DESARROLLO DE UN INTERPRETE DE PROLOG

por

MARIA DEL SOCORRO VARGAS VERA

Tesis presentada a la unidad académica de los Ciclos Profesional y de Posgrado del C.C.H. de la Universidad Nacional Autonoma de México

PARA OBTENER EL GRADO DE MAESTRO EN CIENCIAS DE LA COMPUTACION

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO.
enero, 1988

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

Desarrollo de un intérprete de PROLOG

Aprobada por el siguiente jurado:

Dr. Francisco Cervantes P.	presidente
M en C. Ricardo Ciria M.	secretario
M en C. Guillermo Levine G.	vocal
M en C. Sergio R. Cárdenas G.	suplente
M en C. Luz Marina Quiroga C.	suplente

Tabla de contenido

Resumen..... ix

CAPITULO 1: Introducción..... 1

1.1 Motivación..... 1

1.2 Objetivos del trabajo..... 2

1.3 Sistema de bases de datos para CAD..... 3

1.4 Organización del trabajo..... 7

CAPITULO 2: Descripción del lenguaje PROLOG..... 8

2.1 Conceptos generales de PROLOG..... 9

2.1.1 Trabajos previos..... 9

2.1.2 El lenguaje PROLOG..... 11

2.1.2.1 Tipos de datos..... 11

2.1.2.2 Operaciones..... 18

2.1.3 Construcción de un programa en PROLOG..... 19

2.1.4 Mecanismos usados por PROLOG para la ejecución de un programa..... 23

2.1.4.1 Ejemplo de la ejecución de un programa..... 26

2.2 Fundamentación Matemática..... 28

2.2.1 Programación lógica..... 28

2.2.2 Cálculo de predicados de primer orden..... 29

2.2.3 Formas clausales..... 30

2.2.4 Cláusulas de Horn..... 33

2.3 El lenguaje PROLOG y la lógica..... 38

2.3.1 Los principios de resolución y prueba automática de teoremas..... 38

2.3.2 El uso del principio de resolución en PROLOG..... 41
2.4 Consideraciones semánticas de los programas..... 45
2.5 Características sobresalientes de PROLOG..... 47
2.6 Algunos usos específicos..... 49

CAPITULO 3: Descripción del intérprete..... 51
3.1 Gramática seleccionada..... 51
3.2 Analizador lexicográfico..... 59
3.3 Analizador sintáctico..... 62
3.4 Estructuras de datos..... 63
3.4.1 Estructuras de datos creadas durante el análisis sintáctico..... 63
3.4.2 Estructuras de datos al tiempo de ejecución.... 71
3.5 Representación interna..... 79
3.5.1 Formas de almacenamiento..... 79
3.6 Biblioteca de rutinas del sistema..... 85
3.7 El intérprete..... 99
3.7.1 Unificación.....101
3.7.1.1 Algoritmo de Unificación.....105
3.7.2 Algoritmo ABC para interpretar programas.....108
3.7.3 Optimización en el caso de la recursión.....110
3.7.4 Diagramas de Ferguson.....111
3.7.5 El análisis semántico.....112
3.7.6 Ejemplo de un programa ejecutado por el intérprete.....113

CAPITULO 4: Discusión.....117

CAPITULO 5: Conclusiones.....120

CAPITULO 6: Etapas Futuras.....121

APENDICE 1: Pruebas del intérprete.....125

APENDICE 2: Manual para el usuario.....143

APENDICE 3: Estructura del Intérprete.....146

APENDICE 4: Glosario de términos.....153

BIBLIOGRAFIA.....150

1.1 Sistema manejador de bases de datos para aplicaciones de la computación al diseño (CAD)..... 5
2.1 Poder de representación de PROLOG..... 37
3.1 Ambigüedad de la gramática de PROLOG..... 54
3.2 Diccionario de datos..... 66
3.3 Descriptor para una rutina del sistema..... 67
3.4 Descriptor para una cláusula definida por el usuario..... 69
3.5 Ejemplo de la representación interna..... 70
3.6 Descriptores de los parámetros..... 72
3.7 Estructuras de datos al tiempo de ejecución..... 74
3.8 Representación interna de cadenas..... 81
3.9 Casos de unificación.....103
3.10 Diagrama de Ferguson.....115
A3.1 Arbol de rutinas del intérprete.....147

RESUMEN

Este trabajo describe el desarrollo de un intérprete de PROLOG utilizando el sistema operativo Unix y herramientas como YACC, LEX y MAKE que sirven para el desarrollo de lenguajes y programas para computadora. El intérprete se hizo utilizando la técnica de copia de estructuras[1] y además, a diferencia de otros intérpretes, reconoce directamente el lenguaje PROLOG. Esto es importante ya que otros usan un intérprete para un lenguaje intermedio de menor nivel que PROLOG, pero que maneja unificación y la técnica de retroceso ("backtracking"), para escribir en él el programa que finalmente interpretará PROLOG, lo cual obviamente es más lento por el doble proceso de interpretación que esto implica. Adicionalmente, al intérprete que es presentado en este trabajo, se le incluyeron facilidades para la depuración de programas que permiten rastrear la ejecución de los mismos ("trace").

En el diseño de intérpretes un aspecto que se debe tener en cuenta es la velocidad de ejecución sobre todo si el intérprete va a ser utilizado en aplicaciones que requieren tiempos de respuesta interactiva o de tiempo real por esta razón se decidió usar una representación interna para los programas en lugar de interpretar directamente usando el código fuente como inicialmente fue planteado cuando se propuso su desarrollo. El diseño de la representación interna se organizó de forma que permita acelerar el proceso de interpretación y de retroceso, para lo cual las cláusulas se tuvieron que agrupar por el nombre y aridad. Por otro lado otro factor que influye en la rapidez del intérprete, como ya se mencionó antes, es que reconoce directamente las construcciones del lenguaje PROLOG. En la versión actual todos los operadores son manejados en forma funcional, lo cual significa que no existe el manejo dinámico de precedencias para éstos.

---

[1] Es uno de los métodos que se usan para realizar la representación de términos durante la ejecución de los programas, en el que se crea una copia cuando una variable se acota a un término del tipo functor o lista.

CAPITULO 1

INTRODUCCION

1.1 Motivación

En el IIMAS-UNAM se está desarrollando un sistema manejador de bases de datos para CAD, uno de los módulos de este sistema es el manejador de restricciones de diseño el cual requiere tener la capacidad de definir desde restricciones simples hasta restricciones muy complejas. Adicionalmente y debido a que los datos del diseño se almacenan por medio de un modelo de datos orientado a representar clases e instancias de objetos, se necesita que el manejador de restricciones sea capaz de manipular la herencia de las restricciones de diseño a través de los objetos. Fue por estas características tan complejas del manejo de restricciones que el grupo que está desarrollando el sistema manejador de bases de datos para CAD creyó que PROLOG sería una herramienta que podría resolver el problema del manejo de restricciones gracias a su poder para manipular reglas o información lógica.

Con esta idea en mente se investigaron los costos de varios intérpretes de PROLOG para ambientes de Unix que se venden comercialmente, se encontró que su precio oscilaba alrededor de seis mil dolares y únicamente se proporcionaban los programas objeto. Debido a estos costos y a que el grupo

de CAD deseaba tener los fuentes para poder hacer las extensiones necesarias para manipular las restricciones de diseño, dentro del sistema manejador de bases de datos para CAD, se decidió realizar un interpréte[1] de PROLOG.

Otro motivo por el que se desarrolló este trabajo es el deseo personal de trabajar en el área de inteligencia artificial en el desarrollo de sistemas expertos, aplicación que requiere de lenguajes de muy alto nivel que presenten características similares a las que tiene PROLOG tales como permitir la definición de reglas en forma de programas y el manejo de retroceso ("backtracking") durante la búsqueda de alternativas provocada por la ejecución de estos programas.

1.2 Objetivos del trabajo

El objetivo del trabajo fue desarrollar un intérprete de PROLOG que interpretara directamente sobre el código fuente, del que se tuvieran los programas fuentes para que después de tener el intérprete en una etapa futura (en otra tesis o investigación) se le pudieran agregar la serie de

---

[1] Un intérprete es un programa que permite que la ejecución de un programa fuente se ejecute instrucción por instrucción. No se almacena un código intermedio para ejecutarse posteriormente, como hace un compilador. Dentro de las características que presenta es que se facilita la implantación de construcciones complejas de lenguajes de programación, aunque el tiempo de ejecución de un programa interpretado es más largo que el correspondiente a un programa compilado.

rutinas de trabajo para el manejo de restricciones de diseño. Debido a que el sistema manejador de bases de datos para CAD se estaba desarrollando en lenguaje "C" y bajo el sistema operativo Unix, el intérprete se desarrolló usando el mismo ambiente de programación para facilitar su posterior integración a dicho sistema manejador de bases de datos. Otra característica por la que se usó Unix es que proporciona un ambiente de programación más completo que el de MSDOS y otros sistemas ya que provee un conjunto de herramientas para ayudar al programador de aplicaciones en su tarea.

1.3 Sistema manejador de bases de datos para CAD

El sistema manejador de bases de datos para aplicaciones de CAD [BUCA86] es un modelo semántico basado en la noción de agregados moleculares. Un agregado molecular puede ser visto como un conjunto de objetos y ligas. Estos objetos pueden ser atómicos (indivisibles) y moleculares. Los objetos moleculares son de un grado de abstracción más alto y consisten a su vez en objetos atómicos y otros objetos de grado de abstracción menor. Las ligas son asociaciones que pueden ser especificadas por el usuario y llevan la semántica bajo la cual los objetos están ligados con el agregado en particular [BATD84]. El Sistema Manejador de Bases de Datos para aplicaciones de CAD (véase figura 1.1) está formado por:

1. Una base de datos global de proyecto(BD/P). Contiene la información de un proyecto o proyectos.

2. Varias bases de datos de áreas de trabajo, que son extracciones de la base de datos global que un diseñador utilizará. En algún momento el diseñador solicitará la reintegración de sus modificaciones a la base de datos del proyecto. En ese momento se verificarán las restricciones para asegurar la consistencia de los datos.

3. Un diccionario de datos y control de la base de datos. Contiene la información acerca de la definición de los datos, como las unidades en las que están almacenados los datos.

4. Base de datos de plantillas y símbolos. Esta base de datos almacena la información que permite hacer la conversión entre unidades. Esto da versatilidad al sistema, ya que permite que cada usuario use los mismos datos pero en las unidades con las que acostumbra trabajar por ejemplo puede escoger mostrar longitudes en pulgadas, metros, centímetros, etc.

5. Una base de datos de catálogos. Con esta base de datos se tendrá información técnica sobre catálogos de las componentes que se van a utilizar para el proyecto.

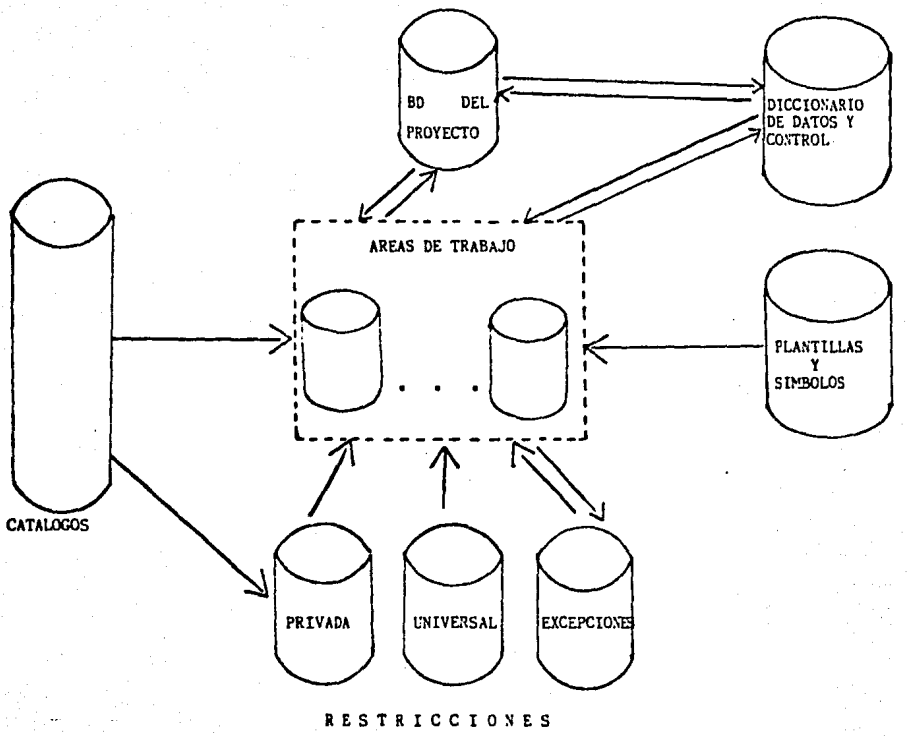

Figura 1.1 Sistema manejador de bases de datos para aplicaciones de la computación al diseño (CAD)

6. Bases de datos de restricciones. En éstas se almacenan las restricciones que define el usuario para asegurar que un diseño cumpla con las especificaciones pedidas.

Los tipos de restricciones que se van a poder verificar en el sistema manejador de bases de datos se han clasificado en los dos tipos siguientes:

1. Restricciones de valores

   a. Restricciones al dominio. Por ejemplo determinar si x está dentro de un intervalo de valores. Ejemplo: 0‹= X ‹= 1.

   b. Restricciones de enumeración. Verificar que un valor esté dentro de un conjunto de valores.

   c. Restricciones de relación. Estas restricciones pueden ir desde su forma más sencilla como X_d ›= X-op*.25 hasta restricciones de consistencia complejas que involucran muchos atributos y evaluación de procedimientos complejos.

2. Restricciones estructurales

   a. Clases de componentes. Estas restricciones se definen a nivel clase y serán heredadas a

los objetos que forman esa clase.

b. Número de objetos que lo componen. Estas restricciones especifican cuántas instancias de los objetos componentes pueden hacer uso de otro objeto.

c. Relaciones estructurales entre objetos componentes. Estas restricciones definen la topología de los objetos y pueden incluir las especificaciones del orden legal de los objetos componentes, y correspondencias entre objetos y diferentes niveles de abstracción.

1.4 Organización del trabajo

El trabajo está organizado de la siguiente forma:

En el capítulo 1 se hace una descripción del lenguaje PROLOG para familiarización con el lenguaje.

En el capítulo 2 se analiza la fundamentación lógica de PROLOG.

