpo de una función es delimitado por el símbolo de llave. Verificar que cada llave que abre tenga asociada una llave que cierra.

NAC-0008. Un programa debe incluir la definición de relaciones.

Es un error fatal debido a que un programa debe contener las definiciones de las relaciones antes de la definiciones de las reglas (ver sección 4.3). Pudo ocurrir debido a que estas se encuentran fuera del orden correcto, o bien se pudo haber omitido la palabra clave RELATION_DEF, la cual indica inicio de la definición de relaciones.

NAC-0009. La definición de una relación debe iniciar con la palabra clave RELATION.

Dentro de la definición de relaciones se encontró posiblemente la declaración de una relación sin estar encabezada de la palabra reservada RELATION.

NAC-0010. La relación ya está definida.

Se encontraron dos relaciones con el mismo identificador. Verificar los nombres asociados a las relaciones. Verificar los nombres de las relaciones.

NAC-0011. Tipo de atributo inválido.

El tipo asociado a un atributo de una relación no es un tipo válido en el lenguaje. Los tipos válidos de los atributos son: STRING, CHAR, INTEGER, REAL, LONG o DOUBLE.

NAC-0012. Se espera el nombre del atributo.

En la definición de una relación no se encontró el identificador de un atributo. Posiblemente ocurrió debido a un error previo.

NAC-0013. Falta el caracter ":".

El compilador esperaba dentro de una expresión el separador dos puntos y encontró otro símbolo. Verificar la sintaxis de la expresión.

NAC-0014. Debe aparecer una constante númerica.

El compilador esperaba recibir un símbolo constante y encontró otro símbolo. Posiblemente fue ocasionado por un eror previo.

NAC-0015. Falta el caracter ";".

El compilador encontró una expresión sin el separador punto y coma adecuadamente. Verificar la sintaxis de la expresión.

NAC-0016. La definición de una regla comienza con la palabra clave RULE.

Dentro de la definición de reglas se encontró posiblemente la declaración de una regla sin estar precedida por la palabra clave RULE. Posiblemente es causa de algún error anterior.

NAC-0017. Debe aparecer el nombre de la regla.

El compilador esperaba un identificador como nombre de la regla y encontró algún otro símbolo.

NAC-0018. Debe aparecer el nombre del consecuente de la regla.

Se esperaba un identificador como nombre del consecuente de la regla . Posiblemente es ocasionado por un error previo.

NAC-0019. Se espera un identificador.

Se esperaba un identificador en la posición del error y se encontró otro símbolo. Un identificador debe comenzar con letra, el símbolo - o el símbolo _.

NAC-0020. La función no se ha definido.

Se hace referencia a una función la cual no fue definida en el bloque de funciones. Verificar los nombres de las funciones definidas.

NAC-0021. Debe aparecer el operador lógico <-.

Se detectó fin en la definición del consecuente de la regla y no se encontró el símbolo lógico <-. Verificar la sintaxis de la cláusula, en la cual debe aparecer el consecuente, el operador lógico <- y el antecedente.

NAC-0022. La instrucción debería tener en esta posición un predicado del sistema.

El símbolo encontrado no corresponde al identificador de algún predicado del sistema y encontró otro símbolo. Verificar la sintaxis de la expresión.

NAC-0023. El número de paréntesis que abren es distinto a los paréntesis que se cierran.

Al terminar la definición de relaciones, reglas o restricciones el compilador detectó la carencia de algunos paréntesis. Verificar que cada paréntesis que abre tiene asociado un paréntesis que cierra.

NAC-0024. Error en la sintaxis de la expresión.

Este mensage indica que el compilador detectó un error fatal y no es lógico el segmento de programa que se encontró. Posiblemente un error previo no se recuperó, o bien existe alguna incoherencia, por ejemplo: dos operadores sin operandos que los separe, una expresión incorrectamente expresada, paréntesis desbalanceados, etc.

NAC-0025. Una restricción debe iniciar con la palabra clave CONSTRAINT.

El compilador no encontró la palabra reservada CONSTRAINT durante la definición de restricciones. Una restricción de integridad se declara encabezada con la palabra clave CONSTRAINT, el nombre con que el usuario identifica la regla, el consecuente , en caso de existir, el operador lógico <- y el antecedente.

NAC-0026. Se espera el nombre asociado a la restricción.