En el capítulo 3 se muestra el desarrollo del intérprete presentando sus componentes: la gramática del lenguaje, los analizadores lexicográfico y sintáctico, estructuras de datos usadas para almacenar los programas y las estructuras de datos para llevar el control del módulo que se encarga del proceso de interpretación, la biblioteca

de rutinas del sistema y finalmente el módulo que se encarga del proceso de interpretación que es el responsable de la carga semántica.

En el capítulo 4 se presentan la discusiones tomadas durante el desarrollo del intérprete.

En el capítulo 5 se presentan las conclusiones del trabajo.

En el capítulo 6 se presentan las extensiones que se le pueden hacer al intérprete a corto, mediano y largo plazo.

En el apéndice 1 se presentan unas pruebas del sistema.

En el apéndice 2 se presenta un manual para el usuario que da una guía sobre cómo utilizar el intérprete.

En el apéndice 3 se muestra la estructura del sistema y ejemplo de programas fuente.

En el apéndice 4 se incluye un glosario de términos. Finalmente en la sección de bibliografía se encontrarán las referencias más importantes usadas para el desarrollo de este trabajo.

CAPITULO 2

DESCRIPCION DEL LENGUAJE PROLOG

2.1 CONCEPTOS GENERALES DE PROLOG

2.1.1 Trabajos Previos

PROLOG es un lenguaje basado en lógica de primer orden. Fue desarrollado por A. Colmerauer y su grupo en la Universidad de Marsella [COEH82] y que fue concebido como una herramienta de programación lógica. Actualmente PROLOG ha cobrado importancia debido a que ha sido adoptado como lenguaje núcleo del proyecto japonés de Quinta Generación. El primer intérprete de PROLOG fue escrito en ALGOL-W por Philip Roussel, aunque los intérpretes más usados fueron los escritos en FORTRAN por Battani y Meloni; otra realización es la del intérprete y compilador DEC-10 PROLOG escritos en PROLOG, hechos por Luis Moniz, Fernando Pereira y David Warren[MONL78]. Actualmente se cuenta con desarrollos de PROLOG en varias máquinas, tales como Macintosh, IBM-PC, VAX (para los sistemas operativos VMS o UNIX), mainframes tales como DEC-10, DEC-20, e IBM 3081. Los más recientes desarrollos intentan ofrecer interfaces variadas con otros lenguajes (como C, Pascal, LISP, etc). Algunos de los intérpretes de PROLOG disponibles actualmente son:

1. ARITY/PROLOG, opera en la IBM-PC y en computadoras compatibles. Maneja llamadas al sistema y operaciones de E/S tipo UNIX.

2. EL "Logic WorkBench" PROLOG, está disponible en sistemas UNIX basados en un microprocesador motorola 68000. Maneja una interfaz con C y bases de datos externas. También ofrece la posibilidad de almacenar grandes bases de datos de PROLOG en disco y utilizarlas sin tener que cargarlas completas en la memoria principal.

3. POPLOG, desarrollado en la Universidad de Sussex, fue hecho en una DEC-VAX bajo el sistema operativo VMS y también está disponible bajo UNIX. El sistema POPLOG es una combinación de lenguajes de programación: desarrollo de ambientes interactivos de programas, editor de textos, sistemas de ayuda y sistemas de enseñanza. Maneja compilación incremental y ofrece interfaces con rutinas en C, Fortran y Ada.

4. TURBO PROLOG, disponible en computadoras IBM compatibles bajo el sistema operativo MS-DOS, incluye un editor.

5. QUINTUS PROLOG, disponible en VAX y en estaciones de trabajo SUN, contiene un compilador incremental

con optimización y una interfaz con C.

2.1.2 El lenguaje PROLOG

PROLOG es un lenguaje no convencional. En particular las estructuras de datos son un tanto diferentes a las de otros lenguajes de programación. Es usado para resolver problemas que involucran objetos y relaciones entre objetos.

2.1.2.1 Tipos de datos

Los tipos de datos que pueden ser manejados por PROLOG son:

a). **Constantes**

Las constantes son las estructuras de datos más primitivas del lenguaje, con frecuencia son llamadas átomos. Sirven para nombrar objetos y relaciones especificas. En PROLOG, como en otros lenguajes tales como LISP, no hay necesidad de declarar constantes o agruparlas en tipos. Se usan libremente simplemente escribiendo su nombre.

El nombre de una constante válida dentro de PROLOG puede ser:

1. Una secuencia de dígitos posiblemente prefijada, con un símbolo menos opcional. Por convención tales constantes son llamadas ENTEROS.

   Ejemplos: -2, 10, 1000, etc.

2. Un identificador, formado por letras, dígitos y símbolos "_", pero que debe empezar con una letra minúscula.

   Ejemplos: alicia, rosa, formado_por, etc.

3. Un símbolo que es una secuencia no vacía de cualesquiera de los siguientes caracteres:

   + - * / < = > . : ? $ & @ # ~

4. Cualquiera de los caracteres

   , ; !

5. El símbolo [] (llamado "nil")

6. Una secuencia de caracteres encerrados en apóstrofos.

   Ejemplo: ' Maestría en Ciencias de la Computación '

b). **Cadenas**

Son secuencias de caracteres delimitados por comillas. Ejemplo " Maestría en Ciencias de la Computación ".

c). **Listas**

Una lista es una secuencia de elementos de cualquier longitud, y cuyos elementos pueden ser constantes, cadenas, estructuras, variables o listas.

Una lista o es vacía (no tiene elementos) o es una estructura con dos componentes: cabeza y cola. En notación PROLOG el nombre del functor usado para listas es el punto (.) y se acostumbra, al final de una lista, poner como cola la lista vacía. Por ejemplo: .(a,[]) representa la lista que contiene un solo elemento: el elemento a. Otra notación más cómoda para representar listas consiste en poner los elementos separados por comas, y toda la lista encerrada entre paréntesis cuadrados. Tomando el mismo ejemplo tendríamos que .(a,[]) es equivalente a [a].

d). **Las estructuras u objetos compuestos**

La concepción básica de la estructura en PROLOG es similar, a la de estructuras en el lenguaje "C" y los

registros de "PASCAL", ya que su propósito es agrupar varios objetos simples bajo un sólo nombre. De manera que una estructura de PROLOG es un conjunto de una o más variables, constantes, listas o functores (estos se explicarán más adelante en esta sección) agrupados bajo el mismo nombre. Esta construcción del lenguaje permite que las referencias a ese grupo afin de información se haga como si se tratara de un objeto simple en vez de trabajar con las partes independientes que la forman.

En lenguajes de programación convencionales como "C" la estructura de un objeto compuesto es definida al describir la forma en que el objeto y sus componentes se encuentran asociados. La definición de esta asociación se limita a darle un nombre a la agregación de varios componentes, con lo cual se construye un nuevo tipo de objeto complejo. Como ejemplo se presenta la definición en "C" y PROLOG del tipo fecha que tiene tres componentes de tipo entero que son día, mes y año.

```
struct fecha
       {
        int     dia;
        int     mes;
        int     año;
       };
```

Definición del tipo fecha en lenguaje "C"

```
fecha  x;
       x.día = 28;
       x.mes = 12;
       x.año = 1987;
```

Declaración de una variable tipo fecha y su posterior instanciación con el valor 28 de diciembre de 1987

```
fecha(Día, Mes, Año).
```

Definición de fecha en lenguaje PROLOG

```
fecha(28, 12, 1987).
```

Declaración de una instancia u ocurrencia de fecha con valor 28 de diciembre de 1987

Del ejemplo se observa que la declaración de tipo establece el nombre de la relación objeto-componente, y da información adicional acerca de la estructura o tipo de los componentes. La definición del nombre se encuentra textualmente separada de sus ocurrencias. En PROLOG se usó una solución diferente en la que el nombre del tipo es parte integral de todas las ocurrencias de la descripción del objeto.

La notación usada en PROLOG para representar un objeto compuesto es el nombre del tipo, seguido por una secuencia entre paréntesis de descripciones de sus componentes, separados por comas. Debido a que la notación es muy similar a la de funciones, el nombre del tipo es llamado **functor**, y sus componentes son llamadas argumentos. Un functor tiene dos atributos:

nombre y aridad; esta ultima se refiere al número de argumentos. Para PROLOG los functores con mismo nombre pero diferente aridad son diferentes.

Las estructuras pueden ser tan complejas como sea necesario, por ejemplo podemos definir el registro "empleado" como sigue:

```
empleado(nombre(juan,perez),direccion(calle(rosas),
numero(150)))
```

El functor más externo, "empleado", define la estructura general y es llamado el functor principal.

e). Variables

Las variables son un tipo de datos que permiten hacer referencia objetos aún no conocidos. Un nombre de variable se escribe como un identificador empezando con una letra mayúscula o con un símbolo '_'. Ejemplo: Lu, _atomo, X, Y, etc.

Una variable es un objeto cuya estructura es totalmente desconocida. Durante el progreso de la ejecución del programa, la variable puede llegar a ser "instanciada" o acotada. Esto es, puede ser acotada a otra variable, a una lista, a una constante, a un objeto estructurado, o a una cadena.

También es posible manejar variables anónimas. El

nombre de una variable anónima se escribe como '_'. Se usan variables anónimas cuando se necesita usar una variable pero su valor nunca será usado. Ejemplo: si queremos saber si a Juan le gusta jugar, pero no necesitamos saber explícitamente qué juega, se puede usar una variable anónima:

```
?-juega(_,juan).
```

El signo de interrogación seguido de un guión antes de la cláusula que desea probar es la notación usada en PROLOG para formular preguntas.

f). Cláusulas

Una cláusula empieza con una cabeza o punto de entrada al procedimiento y continúa con un cuerpo. Si el cuerpo no es vacío, se separa de la cabeza por ":- ". Toda cláusula termina con un punto ".". La cabeza muestra la posible forma de los argumentos del predicado. El cuerpo consta de un número (posiblemente cero) de metas o llamadas a procedimientos, las cuales imponen condiciones para que la cabeza sea cierta.

En forma general una cláusula tiene la siguiente forma:

```
A :- B1, B2,...BN
```

nombre de una variable anónima se escribe como '_'. Se usan variables anónimas cuando se necesita usar una variable pero su valor nunca será usado. Ejemplo: si queremos saber si a Juan le gusta jugar, pero no necesitamos saber explícitamente qué juega, se puede usar una variable anónima:

```
?-juega(_,juan).
```

El signo de interrogación seguido de un guión antes de la cláusula que desea probar es la notación usada en PROLOG para formular preguntas.

f). Cláusulas

Una cláusula empieza con una cabeza o punto de entrada al procedimiento y continúa con un cuerpo. Si el cuerpo no es vacío, se separa de la cabeza por ":- ". Toda cláusula termina con un punto ".". La cabeza muestra la posible forma de los argumentos del predicado. El cuerpo consta de un número (posiblemente cero) de metas o llamadas a procedimientos, las cuales imponen condiciones para que la cabeza sea cierta.

En forma general una cláusula tiene la siguiente forma:

```
A :- B1, B2,...BN
```

que se interpreta como, "A es cierta si B1 y B2 y ... BN son ciertas". Se dice que A es la cabeza de esa cláusula y B1,B2,...BN forman el cuerpo. Si N=0 entonces tenemos un **hecho** o cláusula unitaria y ésta se escribe como "A." Si N>0 tenemos una regla o cláusula no unitaria. A su vez, B1, B2, ... BN, son cláusulas y cumplen con lo anterior.

2.1.2.2 Operaciones

La mayoría de las operaciones en programas en PROLOG consisten en llamadas a procedimientos[11] definidos por el usuario. Las operaciones estandáres como suma, comparaciones, E/S, etc, no necesitan ser definidas por el usuario. Para la realización de estas operaciones se tienen procedimientos o predicados predefinidos que el sistema proporciona para facilitar la tarea del programador, liberándolo así de tener que definirlos él mismo.

Características sobresalientes de las operaciones sobre constantes

Todas las constantes son puramente simbólicas y no tienen una interpretación propia; sin embargo, algunas operaciones de PROLOG las tratan en forma especial.

---

[11] Procedimiento es un conjunto de cláusulas con el mismo nombre]

1. Las operaciones aritméticas interpretan las cadenas de números como representación de enteros.

2. Las operaciones de comparación interpretan las cadenas de números como valores enteros, y todas las otras constantes como representación de la secuencia de caracteres que forman su nombre.

3. Las operaciones de E/S interpretan todos los símbolos como la secuencia de caracteres que forman su nombre.

2.1.3 Construcción de un programa en PROLOG

Un programa en PROLOG consiste en un conjunto de procedimientos, cada uno de los cuales constituye la definición de un cierto predicado.

Un procedimiento puede consistir en varias cláusulas; todas ellas deben tener una cabeza con el mismo símbolo de predicado, pero las especificaciones del procedimiento pueden diferir. Como ejemplo podríamos definir el procedimiento a:

Procedimiento a:

```
a([],L,L).
a([X|L1],L2,[X|L3]) :- a(L1,L2,L3).
```

El nombre de un procedimiento con frecuencia es llamado símbolo de predicado; al igual que los functores tiene dos atributos: nombre y aridad.

Algunas versiones de PROLOG permiten usar símbolos de predicado en notación prefija, postfija e infija.

En PROLOG, se crea una base de datos en la que se encuentran definidas todas las cláusulas del programa (axiomas), las cuales pueden ser usadas para contestar una pregunta. Una pregunta es una llamada a un procedimiento o una secuencia de llamadas a procedimientos, separados por un operador lógico. Su ejecución consiste en la ejecución de sus llamadas y resultado establece si la respuesta a la pregunta es cierta o falsa.

```
Ejemplo
juega(ajedrez,juan).
?- juega(X,juan).
TRUE
```

Ejemplos de programas escritos en PROLOG

Ejemplo 1. El siguiente ejemplo es tomado de [CLOW81], y trata de encontrar la densidad de población a partir de la población y el área de un país.

El predicado "población" relaciona un país X con Y millones de personas y el predicado "área" denota relaciones

entre el país Y y su área en millones de millas cuadradas.

En el ejemplo se usa la notacion de Edinburgo, esto es la coma que separa metas es el símbolo usado para conjunción de metas.

El predicado is es un predicado interconstruido que lo que hace es evaluar la expresión aritmética que está del lado derecho y asignarla a la variable que se encuentra del lado izquierdo.

```
poblacion(eu,203).
poblacion(india,548).
poblacion(china,800).
poblacion(brasil,108).
area(eu,3).
area(india,1).
area(china,4).

area(brasil,3).

densidad(X,Y) :-  poblacion (X,P), area(X,A),
                  Y is P/A.
```

```
La regla se lee como sigue:
  la densidad de población de una ciudad X es Y si:
   la población de X es P y
   el área de X es A y
   Y se calcula dividiendo P entre A.
```

Por ejemplo, para obtener de densidad de población de China preguntamos:

```
?- densidad(china,X), write(X).
```

A lo que PROLOG responderá:

```
X=200.
```

Ejemplo 2. El siguiente ejemplo determina si un elemento X está en una lista.

Para facilitar la explicación del ejemplo se numeraron las cláusulas del programa, pero esto no forma parte del lenguaje PROLOG.

Se define el predicado elemento_de_lista que tiene dos argumentos, el primero es el elemento a buscar 'X', y el segundo es la lista en la cual se realizará la búsqueda del elemento X.

Lo primero que se verifica es si X es la cabeza de la lista y de esto se encarga la cláusula (1). La cola de lista se puso como variable anónima porque no interesa el valor al cual se acota.

Lo segundo es buscar si X es elemento de la cola de la lista y esto se logra con la cláusula (2). En este caso no interesa el valor de la cabeza de la lista, razón por la cual usamos una variable anónima.

Por ejemplo, para obtener de densidad de población de China preguntamos:

```
?- densidad(china,X), write(X).
```

A lo que PROLOG responderá:

```
X=200.
```

Ejemplo 2. El siguiente ejemplo determina si un elemento X está en una lista.

Para facilitar la explicación del ejemplo se numeraron las cláusulas del programa, pero esto no forma parte del lenguaje PROLOG.

Se define el predicado elemento_de_lista que tiene dos argumentos, el primero es el elemento a buscar 'X', y el segundo es la lista en la cual se realizará la búsqueda del elemento X.

Lo primero que se verifica es si X es la cabeza de la lista y de esto se encarga la cláusula (1). La cola de lista se puso como variable anónima porque no interesa el valor al cual se acota.

Lo segundo es buscar si X es elemento de la cola de la lista y esto se logra con la cláusula (2). En este caso no interesa el valor de la cabeza de la lista, razón por la cual usamos una variable anónima.

Este ejemplo es especialmente importante porque sirve para mostrar la recursión que puede hacerse en PROLOG.

```
1   elemento_de_lista(X,[X|_]) :- .
2   elemento_de_lista(X,[_|Y]) :- elemento_de_lista(X,Y).
```

Si preguntamos:

```
?- elemento_de_lista(2,[3,4,5,2]).
```

El sistema responderá:

SI

y si formulamos la pregunta:

```
?- elemento_de_lista(2,[1,3,5]).
```

responderá:

NO

2.1.4 Mecanismos usados por PROLOG para la ejecución de un programa

Unificación

Este mecanismo verifica la igualdad entre términos y encuentra sustituciones de términos para las variables no acotadas (ie. las instancia[2]) con el fin de obtener

---

[2] Instanciación es un término usado cuando nos referimos a alguna variable. Una variable está instanciada cuando existe un objeto al que se encuentra acotada. Si no es así se le denomina no instanciada.

expresiones idénticas entre sí. Se debe evitar que una variable sea unificable con un término en el cual ocurre esa variable, para que no se presente el caso de que se puedan crear estructuras cíclicas. La versión original del algoritmo de unificación, tomado desde los probadores automáticos de teoremas, contenía la verificación de ocurrencias ("occur check"), pero muchas versiones de PROLOG ya no lo hacen, por consideraciones de incremento en tiempo del algoritmo.

¿Qué pasa cuando la unificación falla?

Cuando la unificación de una llamada con la cabeza de la primera regla es exitosa, la primera cláusula ejecuta su cuerpo (si tiene), y cuando la unificación falla, todos los efectos se deshacen. Esto es, todas las variables que fueron instanciadas al intentar la unificación son restablecidas a sus estados originales y la llamada se verifica entonces con la cabeza de la segunda cláusula. Si tiene éxito, la segunda cláusula se ejecuta, de otra forma se intenta la tercera cláusula, y así sucesivamente. Ejecutar un procedimiento consiste entonces en ejecutar la primera de sus cláusulas cuya cabeza concuerde con la llamada.

Retroceso

Retroceso, como se dice en [CLOW81], cercanamente se

asemeja a nuestro comportamiento en una búsqueda de solución para un problema. Si caemos en un callejón sin salida regresamos al punto más cercano en el cual podamos aplicar otra alternativa. En PROLOG, al comportamiento de intentar repetidamente satisfacer y resatisfacer metas que forman parte de una conjunción se le conoce como retroceso.

Cut

CUT es un procedimiento (escrito como '!') que es un mecanismo especial usado por los programadores para decirle a PROLOG cuáles elecciones previas no necesitan ser consideradas otra vez, cuando ocurre retroceso en la secuencia de metas (o llamadas a procedimientos) a ser satisfechas. Clocksin [CLOW81] señala que hay dos razones por las cuales es importante hacer esto:

1. El programa opera más rápido porque no tiene que perder tiempo intentando satisfacer metas que de antemano nunca contribuirán a la solución

2. El programa ocupa menos espacio en memoria porque puede lograrse un uso económico de la misma si no tienen que registrarse puntos de retroceso para examinación posterior.

El efecto de un simbolo CUT es el siguiente: cuando se

encuentra un CUT como meta en el cuerpo de una cláusula, su ejecución causará que el sistema dé por terminada la búsqueda de alternativas para todas aquellas cláusulas entre la cabeza de la que inició toda la ejecución y el CUT. Por lo tanto, todo intento de resatisfacer cualquier meta entre la meta padre y la meta CUT fallará.

2.1.4.1 Ejemplo de la ejecución de un programa

```
gusta(maria,chocolates)
gusta(juan, cerveza).
gusta(maria,vino).
gusta(juan,vino).
?- gusta(maria,X) , gusta(juan,X)
```

Para ejecutar la conjunción de metas en PROLOG se intenta satisfacer cada meta, trabajando de izquierda a derecha, la primera meta a ser satisfecha en este caso es gusta(maria,X), la cual unifica con la primera cláusula que se encuentra en la base de datos asi X se instancia a chocolates y PROLOG deja una marca asociada a esa cláusula. Entonces intenta satisfacer la siguiente meta a la derecha gusta(juan,X) empezando desde el inicio de la base de datos. Como PROLOG no puede satisfacer la meta gusta(juan,chocolates), PROLOG intenta resatisfacer la meta inmediata a la izquierda, comenzando la búsqueda en la base de datos a partir de la marca y además desacotando las variables que habia instanciado, esto es, X que estaba acotada al valor chocolates queda libre y trata de probar

nuevamente gusta(maria,X). Esta meta tiene éxito con la tercera cláusula. de la base de datos y X se acota a valor vino, con lo cual ahora se trata de probar gusta(juan,vino) y esta meta es tratada de satisfacer haciendo una búsqueda desde el inicio de la base de datos. Como la meta es satisfecha PROLOG pone una marca en la cuarta cláusula para el caso de que se tenga que resatisfacer la meta. Sin embargo el ejemplo termina porque solo pedimos que se encontrara una X que le gusta tanto a María como a Juan en este caso es el vino.

2.2 FUNDAMENTACION MATEMATICA

2.2.1 Programación lógica

La lógica estudia las interrelaciones surgidas de las implicaciones entre suposiciones y conclusiones [DEYL84]. Fue diseñada como el medio formal para representar el lenguaje natural y el razonamiento humano. Originalmente fue concebida como una herramienta para representar proposiciones permitiendo que fuera posible determinar formalmente si eran válidas. Así, se puede usar la lógica para expresar proposiciones, la relación entre ellas y determinar cómo se pueden inferir, válidamente, algunas proposiciones con base en otras que han sido previamente determinadas (prueba de teoremas). Debido a su origen en la representación de lenguaje natural, ha sido ampliamente usada en la especificación de lenguajes para programas de computadora, y como base fundamental en el desarrollo de lenguajes para la interrogación de bases de datos. Al hacer una analogía entre la lógica y lo que pretenden ser los sistemas de bases de datos, éstos resultan una especie de sistemas de propósito general para responder preguntas (o solución de problemas) en los que el conjunto de hechos necesarios para responder las preguntas puede ser visto como los axiomas de un teorema. Esta es la razón por la que la lógica puede llegar a jugar un papel tan importante dentro del área de los sistemas manejadores de bases de datos,

permitiendo la respuesta a cualquier clase de preguntas sobre información contenida en la base de datos.

2.2.2 El cálculo de predicados de primer orden

Una forma particular de la lógica es la llamada cálculo o lógica de predicados de primer orden. En ésta los objetos lógicos son representados por términos y se usan fórmulas para representar las relaciones entre ellos. Para que tales representaciones estén bien formadas deben cumplir con las siguientes definiciones.

Términos. (Los términos se definen recursivamente)

1. Cualquier constante es un término.
2. Cualquier variable es un término.
3. Si f es un símbolo de función n-ario, y t1,...,tn son términos, entonces f(t1,...,tn) es un término.
4. Todos los términos son generados aplicando las tres reglas anteriores.

Fórmulas atómicas. Si P es un símbolo de predicado n-ario, y t1,...,tn son términos, entonces P(t1,...,tn) es una fórmula atómica. Ninguna otra expresión puede ser fórmula atómica.

Fórmulas bien formadas. Las fórmulas bien formadas (que por comodidad denotaremos como FBF) son definidas rescursivamente de la siguiente manera:

1. Una fórmula atómica es una FBF.

2. Si F y G son FBF, entonces ~F, F & G, F # G, F -› G, y F ‹-› G son FBF. Donde ~, & , # , -› y ‹-› son los conectivos negación, conjunción, disyunción, implicación y equivalencia, respectivamente.

3. Si F es una FBF, y x es una varible libre, entonces (∀x)F y (∃x)F son FBF, donde ∀x y ∃x son los cuantificadores universal y existencial respectivamente.

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

2.2.3 Formas clausales

En una FBF un conectivo puede ser expresado a base de otros conectivos de acuerdo con las siguientes equivalencias.

```
F <-> G      == (F -> G) & (G -> F)
F -> G       == ~F # G
~~P          == P
~(F & G)     == ~F # ~G
~(F # G)     == ~F & ~G
(∃x)P     == ~((∀x)(~P))
(∀x)P     == ~((∃x)(~P))
```

Como resultado de estas equivalencias se puede concluir que un conjunto funcionalmente completo de conectivos (para el cálculo de predicados) suficiente para expresar cualquier fórmula en una forma equivalente podría ser (&, ~, ∀). Sin embargo, como resultado de la redundancia de conectivos, existen muchas maneras diferentes de expresar la misma fórmula en una forma equivalente. (Por cierto cabe señalar que éste es uno de los problemas que se presentan en los lenguajes para interrogación de bases de datos). Esta característica se torna en una fuerte inconveniencia cuando se desea llevar a cabo manipulaciones formales usando fórmulas del cálculo de predicados, ya que sería más sencillo si todo lo que se desea decir pudiera ser expresado en forma única. Con esta restricción en mente consideremos la forma clausal, en la que existen menos maneras de decir lo mismo debido a que se tiene un conjunto mínimo de conectivos.

Formas clausales. Una cláusula es una expresión de la forma

B1, B2, ..., Bm ‹- A1, A2, ...,An

donde B1, B2,..., Bm, A1, A2,..., An son fórmulas atómicas, n≥0 y m≥0. Las fórmulas atómicas A1, A2, ...,An son las condiciones conjuntivas de la cláusula y B1, B2,..., Bm son las conclusiones alternativas. Si la cláusula contiene las variables x1, x2,..., xk entonces debe interpretarse como si estableciera que para todas las x1, x2,..., xk, debe ser cierta alguna de las conclusiones B1 o B2 o ... o Bm, si todas las A1, A2, ... , An son ciertas. Se presentan tres casos especiales de estas formas clausales:

1. Si n es cero se tiene,

B1, B2, ..., Bm ‹-

entonces para toda x1, ..., xk, alguna de las B1 o B2 o ... o Bm es incondicionalmente cierta.

2. Si m es cero se tiene,

‹- A1, A2, ..., An

entonces para toda x1, ..., xk, no es el caso que A1 y A2 y ... y An sea cierta.

3. Si ambos, m y n son cero, se tiene la cláusula vacia,

<-

entonces se tiene que esta fórmula siempre falla: es una contradicción.

2.2.4 Cláusulas de Horn

Tratando de determinar cuál es el subconjunto de la lógica matemática que puede ser representado directamente en PROLOG, llegamos a las cláusulas que contienen cuando más una sola conclusión y que son conocidas como cláusulas de Horn debido a que Alfred Horn fue el primero que investigó este tipo de cláusulas en 1951.

Las cláusulas de Horn se clasifican en:

1. Cláusulas de Horn con cabeza

Son cláusulas que sólo tienen una conclusión, a la que se le da el nombre de cabeza de la cláusula. Se conocen dos formas de cláusulas de Horn con cabeza:

```
B <- A1, A2, ..., An
B <-
```

En ambos casos B es la cabeza de la cláusula.

Las primeras son conocidas como reglas de

inferencia en terminología de PROLOG, y las segundas son conocidas como hechos.

2. Cláusulas de Horn sin cabeza.

Son aquelllas que no tienen conclusión alguna. Se conocen dos formas de cláusulas sin cabeza.

‹- A1, A2, ..., An

cláusula vacía representada como ‹-

En PROLOG la primera es la meta a resolver, y la segunda se obtiene cuando no se encuentra una solución al problema que se trata de resolver.

Cualquier problèma soluble que pueda ser expresado en cláusulas de Horn, tiene que ser expresado en tal forma que:

1. Se tenga una cláusula de Horn sin cabeza.

2. El resto de las cláusulas sean cláusulas de Horn con cabeza.

¿Por qué debe haber al menos una cláusula sin cabeza para que el problema sea soluble?

Esto se debe a que el resultado de resolver dos cláusulas con cabeza es a su vez otra cláusula con cabeza. Sin embargo, si se tiene una cláusula de Horn sin cabeza se

aplica el principio de resolución (véase apartado 2.3.1) el cual tratará de encontrar una solución al problema utilizando la estrategia de búsqueda por profundidad ("depth first").

En PROLOG el tipo de cláusulas que se pueden representar directamente son las cláusulas de Horn. Esto no es restrictivo debido a que se puede pasar de fórmulas del cálculo de predicados a cláusulas de Horn, con lo cual podemos tener en PROLOG la representación de la lógica de primer orden con restricciones de las operaciones lógicas como la negación (la cual está definida en forma parcial[3]) y la conjunción (and) y disyunción (or), que tienen un peculiar significado por razones de eficiencia (véase el apartado 3.2).