El compilador espera un identificador que corresponda al nombre de la restricción de integridad. Posiblemente fue causado por un error previo. Verificar la sintaxis para declarar las restricciones de integridad.

C.2 Mensajes de Error del Monitor.

NAC-1000. La tabla de relaciones no se encuentra.

El intérprete no encontró el archivo que contiene la representación interna de las relaciones. El nombre del archivo donde se almacena la tabla de relaciones es el nombre del programa con la extensión T05.

NAC-1001. La tabla de reglas no se encuentra.

El intérprete no encontró el archivo que contiene la representación interna de las reglas. El nombre del archivo donde se almacena la representación interna de las reglas es el nombre del programa con la extensión T08.

NAC-1002. La tabla de predicados del sistema no se encuentra.

El intérprete no encontró el archivo que contiene los predicados del sistema. El sistema almacena los predicados del sistema en el archivo NACSYSPR.DAT. Verificar la instalación del sistema.

NAC-1003. El archivo con el nombre dado no se encuentra.

El sistema no puede abrir el archivo por que no aparece almacenado en el lugar donde es buscado. Verificar el nombre del archivo que indicó al sistema o si está posicionado en el lugar correcto..

NAC-1004. Durante el proceso de unificación ocurrió falla.

Durante el proceso de efectuar el "merge" de dos multiecuaciones se encontró inconsistencia en el proceso deductivo. Posiblemente la consulta introducida al sistema no es correcta.

NAC-1005. Durante la unificación dos términos no fueron consistentes.

Durante el proceso de efectuar el "merge" de dos multitérminos se encontró que dos términos no son consistentes y por lo tanto no pueden ser unificados. Verificar la consulta.

NAC-1006. El proceso de compactación falló.

El sistema encontró una incongrüencia al efectuar la transformación de compactación de dos términos durante el proceso deductivo. Posiblemente la consulta no es correcta.

NAC-1007. La evaluación de un predicado aritmético falló.

Las expresiones que aparecen como operandos de un predicado aritmético peden no ser correctas. Puede ocurrir como consecuencia de algún error previo.

NAC-1008. El proceso de unificación falló.

No fue posible obtener el unificador mas general para un par de términos durante el proceso deductivo. Verificar que los términos a unificar sean consistentes. Verificar la sintaxis de la consulta.

NAC-1009. La preparación de la consulta por medio de PRO*C falló.

No se puede preparar una consulta dinámica posiblemente por un error de sintaxis o por que no se han creado previamente las relaciones en las base de datos. Verificar la sintaxis de la consulta.

APENDICE D

ARCHIVOS DEL SISTEMA.

D.1 Programas Fuentes.

NACLIST.C	Módulo que contiene todas las funciones para manipulación de listas ligadas.
NACPARSE.C	Parser del lenguaje orientado a reglas, el cual incluye el submódulo del análisis léxico.
NACTREE.C	Submódulo del intérprete que contiene el proceso de generación del árbol de inferencia.
NACSQL.C	Submódulo del intérprete que contiene el proceso de evaluación del árbol de inferencia, el cual es transformado a una expresión equivalente del álgebra relacional.
NACUNIFY.C	Submódulo del intérprete que efectúa el proceso de unificación.
NACQUERY.C	Parser las consultas y las transforma a su representación interna.
NACMONI.C	Módulo monitor.
NACORAC.PC	Interfaz con el precompilador para C de ORACLE.

D.2 Archivos de Definición de datos del Sistema.

NACDEFTO.H	Constantes del sistema.
NACSE.H	Contiene los includes del sistema sin referencias externas.
NACSEANT.H	Contiene los includes del sistema con referencias externas.
NACPARSE.H	Variables del parser
NACUNIF.H	Variables y estructuras para unificación.
NACLIST.H	Variables y estructuras para el manejo de listas.
NACVARSP.H	Variables globales del sistema.
NACANTEC.H	Contiene las definiciones externas de las variables.

D.3 Archivos de Definición de datos del Sistema Operativo MS-DOS.

STDIO.H	Librerias estándares de C.
STDLIB.H	Librerias adicionales de C.
CTYPE.H	Librerias adicionales para manipular caracteres.
STRING.H	Definición de funciones para manejo de cadenas.

D.4 Archivos de Definición de datos Empleados por el precompilador PRO*C.

SQLCA.H	Contiene las definiciones para la manipulación de errores.
SQLDA.H	Definición del descriptor para consultas dinámicas

D.5 Archivos de Trabajo.