Otra diferencia de PROLOG con respecto a la lógica de primer orden es que en PROLOG no se contempla la separación de los símbolos de función y de los símbolos de predicado[4].

El camino a seguir para pasar de fórmulas del cálculo de predicados a cláusulas de Horn se puede resumir en los

---

[3] La negación en PROLOG está definida como fracaso de la regla, esto es, not(P) es verdadera si P es falsa y falso en otro caso.

[4] El símbolo de función es usado para construir los argumentos, y los símbolos de predicado son usados para construir proposiciones.

siguientes pasos:

1. Pasar de fórmulas del cálculo de predicados a formas clausales.

2. Pasar de formas clausales a cláusulas de Horn.

La transformación de fórmulas del cálculo de predicados a formas clausales se hace utilizando un algoritmo de Davis Martin [CLOW81].

El paso de formas clausales a cláusulas de Horn es inmediato. Como ya vimos, las formas clausales tienen la siguiente forma:

B1, B2, ..., Bm <- A1, A2, ... , An

La transformación a cláusulas de Horn se logra estableciendo cada una de las conclusiones por separado y evaluando si alguna de las Bi es cierta suponiendo que todas las Ai son ciertas, esto es:

```
B1 <- A1, A2, .... , An
B2 <- A1, A2, .... , An
.
.
.

Bm <- A1, A2, .... , An
```

De lo anterior se concluye que el poder de representación de PROLOG no se limita únicamente a cláusulas de Horn como podría pensarse en el primer momento, sino que

se pueden representar fórmulas del cálculo de predicados adecuándolas al tipo de cláusulas que se pueden reconocer directamente, como ya vimos anteriormente.

El siguiente diagrama (figura 2.1) permite visualizar la relación entre el poder de las diferentes herramientas de la lógica matemática que se pueden usar en la programación lógica.

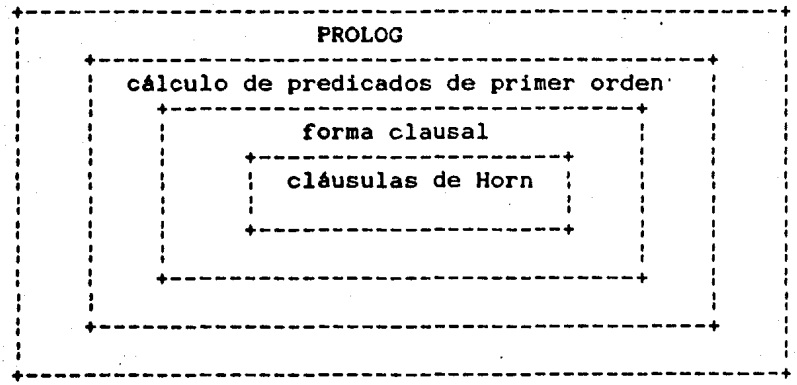

Figura 2.1 Poder de representación de PROLOG

2.3 El LENGUAJE PROLOG Y LA LOGICA

A continuación se tratará de relacionar la lógica con PROLOG, para lo cual primero se explicarán los principios en los que se basa la prueba automática de teoremas y cómo fueron usados en PROLOG, ya que en un principio este lenguaje fue ideado como una herramienta para la prueba de teoremas.

2.3.1 Los principios de resolución y prueba automática de teoremas

Hasta el momento hemos visto la manera de representar los predicados lógicos usando el cálculo de predicados, la forma clausal y las cláusulas de Horn. El siguiente paso será encontrar qué se puede hacer con dichas cláusulas. En el contexto de la lógica la siguiente acción a tomar una vez que se tiene un conjunto de proposiciones es investigar qué se puede concluir de ellas y determinar si la conclusión obtenida es de alguna utilidad o no. Llamaremos axiomas a aquellas proposiciones que tomaremos como verdaderas con el propósito de probar algún argumento, y se llamarán teoremas a las proposiciones que se concluyen de los axiomas.

En la década de los 60 se empezó a investigar si las computadoras podían ser programadas para realizar la prueba automática de teoremas. Fue entonces cuando J. Alan Robinson [ROBJ65] descubrió el principio de resolución y su

aplicación a la prueba automatizada de teoremas. El principio de resolución es una regla de inferencia que dice cómo puede ser concluida una proposición de otras. Usando este principio se puede lograr la prueba de teoremas en forma totalmente mecánica, basándose en axiomas previamente establecidos. Lo único que se tiene que decidir es cómo escoger las proposiciones que se aplicarán, después de lo cual las conclusiones serán producidas en forma automática.

El principio de resolución fue diseñado para trabajar con fórmulas en la forma clausal. La idea básica es que si una fórmula atómica aparece en el lado derecho de una cláusula y el izquierdo de otra, entonces la cláusula obtenida al adecuar las dos anteriores, pero eliminando la fórmula duplicada, es una conclusión válida de ellas. Para aclarar este concepto diremos que en realidad las fórmulas atómicas no tienen que ser idénticas, sólo necesitan poderse igualar. Además, la cláusula que se concluye de las dos primeras es obtenida realizando la instanciación de las variables que aparezcan en la fórmula común para que lleguen a ser idénticas.

El principio de resolución tiene dos propiedades muy importantes que PROLOG usa. La primera es la refutación completa. Esto significa que si un conjunto de cláusulas es inconsistente entonces lo único que el principio de resolución podrá derivar de ellas será la cláusula vacia.

<-

Y dado que el principio de resolución también es correcto sólo será capaz de derivar la cláusula vacía en esta circunstancia. Un conjunto de fórmulas es inconsistente si no existe una posible interpretación para los predicados, símbolos de constantes y de funciones que los haga expresar proposiciones verdaderas en forma simultánea. Recordemos que la cláusula vacía es la expresión lógica de la falsedad, esto es, que no hay posiblidad de que sea cierta. Así, PROLOG o el principio de resolución garantizan que se indicará que un conjunto de fórmulas es inconsistente al derivar esta clara expresión de contradicción, sólo que PROLOG no imprimirá la cláusula vacía sino un mensaje que diga FALSO.

Finalmente aunque el principio de resolución dice cómo derivar una consecuencia de dos cláusulas, no dice cómo decidir qué cláusulas deberán ser las siguientes a tratar de probar o cuáles literales se deberán igualar. Generalmente, si se tiene un gran número de axiomas, entonces habrá muchos caminos de solución a intentar para cada una de ellas. Además cada vez cada vez que derivemos una nueva cláusula, se convierte en un nuevo candidato a tomar parte en las subsecuentes resoluciones. La mayor parte de las

posibilidades serán irrelevantes, por lo que si no se tiene cuidado al escoger, se pasará mucho tiempo en ellas antes de encontrar si hay o no solución. Fue así que las cláusulas de Horn aparecieron para dar una solución viable a este problema y restringir la representación a una forma más sencilla. Si se está usando un probador clausal de teoremas para determinar los valores de las funciones computables, sólo será necesario usar cláusulas de Horn para su representación.

2.3.2 El uso del principio de resolución en PROLOG

La sintaxis usada en PROLOG para la representación de cláusulas sustituye el uso de los conectivos ←, ^ y v por :- , & y # respectivamente, sin embargo esto varía de una a otra realización. Otra peculiaridad del lenguaje es que sólo se pueden representar cláusulas de Horn en forma directa, todo otro tipo de cláusulas tendrá que ser convertido a la forma de Horn antes de poderlas representar en PROLOG (véase sección 2.2.4).

Ya se ha visto que para cualquier problema que se desee resolver usando cláusulas de Horn es suficiente tener exactamente una cláusula sin cabeza. En PROLOG la situación es la misma: todas las cláusulas del programa tienen cabeza y sólo una meta (o cláusula sin cabeza) es considerada (ejecutada) a la vez. La sintaxis de la meta en la forma de Horn y PROLOG se muestra a continuación:

Horn:

:- A1, ... ,An.

PROLOG:

?- A1, ... ,An.

PROLOG está basado en un probador de teoremas, fundamentado en el principio de resolución, para cláusulas de Horn. En PROLOG el principio de resolución es simulado haciendo llamadas a las cláusulas y verificando si la llamada puede ser igualada con el lado izquierdo de alguna fórmula previamente definida; si se pueden igualar se continúa la ejecución del lado derecho de la nueva fórmula hasta determinar si ésta es falsa o cierta, momento en el que se regresa al punto donde fue invocada. La estrategia que usa para solucionar un problema es una forma restringida de la resolución lineal en la que la selección de cláusulas con las que se quiere resolver un problema se realiza de la siguiente manera. Primero se toma la meta a resolver, de ella se selecciona la primer fórmula atómica, de izquierda a derecha, a tratar de igualar, y se busca la primera hipótesis que tenga dicha fórmula del lado izquierdo para tratar de obtener una nueva cláusula insertando el cuerpo de la hipótesis en el lugar de la fórmula atómica igualada. Ahora se prosigue tratando de igualar, de izquierda a derecha, las fórmulas de la hipótesis recién insertadas y se

sigue así para cada paso, como se muestra en el siguiente ejemplo:

Notación usada en el ejemplo

es nombre del predicado hecho por el programador para ejecutar la disyunción lógica.

& es el predicado interconstruido que efectua una conjunción lógica.

```
#(X,_) :- call(X).
#(_,X) :- call(X).
madre(juan, maria).
madre(leticia, maria).
mujer(maria).
mujer(leticia).
hombre(juan).
hermano(X,Y):- hombre(Y) & madre(Y,Z) &
               #(hombre(X) , mujer(X)) & madre(X,Z).

?-hermano(leticia,juan) &
          write(" juan si es hermano de leticia").
```

usando resolución e igualación con instanciación de variables se obtiene la nueva meta,

```
?- hombre(juan) & madre(juan,Z) & (hombre(leticia) ;
   mujer(leticia)) & madre(leticia,Z) &
   write(" juan si es hermano de leticia").
```

y se continúa resolviendo hasta obtener una respuesta de si la meta es cierta o falsa

Esta forma de organizar la selección de las cláusulas alternativas para la solución de una meta es conocida como estrategia de búsqueda por profundidad ("depth first") y lo

que hace es considerar sólo una alternativa a la vez usando este orden preestablecido, y únicamente permite considerar la siguiente alternativa cuando la primera ha fallado. Sin embargo esta estrategia de selección es susceptible de llevar al sistema a caer en ciclos de selecciones, ya que no se lleva control de posibles ciclos recursivos infinitos en la selección. Estos no permitirían considerar nunca algunas de las hipótesis, por lo que al momento de programar hay que tener cuidado de que esto no suceda. El motivo por el que no se lleva control de estos ciclos recursivos es por sus grandes requerimientos en cuanto al uso de espacio y tiempo para realizar el control de la selección de alternativas.

2.4. CONSIDERACIONES SEMANTICAS DE LOS PROGRAMAS

Los programadores de aplicaciones en PROLOG deben saber que PROLOG, a diferencia de los lenguajes de programación convencionales, tiene dos formas de entender su semántica: como semántica declarativa o como semántica procedural [COEH82].

La semántica declarativa de PROLOG que heredó de la lógica define cuando una meta es cierta con respecto a un programa dado, y si es cierto, para qué instanciación de variables es cierta.

Características de la semántica declarativa según la lógica:

1. El significado declarativo de los programas en PROLOG no depende del orden de las cláusulas ni del orden de las metas en las cláusulas.

2. El aspecto declarativo de PROLOG es responsable de promover una programación clara, rápida y precisa. Permite que un programa sea dividido en partes y permite cierto entendimiento de un programa sin los detalles de cómo es ejecutado.

La semántica procedural es una restricción a la semántica declarativa, se debe a la forma en que esta hecho el probador de teoremas ya que utiliza la estrategia de búsqueda primero por profundidad "depth first" (véase

apartado 2.3.1), a las restricciones en las operaciones lógicas (véase apartado 3.7) y también a la forma de procesamiento de las cláusulas de PROLOG que es secuencial.

La semántica procedural de PROLOG especifica cómo contesta PROLOG las preguntas. Para contestar una pregunta se trata de satisfacer el conjunto de metas que forman la pregunta. Estas pueden ser satisfechas si las variables que ocurren dentro de las metas pueden ser instanciadas en tal forma que las metas se infieran lógicamente del programa.

Características de la semántica procedural:

El significado procedural depende del orden de las metas y cláusulas. El orden puede afectar la eficiencia del programa, de tal manera que un orden no apropiado puede llevar a llamadas recursivas infinitas (véase 3.7).

2.4 CARACTERISTICAS SOBRESALIENTES DE PROLOG

1. Una semántica declarativa heredada de la lógica además de la semántica procedural usual.

2. En PROLOG no hay distinción entre lo que es programa y lo que son los datos. Las cláusulas pueden ser empleadas para expresar datos y pueden ser manipuladas como términos por intérpretes escritos en PROLOG.

3. La entrada y salida de argumentos de un procedimiento no tiene que ser distinguida por adelantado, sin embargo pueden variar de una llamada a otra. Los procedimientos pueden usarse para múltiples propósitos.

4. Los procedimientos pueden tener múltiples salidas así como múltiples entradas.

5. Permite retroceso (véase apartado 2.1.4)

6. La definición de objetos compuestos permite definir estructuras de cualquier tamaño y sin restricciones en el tipo de los campos.

7. La unificación de patrones reemplaza el uso de funciones de selección y construcción para operar datos estructurados.

8. Después de la evaluación de un procedimiento se pueden regresar parámetros libres los cuales pueden ser acotados más adelante por otros procedimientos.

9. PROLOG no maneja GO TO, DO, FOR, WHILE ni apuntadores.

10. La semántica procedural de un programa sintácticamente correcto está totalmente definida. Es imposible que una condición de error o que una operación no definida sean ejecutadas. Esta semántica totalmente definida asegura que los errores de programación no resultarán en programas con comportamiento extraño o mensajes de error incomprensibles.

11. El procesamiento de las cláusulas en PROLOG es secuencial.

2.4 ALGUNOS USOS ESPECIFICOS DE PROLOG

PROLOG ha sido usado principalmente para resolver problemas de inteligencia artificial. Algunas de las aplicaciones que se han desarrollado hasta este momento, se describen en [SANE82]:

1. Cálculo de las n primeras derivadas de una función real

   La entrada a este programa es una función real de varias variables de complejidad alta. El programa genera las n primeras derivadas por diferenciación simbólica. Fue hecho en 1979 en Hungria.

2. Entendimiento del lenguaje natural

   Se trata de un subconjunto mínimo del lenguaje natural que puede ser utilizado para crear y consultar bases de datos. Se tienen varios sistemas para consultar bases de datos en diferentes idiomas.