NACAUTOM.DAT	Contiene la tabla de transiciones del reconocedor del lenguaje.
NACSYSPR.DAT	Contiene los predicados del sistema.
NACPREFI.TAB	Contiene la representación interna de la consulta.
NACERROR.ERR	Mensajes de error generados durante la conpilación.
NACERROR.MSG	Mensajes de error del parser.
NACFUNCT.C	Almacena el texto fuente de las funciones.
NACTES.TAB	Almacena las constantes que aparecen en la consulta.
NACVARS.TAB	Almacena las variables que aparecen en la consulta.
NACMERR.MSG	Mensajes de error del monitor.

D.6 Archivos para Almacenar la Versión Compilada.

<nombre programa>.T01	Tabla de Funciones.
<nombre programa>.T02	Tabla de Reglas.
<nombre programa>.T03	Tabla de atributos.
<nombre programa>.T04	Tabla de Restricciones de Integridad.
<nombre programa>.T05	Tabla de Relaciones.
<nombre programa>.T06	Tabla de variables.

<nombre programa>.T07 Tabla de constantes.

<nombre programa>.T08 Representación en notación polaca de las reglas.

APENDICE E

ESTRUCTURAS DE DATOS

E.1 Tablas para almacenar los objetos del lenguaje.

```
struct functions {
        int fnsysname;                  /* Identificador del sistema para la función */
        char fnusername[namesize];      /* Nombre de la función */
        int fntype;                     /* Tipo de función */
        int fpars;                      /* Número de parámetros de la función */
        }; /* Tabla de Funciones */


struct rules {
        int rulesysid;                  /* Identificador del sistema para la regla*/
        char ruleusername[namesize];    /* Nombre de la regla */
        char ruleconsname[namesize];    /* Nombre del consecuente de la regla */
        struct tree *ruletree;          /* Apuntador a la estructura de la regla */
        int rpars;                      /* Número de parámetros de la regla */
        };      /* Tabla de reglas */


struct atrofrel {
        int rulesysid;          /* Identificador del sistema para la relación */
        struct lstatr *plstatr; /* Apuntador a una lista de atributos de la relación */
        };      /* Tabla de relaciones */
```

```
struct atributtes {
        int atrsysname;    /* Nombre del atributo */
        int atrtype;       /* tipo del atributo */
        int atrsize1;      /* tamaño de la parte entera del atributo */
        int atrsize2;      /* tamaño de la parte fraccionaria del atributo */
        };    /* Tabla de Atributos */


struct lstatr {
        int valstatr;                 /* Identificador del sistema para el atributo */
        struct lstatr *nextlstatr;    /* Apuntador al siguiente atributo de la lista */
        };    /* Lista de Atributos */


struct varatr {
        int varatrsysname;                  /* Identificador del sistema */
        char varatrusername[namesize];      /* Nombre de la variable o atributo */
        };    /* Tabla de variables y nombres de atributos */


struct constants {
        int ctesysname;    /* Identificador del sistema */
        char ctetype;      /* Tipo de constante */
        };    /* Tabla de Constantes */


struct ctestring {
        int ctestrsysname;    /* Identificador del sistema */
        char *ctestrvalue;    /* Valor de la constante */
        };    /* Tabla de Constantes "String" */
```

```
struct cteinteger  {
        int cteintsysname;      /* Identificador del sistema */
        int cteintvalue;        /* Valor de la constante */
        };      /* Tabla de constantes  de tipo Entero */


struct ctereal  {
        int cterealsysname;     /* Código de la constante real */
        float cterealvalue;     /* Valor */
        };    /* Tabla de constantes de tipo real */


struct constraints  {
        int constsysname;                   /* Identificador del sistema */
        char constusername[namesize]; /* Nombre de la restricción */
        char constconsname[namesize]; /* Nombre del consecuente */
        struct tree *constmterm;            /* Apuntador a la estructura de la restricción */
        int cpars;                          /* Número de argumentos */
        };     /* Tabla de Restricciones de Integridad  Semántica*/


struct tree  {
        struct mterm *treemterm;   /* Apuntador a un multitérmino */
        int treeop;                  /* Tipo de operación */
        struct listree *treelist;      /* Apuntador a su lista de predicados */
        };      /* Estructura para la representación interna de una regla */


struct listree  {
        struct tree *valuetree;      /* Apuntador al hijo */
        struct listree *next;        /* Ap. al siguiente hijo */
        }; /* Lista de hijos de un nodo del árbol de inferencia */
```

```
struct syspr {
        int syspredid;                          /* Identificador del sistema */
        char syspredusername[namesize];         /* Nombre */
        struct rules *syspredrule;              /* Estructura del predicado */
        }; /* Predicados del sistema */
```