   Para inglés se tiene el sistema de Warren y Pereira hecho en 1981. Para portugues se tiene el sistema hecho por Coelho en 1979. Todos estos sistemas están programados en PROLOG.

3. Planeación de un edificio para talleres de un nivel usando paneles prefabricados.

Este programa fue la primera aplicación de PROLOG que se tuvo en Hungría. La entrada a este programa son los datos de los elementos prefabricados disponibles (tamaño geométrico, peso neto, fuerza de soporte) estos son dados en forma de aserciones del programa. El programa determina el plano base y elige los elementos más apropiados desde el punto de vista de la geometria y de las condiciones estáticas.

4. Un sistema de información para el control de la contaminación del aire

Sistema interactivo para el control de la contaminación del aire. Toma datos acerca de la concentración de industrias contaminantes en Budapest y en ciudades de Hungría. El sistema determina si la contaminación del aire producido por las plantas está dentro del nivel permitido, si no, calcula la altura de la chimenea necesaria para reducir la concentración y además recomienda los equipos de filtrado industrial apropiados para la rama de la industria en cuestión.

CAPITULO 3

DESCRIPCION DEL INTERPRETE

3.1 GRAMATICA SELECCIONADA

El primer paso en la realización del intérprete fue la definición de la gramática de PROLOG, para definirla se llevaron a cabo dos pasos, el primero fue determinar como se define la gramática para cualquier lenguaje; el segundo fue encontrar los componenentes de la gramática de PROLOG en base al punto anterior. Como resultado del primer paso se tiene que una gramática [HOPJ69] se define como una cuarteta G = (VN,VT,P,S) donde

VN es el conjunto de símbolos no terminales.
VT es el conjuno de símbolos terminales.
P es el conjunto de producciones.
S es el simbolo inicial de la gramática.
VN intersección VT es igual al conjunto vacio y
VN unión VT es igual a V donde V es el vocabulario del lenguaje.
P consta de expresiones de la forma $\underline{a} \rightarrow \underline{b}$
donde $\underline{a}$ es una cadena de V^+ y $\underline{b}$ es una cadena de V^*

Para desarrollar el segundo paso se consultó el libro de Clocksin [CLOW81] del cual se obtuvieron los símbolos terminales, se hizo un estudio para determinar las construcciones válidas dentro del lenguaje PROLOG. Con lo cual se pudieron determinar las reglas de producción y los símbolos no terminales, uno de los cuales se escogió como símbolo inicial.

Debido a que para el analisis sintáctico se utilizó Yacc [JHOS81], en la definición de la gramática se usó la siguiente notación: los simbolos terminales se escriben en letra mayúscula y los no terminales con letra minuscula, a la inversa que en la notación de gramáticas; el simbolo ":" reemplaza al simbolo -› usado en la notación de gramáticas y se usa "¦" para separar alternativas correspondientes al mismo lado izquierdo.

El principal problema con el que se encontró una vez que ya se habia definido la gramática de PROLOG es que es ambigua[1]. Algunas de las reglas para reconocer dicha gramática son del tipo:

```
1    A : B

2    A : B  ":-" C

3    B : D

4    C : D

5    D : Ei

6    Ei : Ei-1 OP Ei-1

7    Eo : t1 ¦ t2¦ ...¦t5
```

[1] es cuando para una oración generada por la gramática existe más de un árbol sintáctico para ella.

8 OP : & ¦ :- ¦ # ¦ ... ¦ ¦

Como puede verse en el siguiente ejemplo la gramática es ambigua ya que la cadena a :- b se puede reconocer tanto con la regla (1) como con la regla (2) (véase figura 3.1).

Para evitar estos problemas se decidió eliminar el manejo dinámico de las precedencias para los operadores y se usó una notación especial para el las operaciones que manipulan la base de datos (inserción y supresión en la base de datos). Por el momento sólo se manejan los operadores en forma funcional en esta primera versión del intérprete.

A continuación se muestra la gramática de PROLOG dividida en los cuatro componentes de la definicón formal de gramática.

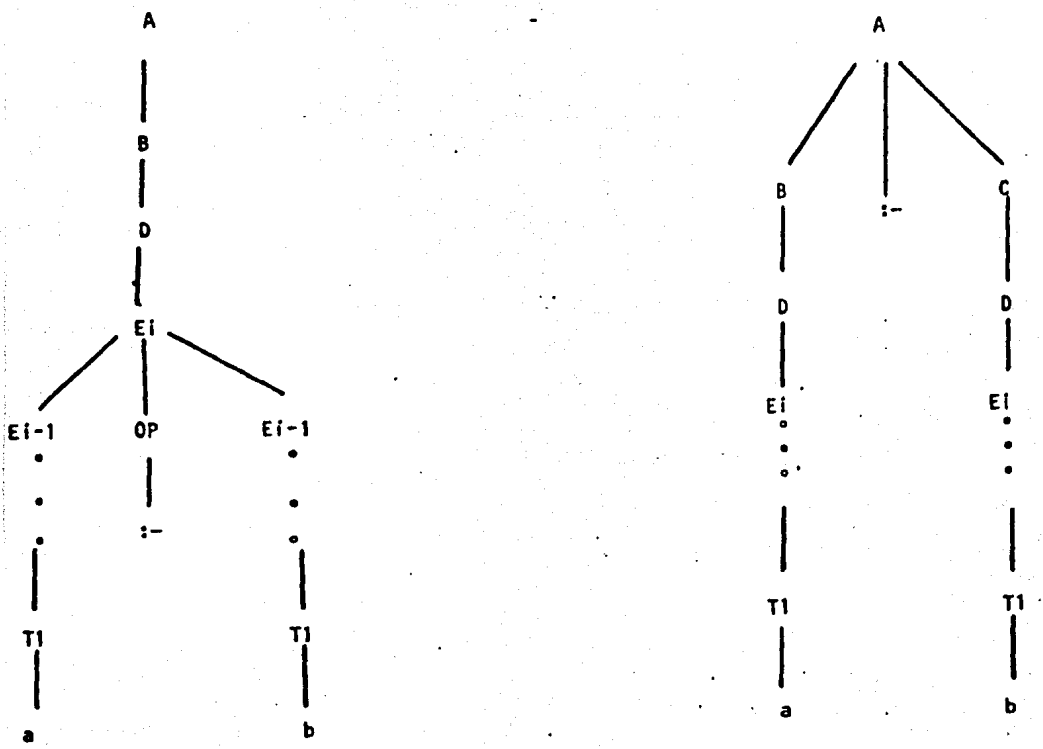

Figura 3.1 Arbol sintáctico para reconocer la cláusula a:- b

```
/* Simbolo inicial de la gramática */
clausula

/* Conjunto de simbolos terminales */

PUNTO                : "."
                     ;

META                 : ":-"
                     ;

AMP                  : "&"
                     ;

ABRED                : "("
                     ;

CIERED               : ")"
                     ;

PREG                 : "?-"
                     ;

VARNA                : ‹mayúscula› ‹alfanumérico›*
                     ;

VARAN                : "_"
                     ;

NUMERO               : "+" ‹entero› | "-" ‹entero›
                     ;

NOMFUNCTOR           : ‹minúscula›* | operador

operador             :  "+" | "-" | "*" |"/" |"#" | is | "!"
                     ;

LISTAVACIA           : "[]"
                     ;
```

```
BARRA                   : "|"
                        ;

ABCUAD                  : "["
                        ;

CCUAD                   : "]"
                        ;
```

```
/* conjunto de simbolos no terminales */

clausula
definicion
clausula_no_unitaria
clausula_unitaria


cabeza
cuerpo
novarent
lista_lla


llamada
termino
pregunta
lista
lista_de_terminos
elem_lista
```

```
/* Conjunto de reglas de producción */

clausula                : definicion
                        | pregunta
                        | error PUNTO
                        ;

definicion              : clausula_no_unitaria
                        | clausula_unitaria
                        ;
```

```
clausula_no_unitaria : cabeza META cuerpo PUNTO
                     ;

clausula_unitaria    : cabeza PUNTO
                     ;

cabeza               : novarent
                     ;

cuerpo               : lista_lla
                     ;


lista_lla            : llamada
                     | lista_lla AMP llamada
                     ;

llamada              : novarent
                     | ABRED cuerpo CIERED
                     ;


novarent             : termino
                     ;

pregunta             : PREG cuerpo PUNTO
                     ;


termino              : VARNA
                     | VARAN
                     | NUMERO
                     | lista
                     | nofun
                     | nofun ABRED termino CIERED
                     | ABRED  termino CIERED
                     ;
```

```
nofun                : NOMFUNCTOR
                     ;

lista_de_terminos    : termino
                     | lista_de_terminos COMA termino
                     ;

lista                : LISTAVACIA
                     | abcuad elem_lista CCUAD
                     | abcuad  elem_lista BARRA termino CCUAD
                     ;

abcuad               : ABCUAD
                     ;

elem_lista           :  termino
                     |  elem_lista COMA termino
                     ;
```

3.2 ANALIZADOR LEXICOGRAFICO

Se encarga de leer código escrito en PROLOG y obtener tokens de los tipos que se pueden reconocer dentro de la gramática de PROLOG. Se estudió la posibilidad de construir el analizador lexicográfico utilizando una herramienta del sistema operativo Unix llamada Lex, que sirve para generar automáticamente analizadores lexicográficos; para esto se definió una especificación que consiste en expresiones regulares con acciones para cada una de ellas. Finalmente se decidió escribir el analizador lexicográfico directamente en C, debido a razones de eficiencia. Esto es porque Lex, la herramienta de Unix, construye un reconocedor lento, basado en tablas y un programa autómata que sigue la tabla. Y dado que durante el desarrollo del intérprete (que implica pruebas y rediseño de algunas reglas de producción) los tokens no tienen la misma probabilidad de variar en sus definiciones como lo hacen las reglas para el análisis sintáctico se decidó que no era necesario usar Lex y si Yacc.

El método que se usó para reconocer los tokens fue a través de diagramas de transición [AHOA77], los cuales son una forma de representar autómatas. Una vez definidos los diagramas se pudo proceder, en forma inmediata, a su programación.

Los tipos de tokens que reconoce el analizador lexicográfico desarrollado son los siguientes:

LISTA DE TOKENS

Variable anónima:

VARAN	"_"

Variable no anónima:

VARNA	‹mayúscula›‹alfanumérico›*.

Número entero positivo o negativo:

NUMERO	‹+¦-›‹entero›

Nombre de un functor:

NOMFUNCTOR	‹minúscula›* ¦ ‹operador›
‹operador›	‹+¦-¦*¦/¦!¦#¦is›
+ - * /	operadores aritméticos
!	cut
#	disyunción lógica
is	asignación

Símbolo para la terminación de una cláusula:

PUNTO	"."

Conjunción lógica:

AMP	"&"

```
Token para el manejo de errores:
        ERROR           ‹secuencia desconocida›

Separador entre la cabeza y cuerpo de las cláusulas:
        META            ":-"

Separador entre los elementos de las listas y
entre los argumentos de los functores:
        COMA            ","

Representación de la lista sin elementos:
        LISTAVACIA      []
Separador que identifica el inicio de una pregunta:
        PREG

Paréntisis para agrupar parámetros y elementos
de listas:
        ABRED           "("
        CIERED          ")"
        ABCUAD          "["
        CCUAD           "]"

Separador entre la cabeza y cola de una lista:
        BARRA           "¦"
```

3.3 ANALIZADOR SINTACTICO

Reconoce las construcciones del lenguaje descritas por la gramática anteriormente definida. Para el desarrollo de esta etapa se utilizó la herramienta de Unix llamada Yacc, que genera un parser ascendente ("bottom up"). Es importante mencionar que si Yacc no se conoce bien o no se tienen los manuales adecuados, en vez de convertirse en una herramienta útil se convierte en un obstáculo a vencer [JHOS81], [CHRK83]. Además en esta etapa también se genera la representación interna del programa en PROLOG. Dicha representación es generada a través de rutinas escritas en "C", las cuales son llamadas por Yacc en las reglas gramaticales adecuadas. El propósito de generar esta representación interna es lograr que el proceso de interpretación sea rápido al eliminar cualquier tipo de análisis durante la etapa posterior de ejecución.

Los problemas con los que se tropieza el que usa esta herramienta es que el tipo de gramática que acepta Yacc es del tipo LALR(1) ("lookahead Left to right"), y hay que adecuar la grámatica que se va a reconocer a este tipo.

El parser se puede programar directamente; sin embargo en este trabajo no se hizo así porque la idea era utilizar herramientas que se tienen como ayuda en el desarrollo de lenguajes y programas para computadora como Yacc, Lex y Make.

3.4 ESTRUCTURAS DE DATOS

Las estructuras de datos utilizadas para representar en forma interna los programas en PROLOG y llevar el control de su interpretación son una base de datos y un conjunto de pilas, que permiten "recordar" el ambiente que existe y el que se crea durante cada una de las llamadas a cualquiera de las cláusulas del programa

La organización fue diseñada de tal manera que cumpla con los requerimientos del módulo que se encarga del proceso de interpretación los cuales son:

a. Que sea sencillo agregar y suprimir cláusulas en la base de datos.

b. Que la interpretación se pueda hacer fácilmente.

c. Que facilite el manejo del retroceso.

3.4.1 Estructuras de datos creadas durante el análisis sintáctico

A continuación se explican las estructuras de datos que fueron ideadas con el propósito de cumplir con los requerimientos expuestos.

Dado que a PROLOG se le han agregado predicados extralógicos para insertar y suprimir cláusulas durante la

ejecución de los programas, fue necesario tener una representación que permitiera realizar estas operaciones en forma dinámica. Debido a esta necesidad, se decidió almacenar la representación o esqueleto[12] de cada cláusula usando memoria dinámica, a través de las rutinas de asignación dinámica de memoria que tiene la interface del lenguaje C con el sistema operativo Unix. Esta solución permite que sea sencillo agregar y suprimir cláusulas en la base de datos.

Para que la interpretación se pueda hacer fácilmente se propuso que las estructuras de datos permitieran acceso rápido a las cláusulas, lo que se logra a través de un diccionario de datos en el que la búsqueda de cualquier cláusula se hace por medio de una función de dispersión usando el método de la segunda dispersión para la resolución de colisiones.

El diccionario de datos contiene información de las cláusulas definidas en el programa, tal como el nombre del functor, la aridad y un apuntador a la serie de cabezas de lista (llamadas descriptores de cláusulas) que describen las

---

[12] Se denomina esqueleto a la representación de cualquier cláusula en la cual no existe instancia de variable alguna, es decir, ninguna variable ha sido acotada a algún objeto concreto. Cabe aclarar que el proceso de copia de alguna estructura hacia la pila de copias, se realiza justo en el momento en que se acota una variable a la estructura en cuestión.

cláusulas que tienen el mismo nombre y aridad.

El diccionario contiene los siguientes campos (véase figura 3.2):

APNOM - apuntador al nombre de la cláusula.

OCP - bandera que indica si el registro del diccionario esta o no ocupado.

ARIDAD - número de parámetros.

APROC - apuntador al descriptor de la cláusula.

En los descriptores de cláusulas se guarda el número de variables, la bandera para señalar si son rutinas del sistema o cláusulas definidas por el usuario, apuntador al siguiente descriptor de cláusulas, apuntador al esqueleto de la cabeza de la cláusula y el apuntador al esqueleto del cuerpo de la cláusula.

Se definieron dos tipos de descriptores de cláusulas

1. Para rutinas del sistema

2. Para cláusulas definidas por el usuario

Los descriptores para rutinas del sistema (véase figura 3.3) estan formados por los siguientes campos:

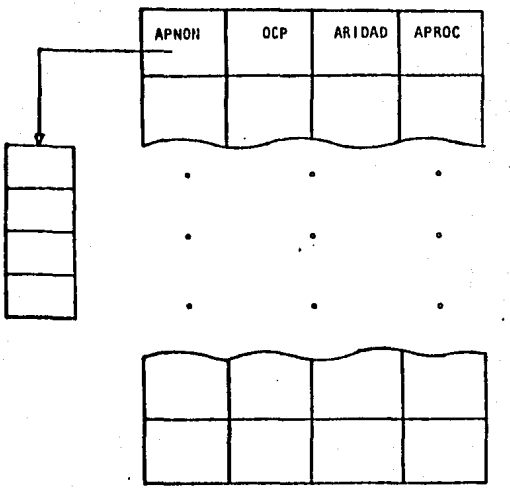

Figura 3.2 Diccionario de datos

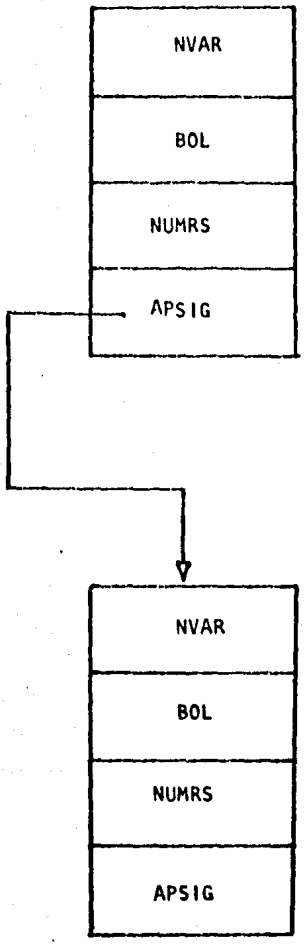

Figura 3.3 Descriptor para una rutina del sistema

NVAR - número de variables de la cláusula.

BOL - bandera que indica si es o no rutina del sistema.

NUMRS - número que le corresponde a esa rutina del sistema.

APSIG - apuntador al siguiente descriptor.

Los descriptores para cláusulas definidas por el usuario (véase figura 3.4) estan formados por los siguientes campos:

NVAR - número de variables de la cláusula.

BOL - bandera que indica si es o no rutina del sistema.

CABEZA - apuntador al esqueleto de la cabeza de la cláusula.

CUERPO - apuntador al esqueleto del cuerpo de la cláusula.

APSIG - apuntador al siguiente descriptor.

Las estructura de datos empleadas para describir los esqueletos de las cláusulas (véase figura 3.5)

consiste en listas ligadas, tipo LISP, en las que cada átomo

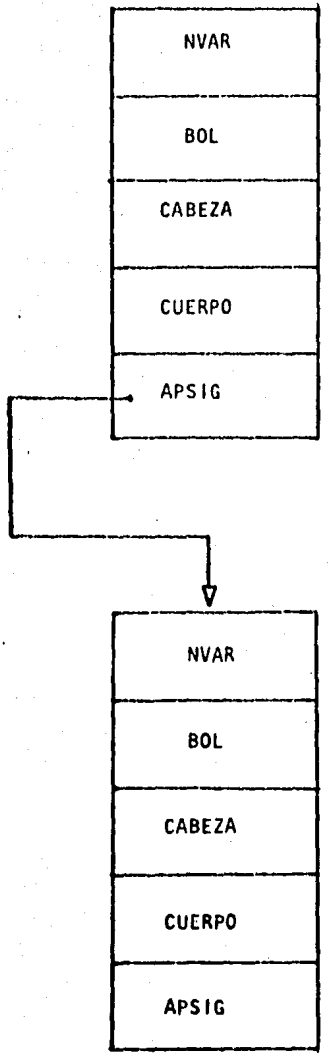

Figura 3.4 Descriptor para una cláusula definida por el usuario

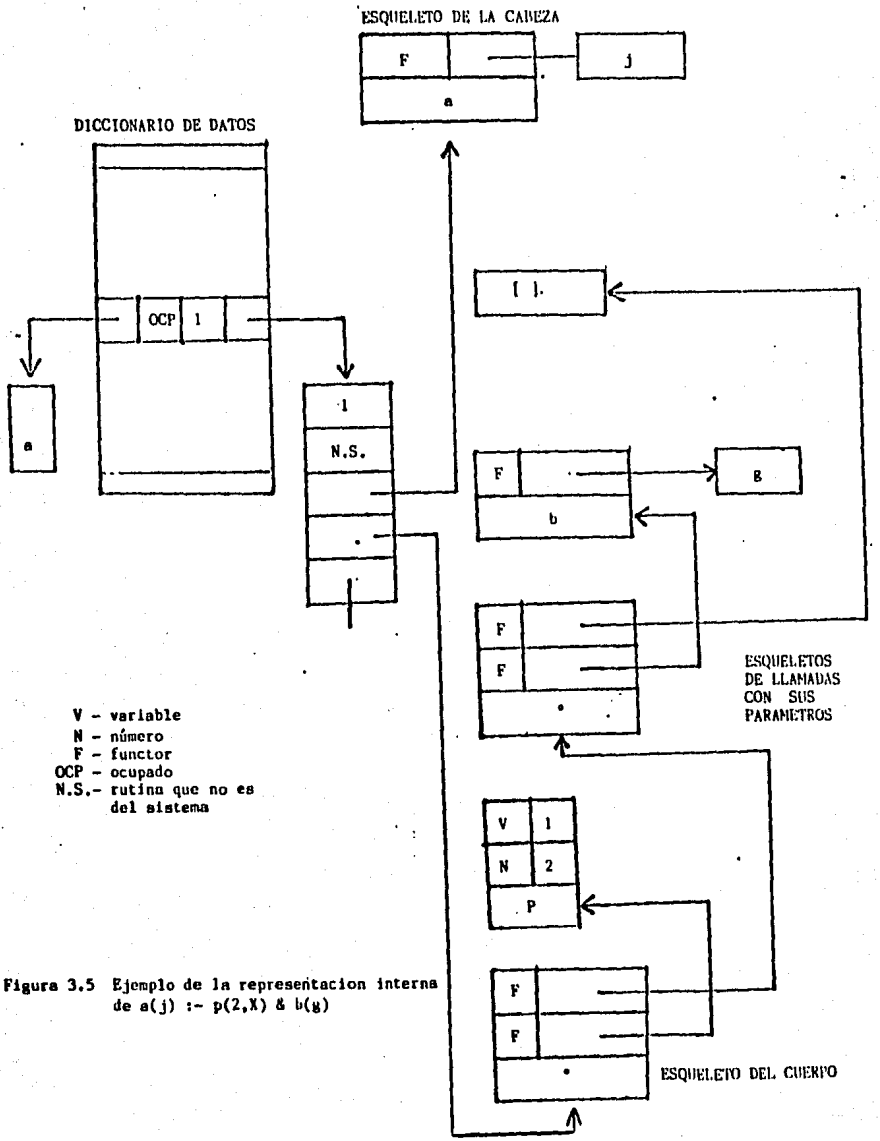

Figura 3.5 Ejemplo de la representacion interna de a(j) :- p(2,X) & b(g)

representa un functor con sus parámetros, o más propiamente, es el descriptor del functor con los descriptores de sus parámetros. El descriptor del functor es la dirección de éste en el diccionario de datos y los descriptores de los parámetros tienen la forma mostrada en la figura 3.6.

3.4.2 Estructuras de datos al tiempo de ejecución

Por otro lado también se necesitaba que la representación interna facilitara el control durante la interpretación, esto es, que fuera sencillo saber cuál llamada del programa se está ejecutando y cuáles variables se han modificado durante la ejecución de cada cláusula. Al igual que en otros lenguajes en los que se manejan variables locales y globales, esta información sólo se guarda mientras no haya terminado la ejecución de una cláusula. La diferencia con respecto a otros lenguajes es que si termina debido a que fracasó, la información es usada para realizar el retroceso restableciendo aquellas variables que hallan sido modificadas. Las estructuras de datos empleadas son:

1. Una pila en la que se almacena la información de control y las ligas hacia los ambientes de variables. A esta pila se le llamó pila de armazones.

2. Una pila a la que se copia cada uno de los términos cuando alguna variable se acota a ellos,

N	V.NUMERO

Si se trata de un número se almacena directamente su valor.

V	N.VARIABLE

Si se trata de una variable se almacena el número de la variable de acuerdo a la posición como apreció en la cláusula.

F	AP.ESQUELETO

Si se trata de un functor se almacena un apuntador al esqueleto de la llamada a ese functor.

N - número V.NUMERO - valor del número

V - variable N.VARIABLE - número de la variable

F - functor A.P.ESQUELETO - apuntador al esqueleto

Figura 3.6 Descriptores de los parámetros

durante la ejecución. Y en la que también se llevan las variables correspondientes a cada una de las llamadas. A esta pila se le llamó pila de copias y variables.

3. La pila con la que se controlan los puntos de retroceso. A esta pila se le denominó pila de retroceso.

4. La pila con la que se lleva el registro de cada una de las variables que son acotadas durante la ejecución de una llamada, para que en caso de necesitar hacer retroceso se desacoten dichas variables. A esta pila se le llamó pila del rastro de variables.

A continuación se da una explicación de los campos de cada una de las pilas utilizadas en la interpretación.

Pila de armazones

En la pila de armazones se almacenan los armazones que corresponden a los registros de activación usados en las implantaciones de lenguajes de programación con estructura de bloques. Un armazón está compuesto de los siguientes campos (véase figura 3.7):

APSIG - apuntador al siguiente descriptor.

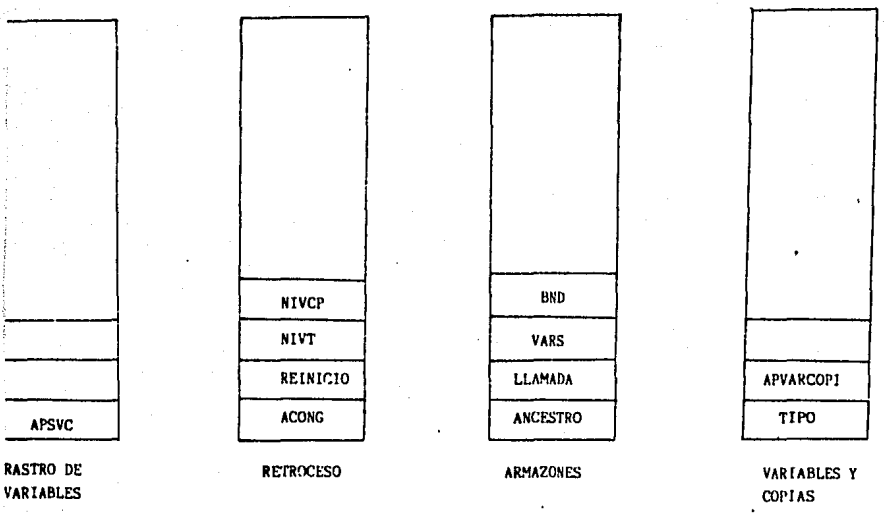

Figura 3.7 Estructuras de datos al tiempo de ejecución

BND - bandera que indica si el término se encuentra en la pila de copias o en el área de esqueletos de las cláusulas.

VARS - un apuntador al ambiente de variables

ANCESTRO - apuntador al armazón padre del registro de activación actual.

LLAMADA - un apuntador al descriptor de la cláusula a ejecutarse

Pila de copias y ambientes de variables

La pila de copias tiene como propósito guardar todos los nuevos términos construidos durante el tiempo de ejecución. La pila esta compuesta de dos campos (véase figura 3.7):

TIPO - bandera que indica si el tipo del término que esta en la pila, esto es, si es: functor, variable o número.

APVARCO - es un apuntador dentro de la misma pila, apunta hacia el área de variables o copias.

¿Cuándo se crea un término en la pila de copias?

Un nuevo objeto se crea, al tiempo de ejecución, bajo las siguiente circunstancia:

Durante la unificación, cuando una variable libre se acota a un objeto estructurado (functor o lista) conteniendo variable. Una nueva copia del objeto estructurado se hecha en la pila de copias y un apuntador es asignado desde la variable libre a la copia.

Los objetos se regresan y se recupera su almacenamiento bajo las siguientes circunstancias:

1. Por el fracaso en la llamada a un predicado. Todo el almacenamiento usado en la pila de copias durante la ejecución es recuperado.

2. Cuando hay sobreflujo ("overflow") en la pila de copias algunos de los objetos se hacen inaccesibles.

En la parte inferior de la pila se manejan los ambientes de variables en los cuales se almacenan los parámetros de las cláusulas y las variables locales[9] como en los procedimientos tipo Algol. El ambiente de variables para una cláusula unificada es igual al número de variables que aparecen en la cláusula. Las variables de la cláusula elegida para ejecutarse están asociadas con el registro de

---

[9] Variable local: son variables que bajo ninguna circunstancia son usadas para formar instanciación de variables fuera de la cláusula en la cual estan definidas. Los parámetros son también llamados variables globales.

activación de la llamada, así que las submetas se pueden referir a un nodo común para consultar los parámetros y variables locales sin tener que hacer una copia.

Pila para el manejo de los puntos de retroceso

La pila de retroceso guarda la información del apuntador a la lista de cláusulas candidatos a unificar con la llamada actual.

Esta pila tiene cuatro campos (véase figura 3.7)

NIVCP - apuntador a la pila de variables y copias.

NIVT - apuntador a la pila de rastro de variables.

REINICIO - apuntador al punto donde se reinicia la ejecución de una meta en caso de que falle.