E.2 Estructuras de datos para Unificación.

```
struct system {
        struct listmeq *T;          /* Parte T */
        struct upart *U;            /* Parte U */
        }; /* Sistema R para el proceso de unificación */


struct upart {
        int meqnumber;                          /* Número de multiecuaciones */
        struct listmeq *zerocountermeq;         /* Multiecuaciones con contador cero*/
        struct listmeq *equations;              /* Multiec. con contador distinto de cero*/
        }; /* Parte U */


struct mterm {
        char *fsymb;                    /* Nombre del Multitérmino */
        char ftype;                     /* Tipo del Multitérmino */
        struct listtemeq *args;         /* Argumentos del Multitérmino */
        }; /* Multitérmino */


struct meq {
        int counter;                /* Contador */
        int varnumber;              /* Número de variables del lado izquierdo */
        struct listvars *S;         /* Lista de variables */
        struct mterm *M;            /* Multitérmino */
        }; /* Multiecuación */
```

```
struct tempmeq   {
          struct queuevars *S;          /* Lista de variables */
          struct mterm *M;              /* Multitérmino */
          }; /* Multiecuación temporal */


struct variable   {
          char *namevar;             /* Nombre */
          struct meq *M;             /* Multiecuación donde aparece la variable */
          }; /* Variables */


struct listmeq   {
          struct meq *valmeq;          /* Apuntador a la multiecuación */
          struct listmeq *next;        /* Ap. a la siguiente multiecuación */
          }; /* Lista de multiecuaciones */


struct listtemeq   {
          struct tempmeq *valtemeq;       /* Ap. a la multiecuación temporal */
          struct listtemeq *next;         /* Ap. a la siguiente multiec. temporal*/
          }; /* Lista de multiecuaciones temporales */


struct listvars   {
          struct variable *valvars;     /* Apuntador a la variable */
          struct listvars *next;        /* Ap. al siguiente elemento de la lista*/
          }; /* lista de variables */
```

```
struct queuevars  {
        struct variable *valqvar;        /* Valor */
        struct queuevars *next;          /* Siguiente variable */
        }; /* Cola de variables */


struct vars_in_mterm  {
        char *vars_tvaratr;                  /* Valor */
        struct variable *vars_variable;      /* Ap. a la estructura de la variable */
        struct vars_in_mterm *next;          /* Ap. a la siguiente variable */
        }; /* variables que aparecen en un multitérmino */
```

E.3 Estructuras de datos para el parser.

```
struct varperule  {
        int varuleini;     /* Identificador para la primera variable*/
        int varulefin;     /* Identificador para la última variable */
        }; /* Variables que aparecen en una regla */


struct stack  {
        char sysidtype;          /* Identificador del tipo de elemento */
        int sysidnum;            /* Número de elementos  */
        struct list *sterm;      /* Lista de términos */
        }; /* Estructura del "stack" */


struct terminfo  {
        char termid;             /* Identificador del tipo de término */
        int termnum;             /* Número de identificación */
        struct list *terms;      /* Lista de términos */
        }; /* Información de un término */
```

```
struct lstprvar {
        struct aritvarpred *valstpred;    /* Valor */
        struct lstprvar *next;            /* Ap. a la siguiente variable */
        }; /* Lista de variables de un predicado aritmético*/


struct aritvarpred {
        struct rules *vpred;         /* Regla */
        struct varatr *vpvar;        /* Variable */
        }; /* Variables de un predicado aritmético */


struct automata {
        char charstate;                  /* Caracter del estado actual */
        int  nextstate;                  /* Número del estado de transición */
        struct automata *otherchar;   /* Ap. al siguente caracter del estado */
        }; /* Autómata */
```

E.4 Estructuras de datos para sustitución.

```
struct varpred {
        struct listvars *P;        /* Variables del predicado */
        struct listvars *Q;        /* Variables del consecuente */
        }; /* Listas de variables Vp y Vq para sustitución */
```

E.5 Estructuras de datos para evaluación del árbol de inferencia

```
struct listviews {
        char *pnameview;        /* Nombre de la vista */
        struct listviews *next; /* Ap. al siguiente nombre de la vista */
        }; /* Lista de nombres de vistas creadas en la base de datos*/
```

APENDICE F.

TABLA DE TRANSICIONES DEL RECONOCEDOR DEL LENGUAJE

Edo.	Transiciones											
0	F / 1	R / 14	C / 37	S / 55	I / 61	D / 68	L / 78	G / 85	E / 89	M / 91	P / 100	A /104
0	N / 107	O / 110	+ / 112	- / 113	• / 114	% / 116	< / 117	> / 120	= / 122	(/ 123	) / 124	, / 125
0	; / 126	: / 127	" / 128	{ / 132	} / 133	! / 129	@ / 130	$ / 131	| / 0	?/-256		
1	U/2	?/130										
2	N/3	?/130										
3	C/4	?/130										
4	T/5	?/130										
5	I/6	?/130										
6	O/7	?/130										
7	N/8	?/130										
8	S/9	?/130										
9	_/10	?/130										
10	D/11	?/130										
11	E/12	?/130										
12	F/13	?/130										
13	#/-1	?/130										
14	U/15	E/23	?/130									
15	L/16	?/130										
16	E/17	?/130										
17	S/18	#/-7	?/130									
18	_/19	?/130										
19	D/20	?/130										

20	E/21	?/130		
21	F/22	?/130		
22	#/-6	?/130		
23	A/24	L/26	?/130	
24	L/25	?/130		
25	#/-11	?/130		
26	A/27	?/130		
27	T/28	?/130		
28	I/29	?/130		
29	O/30	?/130		
30	N/31	?/130		
31	S/32	#/-3	?/130	
32	_ /33	?/130		
33	D/34	?/130		
34	E/35	?/130		
35	F/36	?/130		
36	#/-2	?/130		
37	H/38	O/41	?/130	
38	A/39	?/130		
39	R/40	?/130		
40	#/-9	?/130		
41	N/42	?/130		
42	S/43	?/130		
43	T/44	?/130		
44	R/45	?/130		
45	A/46	?/130		
46	I/47	?/130		
47	N/48	?/130		
48	T/49	?/130		
49	S/50	#/-5	?/130	
50	_ /51	?/130		
51	D/52	?/130		
52	E/53	?/130		
53	F/54	?/130		
54	#/-4	?/130		

55	T/56	?/130			
56	R/57	?/130			
57	I/58	?/130			
58	N/59	?/130			
59	G/60	?/130			
60	#/-8	?/130			
61	N/62	?/130			
62	T/63	?/130			
63	E/64	?/130			
64	G/65	?/130			
65	E/66	?/130			
66	R/67	?/130			
67	#/-10	?/130			
68	O/71	I/69	?/130		
69	V/70	?/130			
70	#/-265	?/130			
71	C/72	U/74	?/130		
72	(/73	I/72	?/130		
73	)/-14	?/73			
74	B/75	?/130			
75	L/76	?/130			
76	E/77	?/130			
77	#/-13	?/130			
78	T/79	E/80	0/82	?/130	
79	#/-517	?/130			
80	Q/81	?/130			
81	#/-518	?/130			
82	N/83	?/130			
83	G/84	?/130			
84	#/-12	?/130			
85	T/86	E/87	?/130		
86	#/-519	?/130			
87	Q/88	?/130			
88	#/-520	?/130			
89	Q/90	?/130			

90	#/-521	?/130			
91	I/92	0/96	U/98	?/130	
92	N/93	?/130			
93	U/94	?/130			
94	S/95	?/130			
95	#/-263	?/130			
96	D/97	?/130			
97	#/-266	?/130			
98	L/99	?/130			
99	#/-264	?/130			
100	L/ 101	?/130			
101	U/ 102	?/130			
102	S /103	?/130			
103	#/ -262	?/ 130			
104	N /105	?/130			
105	D /106	?/130			
106	# /-833	?/130			
107	O /108	?/130			
108	T /109	?/130			
109	# /-834	?/130			
110	R /111	?/130			
111	# /-833	?/130			
112	? /-257				
113	? /-258				
114	? /-259				
115	? /-260				
116	? /-261				
117	- /118	=/119	? /-512		
118	? /-15				
119	? /-513				
120	= /121	?/-514			
121	? /-515				
122	? /-516				
123	? /-769				
124	? /-773				