ACONG - apuntador a los armazones que no se pueden borrar porque van a servir en los siguientes pasos.

El proceso de retroceso [CLOW81] entra en acción cuando una cláusula no puede ser demostrada, y se intenta demostrar usando alguna otra definición de cláusula que tenga el mismo nombre y número de parámetros. De esto se concluye que es necesario mantener agrupadas las cláusulas que tengan el mismo nombre y aridad para facilitar asi el mencionado proceso de retroceso.

Para agrupar un conjunto de cláusulas con el mismo nombre y aridad, desde el diccionario se apunta a una serie de descriptores de cláusula ligados entre sí, y cada una de ellos apunta a la definición de una cláusula. De esta manera el control de retroceso se hace llevando un apuntador, por cada llamada a una cláusula, que permite saber cual descriptor ya fue utilizado y así pasar al siguiente, si lo hay.

Pila del rastro de las variables

Durante la unificación, la dirección de las nuevas variables acotadas se registra en la pila de rastro de variables. Si hay retroceso estas direcciones son usadas para deshacer los acotamientos de las variables.

Esta pila solo tiene un campo (véase figura 3.7)

APSVC - apuntador a las variables que se estan modificando en ese paso de unificación.

3.5 REPRESENTACION INTERNA

Cuando se planteó inicialmente el desarrollo de este intérprete se habia solicitado que el proceso de interpretación se hiciera sobre el programa fuente en forma directa. Sin embargo se tomó la decisión personal de usar una representación interna para los programas en lugar de interpretar directamente usando el código fuente, debido a que resulta más eficiente efectuar la traducción del programa a una representación interna antes de ejecutarlo. El diseño de la representación interna se organizó de tal forma que permita acelerar el proceso de interpretación y retroceso; para este propósito las cláusulas se agruparon por nombre y aridad. La representación interna de los programas se hace utilizando una base de datos para el programa, cuyos principales componentes son un diccionario de datos y una base de datos en la que las cláusulas se almacenan mediante listas ligadas.

3.5.1 Formas de almacenamiento

Listas

Para representar listas se usó la notación de Edinburgo, esto es la lista encerrada entre paréntesis cuadrados y los elementos de la lista separados por comas. Para separar la cabeza de la cola de la lista se usa la

barra. No se usó la notación de Marsella, el punto ('.') tiene dos semánticas diferentes, sirve como separador de los elementos de una lista. Ejemplo: en a.b.c.[] el punto separa los elementos a, b y c de la lista; pero también sirve para separar la cabeza y cola de una lista. Ejemplo: en a.X punto separa "a" como la cabeza de la lista, y como cola la lista que contiene la variable X.

Se almacenarón como funtores de aridad dos en el que el primer paramétro del functor es el primer elemento de la lista y el segundo es el resto de la lista.

Cadenas

Clocksin [CLOW81] señala que las cadenas se representan internamente como listas. Por ejemplo, la cadena maria quedaría de la siguiente forma (véase figura 3.8).

Sin embargo, representar las cadenas de esta manera implica las siguientes desventajas:

Más tiempo de unificación.

Más espacio para la representación.

La ventaja es que se tendrían menos rutinas porque para hacer las operaciones de cadenas se usarían las rutinas que ya se tienen para listas (append, CAR, CDR, etc).

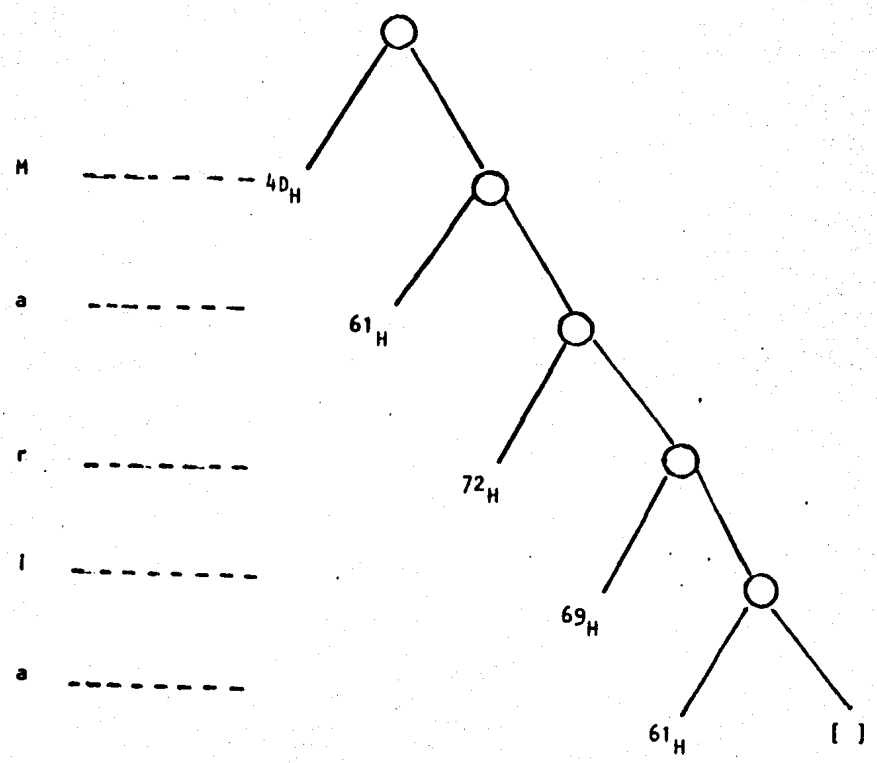

Figura 3.8 Representación interna de las cadenas

En este trabajo se decidió representar las cadenas como nombres de functores con aridad cero, al igual que los átomos constantes debido a que se presentan las siguientes ventajas:

Menos espacio

Menos tiempo de unificación.

La desventaja encontrada es que se requieren más rutinas básicas para el manejo de cadenas.

Análisis de complejidad

Veamos el siguiente análisis de donde se obtiene la cantidad de espacio usado para las dos representaciones.

Si la lista contiene un solo elemento: por ejemplo, [a], con el primer método obtenemos que necesitamos 6 bytes para representarlo (dos bytes para un apuntador al diccionario, dos bytes para el apuntador a la cabeza de la lista y dos bytes más para el apuntador a la cola de la lista). Pero si la lista crece a dos elementos necesitaremos 12 bytes y así sucesivamente. El total de bytes requeridos será la longitud de la cadena multiplicada por 6. Como se ve, esto resulta ineficiente.

Con el segundo método necesitamos 9 bytes, los cuatro campos del diccionario y uno más para el fin de cadena, y

esta es una cantidad constante para todas las cadenas sin importar su longitud. En ambos casos se considera que se necesita un byte para cada carácter de la cadena (no está tomado en cuenta en ambos casos).

Números

Se representarón como estructuras con dos campos: uno para el tipo y otro para el valor numérico.

Functores

Los nombres de éstos se almacenaron como un apuntador al descriptor del functor y los descriptores de sus parámetros los cuales pueden ser:

functores

Números

Variables

En la figura 3.6 hay una descripción gráfica de como se representó lógicamente cada uno de los tipos: functor, número y variable.

Variables

Las variables se almacenaron como estructuras con dos campos: uno para el tipo de variable y el otro para el

número dc la variable. Este número es obtenido del orden de aparición dentro de una cláusula. Las variables anónimas no tienen número, porque estas variables no regresan ningún valor (son lo equivalente a variables void en el lenguaje C). El alcance de las variables es local hasta donde esta el punto de terminación del procedimiento que se esta definiendo.

3.6 BIBLIOTECA DE RUTINAS DEL SISTEMA

Aquí se incluyen rutinas de E/S así como los predicados extralógicos entre los que destacan los operadores aritméticos, is, cut, assert y retract. De ser necesario, la biblioteca puede ser incrementada en un futuro. Unicamente Escribiendo en lenguage "C" la nueva rutina e incluirla en el archivo "rutsis.c" (el archivo rutsis.c contiene el código fuente que se diseñó para cada una de las rutinas del sistema) e insertando la llamada de la rutina en la función rutsis que se encuentra en el archivo rutsis.c.

En la versión actual todos los operadores son manejados en forma funcional, lo cual significa que no existe manejo dinámico de precedencias.

Los predicados interconstruidos, que forman las rutinas de biblioteca que fueron desarrolladas, se pueden clasificar como aritméticos, de comparación, lógicos y de control, para la manipulación de la base de datos y rutinas para depuración de programas.

predicados aritméticos

Suma

Suma dos números enteros y deja el valor de dicha suma en el tercer parámetro. La forma de utilizarlo es:

+(TERMINO, TERMINO, TERMINO)

El predicado suma checa que los dos primeros parámetros sean enteros o variables acotadas a enteros, que el tercero sea una variable libre o no acotada. Si no se cumplen estas condiciones regresa falso y si se cumplen, entoces unifica el parámetro 3 con la suma del parametro 1 más el parámetro 2 regresando verdadero.

Resta

Efectua la resta de dos números enteros y deja el valor de dicha resta en el tercer parámetro. La sintaxis es:

-(TERMINO, TERMINO, TERMINO)

El predicado resta checa que los dos primeros parámetros sean enteros o variables acotadas a enteros, que el tercero sea una variable libre o no acotada. Si no se cumplen estas condiciones regresa falso y si se cumplen, entoces unifica el parámetro 3 con la resta del parametro 1 menos el parámetro 2 regresando verdadero.

Producto

Efectua el producto de dos números enteros y deja el valor de dicho producto en el tercer parámetro. La sintaxis es:

*(TERMINO, TERMINO, TERMINO)

El predicado producto checa que los dos primeros parámetros sean enteros o variables acotadas a enteros, que el tercero sea una variable libre o no acotada. Si no se cumplen estas condiciones regresa falso y si se cumplen, entoces unifica el parámetro 3 con el producto del parametro 1 por el parámetro 2 regresando verdadero.

División

Efectua la división enteroa de dos números y deja el valor de dicha división en el tercer parámetro. La sintaxis es:

div(TERMINO, TERMINO, TERMINO)

El predicado div checa que los dos primeros parámetros sean enteros o variables acotadas a enteros, que el tercero sea una variable libre o no acotada. Si no se cumplen estas condiciones regresa falso y si se cumplen, entoces unifica el parámetro 3 con el resultado de dividir el parametro 1 entre el parámetro 2 regresando verdadero. Esta operación verifica que el parámetro 2 sea diferente de cero.

"is"

El predicado "is" hace que la expresión aritmética del lado izquierdo se evalue y sea asignada a la variable que se encuentra del lado derecho. La sintaxis es:

is(TERMINO,TERMINO)

El predicado "is" checa que el primer parámetro sea variable libre y el segundo sea entero o variables acotadas a entero. Si no se cumplen estas condiciones regresa falso y si se cumplen, entoces unifica el parámetro 1 con parámetro 2 regresando verdadero.

Predicados de comparación

Predicado ‹

Este predicado compara dos términos. La sintaxis es:

‹(TERMINO , TERMINO)

Este predicado checa que los dos parámetros esten acotados a cadenas o enteros, si no se cumple esta condición regresa falso. Si se cumple entonces regresa falso o verdadero dependiendo del resultado de la comparación.

Predicado ›

Este predicado compara dos términos. La sintaxis es:

›(TERMINO , TERMINO)

Este predicado checa que los dos parámetros esten acotados a cadenas o enteros, si no se cumple esta condición regresa falso. Si se cumple entonces regresa falso o verdadero dependiendo del resultado de la comparación.

Predicado ‹=

Este predicado compara dos términos. La sintaxis es:

‹=(TERMINO , TERMINO)

Este predicado checa que los dos parámetros esten acotados a cadenas o enteros, si no se cumple esta condición regresa falso. Si se cumple entonces regresa falso o verdadero dependiendo del resultado de la comparación.

Predicado ›=

Este predicado compara si dos términos. La sintaxis es:

›=(TERMINO, TERMINO)

Este predicado checa que los dos parámetros esten acotados a cadenas o enteros, si no se cumple esta condición regresa falso. Si se cumple entonces regresa falso o verdadero dependiendo del resultado de la comparación.

Predicados lógicos

El predicado lógico not en PROLOG no tiene el mismo significado que el not lógico o de matemáticas, ya que su significado no puede ser completamente entendido sino en términos de la forma en que los programas de PROLOG se ejecutan.

En lógica o matemáticas la negación de una proposición

no significa ejecutar lo que indica la proposición y después simplemente pretender que no se hizo nada de lo que indicaba ésta. Por ejemplo no tirar basura no significa primero tirar basura y luego pretender que eso no se hizo. Lo que realmente significa es nunca hacer esa acción. Esto no es el caso de un lenguaje de programación que está basado en la ejecución de funciones previamente definidas por el usuario en el que para negar una proposición primero se ejecutan las acciones que la forman y después se niega el resultado de verdad o falsedad que resulte de dicha ejecución.. Por esta razón aún en los lenguajes de programación lógica de más alto nivel, se prohibe o restringe severamente el uso de la negación. Se requiere que el programador la desarrolle cuando la necesite o que si existe este conciente de la diferencia que existe entre la negación de la lógica y la del lenguaje.

Puede ser definido de la siguiente manera:

```
not(X) :- X & ! & fail.
not(X).
```

La negación en PROLOG es verdadera si la ejecución de X regresa falso y viceversa.

Por otro lado los conectivos and y or también tienen un significado peculiar, como resultado del cual no son simétricos. Hay tres razones importantes por las que el orden de las cláusulas es importante:

1. Eficiencia en la ejecución

Para entender la eficiencia en la programación se utiliza el siguiente ejemplo tomado de [KOWR74]. Se trata de un ordenamiento extremadamente simple y para definirlo se tienen dos procedimientos permuta y ordenada.

Permuta puede ser utilizado de dos formas dependiendo de los valores de sus parámetros.

a. permuta(L1,L2) donde L1 es una lista no vacia y L2 es una variable libre, genera una permutación de L1 y la regresa en L2.

b. permuta(L1,L2) donde L1 y L2 son listas no vacias, verifica que L2 sea una permutación válida de L1.

Ordenada puede ser utilizado también de dos formas dependiendo de los valores de sus parámetros.

a. ordenada(L1) donde L1 es una lista no vacia, verifica que esten ordenados los elementos de la lista.

b. ordenada(L1) donde L1 es una variable libre, genera una lista ordenada de cualquier número de elementos.

La llamada se hace de la siguiente forma:

ordena([1,2,3], X) donde X es una variable no instanciada o libre

Utilizando los dos procedimientos anteriores hay dos formas de definir el procedimiento ordena como se muestra a continuación.

```
ordena(Lista,Lista_ordenada) :-
                    permuta(Lista,Lista_ordenada) &
                    ordenada(Lista_ordenada).
```

Con esta definición el procedimiento genera permutaciones sucesivas de una lista hasta encontrar que una permutación está ordenada.