125	? /-772				
126	? /-770				
127	? /-774				
128	* /-253	? /128			
129	! /129	? /0			
130	@ /130	$ /130	/130	? /-254	
131	$ /131	? /-255			
132	? /-775				
133	? /-776				

Notación:

<carácter> / <edo. de transición>

donde :

<carácter> ::= <número> | <letra> | <símbolo del lenguaje> | <símbolo de control>

<símbolo de control> ::= @|$| ! | # |? | |

@ - carácter alfanumérico

$ - dígito numérico

! - CR o LF

| - espacio

- Separador

? - cualquier cosa.

APENDICE G

EL PRECOMPILADOR

PARA C DE ORACLE (PRO*C).

PRO*C fue diseñado para convertir un programa de C que contiene expresiones de SQL a un programa en C que puede accesar y manipular datos en la base de datos ORACLE. Un programa en PRO*C está constituido por dos partes: el prólogo de la aplicación y el cuerpo de la aplicación.

El Prólogo de la aplicación.- Es donde se definen las variables y consta de tres componentes:

a) La sección de declaración contiene todas las variables "host" [1]. Esta sección se delimita con las dos cláusulas siguientes:

EXEC SQL BEGIN DECLARE SECTION

.

.

EXEC SQL END DECLARE SECTION

b) Una área de comunicación con SQL para la manipulación de errores. Esta área consiste únicamente de la linea :

EXEC SQL INCLUDE SQLCA;

c) Un enunciado para conexión con ORACLE .

EXEC SQL CONNECT :user INDENTIFIED BY :password

donde user es la clave del usuario para accesar la base de datos y password es una palabra clave de acceso. Otra forma de este enunciado es dando en una misma variable la clave del usuario y el password :

---

(1) Las variables "host: son variables que pueden referenciarse desde expresiones de SQL o desde instrucciones de programación.

EXEC SQL :oracleid

donde oracleid contiene user/password.

El cuerpo de la aplicación .-contiene expresiones de SQL para consultar y manipular los datos almacenados en una base de datos ORACLE.

PRO*C cuenta con cuatro métodos para ejecutar cláusulas de SQL;

1) Ejecución Inmediata.

Precompila cualquier expresión de SQL (exepto un SELECT) y la ejecuta. La sentencia de SQL no puede contener variables host. A una sentencia de ejecución inmediata se le puede pasar la expresión de SQL en una variable host o bien contener directamente la variable host.

EXEC SQL EXECUTE INMEDIATE :

Ejemplo: Considérese el siguiente segmento de programa:

```
EXEC SQL BEGIN DECLARE SECTION;
.
VARCHAR dstring (80);
.
EXEC SQL END DECLARE  SECTION;
scanf ("%s",dstring);
.
EXEC SQL EXECUTE IMMEDIATE :dstring;
```

La limitación de este método radica en que la cláusula de SQL sólo se puede ejecutar una vez con un sólo parámetro y es estática.

2) Preparación-Ejecución.

La ventaja de este método radica en que el enunciado de SQL se analiza(prepara) sólo una vez, pero se puede ejecutar varias veces. En este método la expresión de SQL puede contener una variable host , por ejemplo:

```
DELETE FROM EMP WHERE EMPID :EMPNO
```

donde EMPNO es una variable host que contiene el valor de EMPID a ser seleccionado.

Ejemplo: Sea el siguiente segmento de programa:

```
.
scanf ("%s",stringSQL);
EXEC SQL PREPARE S1 FROM :stringSQL;
scanf("%d",&PEMPNO);
while( PEMPNO != 0 )
{ EXEC SQL EXECUTE S1 USING :PEMPNO;
  scanf("%d",&PEMPNO);  }
```

En este ejemplo la expresión de SQL está contenida en la variable stringSQL, la cual se prepara y se ejecuta mientras la variable PEMPNO sea distinta de cero, sin necesidad de preparar la expresión de SQL nuevamente.

Las limitaciones de este método consisten en que el número de parámetros y sus tipos deben conocerse de antemano debido a que las variables host de entrada se deben declarar en la sección DECLARE.

3) Prepara-Abre-y-busca.

Permite el uso de consultas pre-programadas, es decir, consultas cuya lista de atributos en la cláusula SELECT es conocida. Para realizar consultas PRO*C emplea la noción de cursores. Un cursor es una área de trabajo empleada por ORACLE para almacenar el resultado de las consultas. Existen cuatro comandos para manipular cursores:

DECLARE CURSOR
OPEN CURSOR
FETCH
CLOSE CURSOR

Antes de abrir un cursor es necesario definirlo y asignarlo a una consulta por medio de la cláusula DECLARE CURSOR. Todos las n-adas que cumplen los criterios de la consulta forman un conjunto denominado el conjunto activo del cursor. Las n-adas del conjunto activo se toman una por una por medio de la instrucción FETCH y cuando la consulta se haya concluido se debe cerrar el cursor con la cláusula CLOSE CURSOR [PRO*C].

La limitación de este método consiste en que los tipos de los atributos de la consulta deben conocerse previamente.

4) Consultas Dinámicas empleando descriptores.

El resultado de una consulta se deposita en las variables host, pero la lista de atributos del SELECT se conoce hasta que el usuario efectúa la consulta, por lo que el número y el tipo de las variables host no puede predefinirse.

Para poder efectuar consultas dinámicas se emplean cursores en conjunción con descriptores. La declaración de un descriptor se hace por medio de la sentencia DESCRIBE de SQL, la cual examina la lista del SELECT después de que se preparó y determina el número de atributos y los tipos de los atributos.

El área para almacenamiento asignada por DESCRIBE se denomina SQLDA (SQL desciptor area). Para declarar el SQLDA se debe incluir en el programa la sentencia:

EXEC SQL INCLUDE SQLDA;

la cual no debe incluirse en la sección de declaraciones .

El SQLDA es empleado para almacenar la información obtenida de una consulta y también se emplea para almacenar información de las variables host de entrada denominadas variables de ligado. La estructura que representa el SQLDA se muestra en la figura G-1.

Los pasos para procesar una consulta dinámica son:

1.- Declarar la estructura SQLDA. Se debe de incluir la sentencia ya descrita:

EXEC SQL INCLUDE SQLDA;

2.- Preparar la instrucción de SQL proporcionada por el usuario. Este paso efectúa dos cosas: compila la sentencia de SQL y le asocia un nombre. La cláusula empleada por PRO*C para este paso es:

PREPARE <nombre sentencia> FROM <variable host>

3.- Declarar un cursor para el nombre de la instrucción. La sintaxis empleada para declarar el cursor que se emplea para efectuar la consulta es:

EXE SQL DECLARE <nomb. cursor> CURSOR FOR <nombre sentencia>.

4.- Obtener una instancia de SQLDA. PRO*C cuenta con una función que permite obtener una instancia del SQLDA:

nombre_sqlda = sqlald(a,b,c)

donde: a es el número de variables a ser descritas

b es la longitud de los nombres de las variables host

c es la longitud de los nombres de las variables indicadoras.

5.- Devolver el almacenamiento del SQLDA. Para este paso se emplea la función **sqlclu,** a la cual se le pasa un apuntador al descriptor cuya área será devuelta al sistema.

6.- Declarar el descriptor de ligado en el SQLDA. El descriptor de ligado se emplea para almacenar las variables de entrada empleadas durante la evaluación de la sentencia SQL. La declaración consiste de:

EXEC SQL DESCRIBE BIND FOR sentencia INTO sqlda_nombre

esta instrucción debe aparecer después de un PREPARE y antes de un OPEN.

7.- Abrir el cursor. Debido a que en las consultas dinámicas el cursor esta asociado con un descriptor de ligado, la instrucción para abrir un cursor es:

EXE SQL OPEN USING DESCRIPTOR sqlda_nombre

8.- Declarar el descriptor de selección en el SQLDA. El descriptor de selección es requerido para asociarle a la lista de atributos del SELECT un descriptor de salida, el cual se declara como sigue:

EXEC SQL DESCRIBE SELECT LIST FOR < nombre sentencia >

Esta cláusula debe aparecer despues del PREPARE, DESCRIBE BIND y OPEN, y antes del FETCH.

9.- Traer los resultados. Se emplea la cláusula FETCH por cada tuplo:

EXEC SQL FETCH < cursor > USING DESCRIPTOR < descriptor de salida >.

10.- Cerrar el cursor. Es análoga a consultas estáticas.

```
struct sqlda {
        int N;          /*tamaño del descriptor con respecto al número de elementos*/
        char **V;       /*apuntador a un arreglo de direcciones de las vars. principales*/
        int *L;         /*apuntador a un arreglo que contiene la longitud de los buffers*/
        short *T;       /*apuntador a un arreglo que contiene los tipos de los buffers*/
        short **I;      /*apunt. a un arreglo de direcciones a las vars. indicadoras*/
        int F;          /*número de variables encontradas por DESCRIBE*/
        char **S;       /*apuntador a un arreglo de variables de apuntadores a los nombres*/
        short *M;       /*apunt a un arreglo de longitudes máximas de nombres de variables*/
        short *C;       /*apunt a un arreglo de longitudes actuales de nombres de variables*/
        char **X;       /*apunt a un arreglo de apuntadores a nombres de vars indicadoras*/
        short *Y;       /*apunt a un arreglo de long maximas de nombres de vars indicadoras*/
        short *Z;       /*apunta un arreglo de long actual de nombres de vars. indicadoras*/   }
```

Figura G-1 Estructura del SQLDA.

Bibliografía.

[Alag85] Alagic, Suad.
Relational Database Technology
Springer-Verlag, 1985.

[Ullm82] Ullman, Jeffrey D.
Principles of Database Systems
Computer Science Press, Inc. Segunda edicion, 1982.

[ChLe73] Chang, Chin-Liang & Lee, Richard Char-Tung.
Symbolic Logic and Mechanical Theorem Proving
Academic Press. 1973.

[Ullm88] Ullman, Jeffrey D.
Principles of Database Knowledge-Base Systems Computer Science Press, Inc., 1988, U.S.A.

[Date83] Date, C.J.
An Introduction to Database Systems
Addison-Wesley, USA, Vol. 2, 1983.

Referencias.

[GaMn84] Gallaire, H., Minker, J. & Nicolas, J.M.
Logic and Databases: A deductive Approach
ACM Comp. Surveys, Vol. 16, No. 2, June 1984.

[MaMo82] Martelli, A. & Montanari, U.
An Efficient Unification Algorithm
ACM Trans. on Prog. LAng. and Systems
Vol. 4, No. 2, April 1982.

[HeEn72] Herbert, B. Enderton.
A Mathematical Introduction to Logic
Academic Press, Inc., USA, 1972.

[Dale83] Dalen, D.Van.
Logic and Structure
Springer-Verlag, Germany, 2a. ed, 1983.

[Lloy84] Lloyd, J.W.
Foundations of Logic Programming
Springer-Verlag, Germany, 1984.

[JuAr90] Juárez, Cristóbal & Aranda, Norma.
ADeductive Database System for Expert Systems Programming
Comunicaciones Técnicas IIMAS-UNAM
(serie naranja No. 558), México 1990

[HaWa83] Hayes-Roth, Frederick & Waterman, Donal. Lenat Douglas.
Building Expert Systems
Addison-Wesley Publishing Company Inc., 1988, USA.

[JüGü88] Jüttner, Gerald & Güntzer, Ulrich.
Methoden der Kunstlichen Intelligenz fur Information Retrieval
K.G. Saur, 1988, Alemania.

[BrMy84] Brodie, Michael & Mylopoulos, John.
Topics in Information Systems on Conceptual Modelling
Springer-Verlag, 1984, New York, U.S.A.

[PRO*C] ORACLE. PRO*C User's Guide.
Oracle Corporation, 1987, U.S.A.

[SQL*Plus] ORACLE. SQL*Plus User's Guide
Oracle Corporation, 1987, U.S.A

[KeRi79] Kernighan, Brian & Ritchie, Dennis
The C Programming Language
Prentice Hall Inc., Englewood Cliffs,1979,New Jersey.

[MyLe84] Mylopolus, J & Levesque, H.J.
An Overview of Knowledge representation.
En [BrMy84], Springer Verlag, USA, 1984.

[FrSc90] Freitag, Burkhard., Schütz, Heribert & Specht, Günther
LOLA- A Logic Language for deductive Databases and its Implantation.
Technische Universitat Munchen. November, 1990.

[ApBl88] Apt K.R. , Blair, H.A., & Walker, A.
Towards a theory of Declarative Knoweledge
En [Mink88].

[Mink88] Minker, Jack.
Foundations of Deductive Databases and Logic Programming
Morgan Knufmann Publishers, Inc. 1988, USA.