Si invertimos el orden de permuta y ordenada se obtiene:

```
ordena(Lista,Lista_ordenada) :-
                    ordenada(Lista_ordenada) &
                    permuta(Lista,Lista_ordenada).
```

En esta última definición se van generando listas ordenadas hasta que permuta verifica que una Lista_ordenada sea una permutación válida de Lista (lista que se desea ordenar). A final de cuentas ambos procedimientos expresan la misma definición de lista

ordenada. La diferencia en términos de PROLOG es que la primera (definición de ordena) es costosa pero la segunda es totalmente impráctica con lo que se demuestra que el orden de las cláusulas es importante.

2. Efectos colaterales

Algunos pocedimientos tienen efectos colaterales. Ejemplo write y nl. El orden en el cual las cosas son escritas tiene un efecto en la forma de impresión. Otro ejemlo es assert y retract. Considerese que se tiene una base de datos con los siguientes axiomas:

```
a(b).
a(c).
a(d).
a(e).
```

Y se quiere borrar a(d) que esta en la tercera posición pero se hace assert de a(a) en la primera posición lo cual obligará que las demás cláusulas se recorran una posición. Si se hace retract(a,1,3) & assert(a(a),[],1) se logra hacer lo que se desea pero si se invierte el orden de las cláusulas assert y retract, entonces se borra a a(c).

3. Los cálculos deben ser finitos

Considérese este programa para concatenar dos listas:

```
concatena([],L,L).
concatena([X|Y],L,[X|Z]) :- concatena(Y,L,Z).
```

El cual es aparentemente equivalente a

```
concatena([X|Y],L,[X|Z]) :- concatena(Y,L,Z).
concatena([],L,L).
```

pero el primer procedimiento encuentra la solución en un tiempo finito y el segundo entra en un lazo infinito debido a que la condición de salida está al final de la recursión y nunca la tomará en cuenta en la ejecución, esto es porque la ejecución de los programas en PROLOG es secuencial.

Predicados de entrada/salida

Escribe

El predicado escribe se encarga de escribir un termino. La sintaxis es:

```
escribe(TERMINO)
```

El predicado escribe, manda escribir un término en la salida. Este predicado esta diseñado para poder escribir variables, cadenas, números y functores. Las variables no acotadas son escritas como X1, X2, etc; las acotadas se escriben con el valor del término al que están acotadas. Las cadenas se escriben como cuerdas de caracteres. Los

números como cuerda de enteros. Los functores se escriben en la forma estándar que es el nombre del functor seguido por sus parámetros.

nl

Escribe un retorno de carro a la salida.

Predicados para la manipulación de la base de datos

Assert

El predicado assert permite insertar nuevas cláusulas a la base de datos al tiempo de ejecución. La sintaxis es:

assert(TERMINO, TERMINO, TERMINO)

Ejemplo:

assert(a(X,Y), [b(X),c(Y)], 3).

El parametro 1 debe ser un nombre de functor o una variable acotada a un nombre de functor, el parámetro 2 debe ser una lista de llamadas (escritas en la notación de listas de PROLOG) y el parametro 3 debe ser un entero o variable acotada a entero. En esta rutina el parámetro 1 es la cabeza de la cláusula y el parámetro 2 el cuerpo. La cláusula se inserta despúes de la cláusula cuyo número es el parámetro 3 si existe una cláusula con este número. Si la definición

del procedimiento es vacía o el parámetro es menor que uno, la cláusula se inserta como la primera con ese nombre.

Retract

El predicado retract permite borrar cláusulas de la base de datos al tiempo de ejecución. La sintaxis es:

retract(TERMINO, TERMINO, TERMINO)

El parametro 1 debe ser un nombre de functor o una variable acotada a un nombre de functor y los parametro 2 y parametro 3 deben ser enteros o variables acotadas a enteros. El parámetro 1 y el parámetro 2 definen el nombre y aridad del predicado respectivamente. El parámetro 2 no debe ser un número negativo. Si no se tiene dada de alta una cláusula número el parámetro 3, o si la cláusula no existe, la operación regresa falso. Esta operación tiene efectos colaterales ya que elimina la cláusula de la base de datos si la operación tiene éxito regresando en este caso el valor verdadero.

Predicados de control

stop

Detiene la operación del intérprete.

fail

Fuerza el fracaso de un predicado y por lo tanto obliga al intérprete a hacer retroceso.

true

Siempre tiene éxito.

! (CUT)

El procedimiento CUT encuentra el ancestro más cercano que no sea un call y remueve todos los puntos de falla existentes que esten asociados con este ancestro y todos sus descendientes.

Call

El predicado call efectua una llamada a un procedimiento. la sintaxis es:

call(CALL)

Esta operación hace una llamada al procedimiento cuyo nombre es pasado como parámetro. La restricción para esta operación es que el parámetro no sea entero ni una variable no instanciada ya que esto lleva a un error. Este error es detectado al tiempo de ejecución y no al tiempo de análisis sintáctico.

Predicados de depuración

trace

Este predicado interconstruido permite rastrear la ejecución de un programa. Despliega las llamadas con los valores de los parámetros que se tienen en ese momento y va indicando éxito o fracaso en cada llamada.

notrace

Apaga la bandera de rastreo y regresa el valor verdadero.

3.7 El INTERPRETE

Ejecuta acciones a partir de la representación interna que se genera durante el proceso de análisis sintáctico. Este módulo es un probador de teoremas basado en el principio de resolución de Robinson. En el artículo [EMDM82], Van Emden propuso un algoritmo para interpretar programas en PROLOG; en general dicho algoritmo puede resultar un magnífico punto de inicio para el desarrollo de cualquier intérprete. En este caso en particular sirvió como referencia para desarrollar este trabajo.

Como métodos para la representación o construcción de términos estructurados durante el proceso de unificación existen los llamados estructuras compartidas y copia de estructuras que han sido discutidos por Bruynooghe [BRUM82] y Mellish [MELC82], encontrando ejemplos "patológicos" que hacen ambos métodos altamente ineficientes. El intérprete se hizo utilizando la técnica de copia de estructuras[43] debido a que se deseaba emplear uno de estos dos esquemas de representación.

A continuación se hacen las comparaciones en cuanto a tiempo y espacio para ambos métodos.

---

[43] Es uno de los métodos que se usan para realizar la representación de términos durante la ejecución de los programas, en el que se crea una copia cuando una variable se acota a un término del tipo functor o lista

Estructuras compartidas

1. En este esquema, todo término construido consta de dos apuntadores: un apuntador a la representación del esqueleto de la cláusula donde está ese término, y un apuntador a su área de control. Resulta así rápido construir un nuevo término; lo único que hay que hacer es instanciar dos apuntadores.

2. En los casos en que el término construido se unifique con una constante, uno de los apuntadores se desperdicia.

3. El acceso a los argumentos de un término estructurado es lento debido a que se tienen que recorrer las referencias hacia los argumentos.

Copia de estructuras

Con este esquema pasa lo siguiente:

1. Para crear un término estructurado nuevo se hace una copia del término desde la representación del esqueleto de la cláusula.

2. El acceso de los argumentos de un nuevo término estructurado es muy rápido, ya que puede leerse directamente de la pila de copias.

3.7.1 Unificación

El algoritmo de unificación[5] utilizado esta basado en el algoritmo de Robinson [ROBJ65]. No tiene verificación de ocurrencias[6] ya que es caro e ineficiente. Puesto que no se verifican ocurrencias cuando se trata de unficar una estructura infinita, el algoritmo estará en un lazo infinito hasta que se termine el espacio disponible en memoria y entonces el intérprete terminará la ejecución. Ejemplo:

```
a(X,X).
?- a(f(Y),Y) & escribe(Y).
```

En este ejemplo Y es igualada con f(Y), y al tratar de seguir la instanciación de la variable para tratar de imprimirla sucede lo que recién se mencionó.

El algoritmo de unificación fue diseñado con base en la tabla mostrada en la figura 3.9, en donde se consideran los diferentes casos de unificación. Los átomos constantes formados por una secuencia de una o más letras minúsculas se trataron como nombres de functores con aridad cero, debido a que al hacer un estudio no se encontró una forma de diferenciar cuándo se trata de un átomo constante o cuándo

---

[5] Unificación es un proceso de igualación en el que dos fórmulas son unificables si pueden ser igualadas.

[6] Verificación de ocurrencias que consiste en evitar que una variable sea unificable con un término en el cual ocurra.

se trata de un functor con aridad cero. Para eliminar este problema de tener dos semánticas para los nombres que empiezan con minúsculas se decidió usar la misma representación interna que se utiliza para functores, Las ventajas obtenidas al usar esta representación se analizan en la sección de representación interna.

		ATOMO CTE a2	VNA LIBRE X2	VAR ANON. _	NUMERO n2	NOFUNCTOR f2	LISTA L2
VNA LIBRE	X1	V con X1 <-a2	V con X1 <-X2	V	V con X1 <-n2	V con X1 <-f2	V con X1 <-L2
VAR ANONIMA	_	V	V	V	V	V	V
NUMERO	n1	F	V con X2 <-n2	V	C	F	F
NOMFUNCTOR	f1	C	V con X2 <-f1	V	F	UF	F
LISTA	L1	F	V con X2 <-L1	V	F	F	UL

Figura 3.9 Casos de unificación

Para facilitar el manejo de la tabla se utilizó la siguiente notación para las entradas a la tabla y para las acciones a realizar:

ENTRADAS:

- ATOMO CTE - átomo constante
- VNA LIBRE - Variable no anónima libre
- VAR ANONIMA - Variable anónima
- NUMERO - números enteros
- NOMFUNCTOR - Nombre de functor
- LISTA - lista

ACCIONES:

- C - El resultado de comparación es verdadero si al comparar los términos[c73] ambos son iguales

[c73] Los términos en PROLOG son: Variable libre, variable anónima, número, lista nomfunctor, nomfunctor(término).

V - Siempre es Verdadero
F - Siempre es Falso
UF - Unificar functores:
verificar si los nombres de ambos son iguales, en cuyo caso se prueba que cada uno de los parámetros se unifique, usando para ello la tabla anterior.
UL - Unificar listas:
unificar uno por uno los elementos de la lista, usando la tabla anterior.
Xi <- ter - indica que la variable Xi se acota al término ter

Explicación:

Las casillas de la tabla que son de la forma "V con Xi <- término " significa que la rutina de unificación regresará verdadero y la variable Xi se quedará acotada al término.

El valor V significa que en esos casos la rutina de unificación regresará verdadero.

El valor F significa que en esos casos no es posible efectuar unificación y la rutina regresará falso.

La acción C significa que se efectua una comparación de términos.

La acción UL significa que se debe de realizar una unificación de dos listas.

La acción UF significa que se efectuará una unificación

de dos términos

3.7.1.1 Algoritmo de unificación

A continuación se muestra el algoritmo de unificación que fue implementado.

Algoritmo de Unificación

```
unifica(Term1, Term2)
tiene como entrada dos términos a unificar
regresa un valor booleano
{

/* Term1 y Term2 son unificados en este algoritmo no
   se hace chequeo de ocurrencias */


éxito = VERDADERO
Notación utilizada en este algoritmo :  "<- " significa que
el término del lado derecho se acota al termino del lado
izquierdo.

Si functor(Term1) y functor(Term2)
              /* term1 y term2 son functores */


   Si diferente(Term1, Term2) exito = FALSO
              /* term1 y term2 son functores diferentes */

   sino
       {
       obtener(aridad)
       n = 0


       mientras diferente(n,aridad)
           {
         éxito = unifica(argumento(Term1,Term2))
           n = n + 1
           }
       }
```

```
sino
   {
   si anónima(term1) or anónima(term2) ;
                      /* si alguno de los terminos es variable
                         anónima no se hace nada */
   sino
       {


       si varlibre(term1)
           {
           si varlibre(term2) ligavars()
             /* si los dos terminos resultaron variables entonces
                ligalos la más joven se liga a la más vieja */



           sino term2 no está libre term1 <- term2
            /* si sólo el termino1 es variable libre esta se acota
               a lo que está acotado el termino2 */
           }


       sino
           {
           si varlibre(term2)   term2 <- term1
            /* si sólo el termino2 es variable libre esta se acota
               a lo que está acotado el termino1 */



           sino
               {
               si número(term2)
                    {
                    si número(term1)

                      éxito = compara(term1,term2)
                       /* si los dos terminos son números se
                          comparan sus valores numéricos */
```

```
                    sino éxito = FALSO
                    }
                sino
                    {
                    si número(term1) éxito = FALSO

                    sino ERROR
                    }
                }
            }
        }
    }
}
```

3.7.2 Algoritmo ABC para interpretar programas

El algoritmo para interpretar programas hace búsquedas en un árbol en el que almacena una secuencia de nodos entre la raiz y el nodo actual.

A continuación se presenta el algoritmo de interpretación que fue utilizado en este trabajo.

Algoritmo de interpretación

1. Suponer la existencia de una cláusula en la que el cuerpo de la pregunta funge como su cuerpo.

2. Crear las entradas necesarias en las pilas e iniciar las variables de control para la pregunta.

   PADRE es NULO.

   PROC no esta definido.

   LLAMADA apunta a la primera meta de la pregunta.

   AMB apunta al armazón de variables de la pregunta

   COPIAS es un apuntador a la pila donde se manejan las copias de los terminos estructurados como functores o listas. En

este paso tiene el valor del límite superior

RASTRO es una pila que apunta a una lista de variables, en el árbol de prueba[*], que fueron modificadas como resultado de la unificación. Su estado inicial es NULO.

RETROCESO el control de retroceso.

CLAUSULA_SIGUIENTE apunta a la siguiente cláusula del mismo nombre que pueda ser unificada con LLAMADA.

3. Unificar LLAMADA con CLAUSULA_SIGUIENTE para lo cual se tiene que:

   a. verificar que CLAUSULA_SIGUIENTE sea diferente de NULO

   b. Crear el armazón de variables para el número de variables que tenga CLAUSULA_SIGUIENTE.

   c. Igualar LLAMADA y CLAUSULA_SIGUIENTE agregando a la pila de rastreo cada variable que sea modificada durante la igualación.

---

[*] Es una estructura de datos que almacena en forma no redundante la trayectoria, en un árbol de búsqueda, de la raiz a un nodo actual [EMDM82].

d. Obtener otra CLAUSULA_SIGUIENTE.

e. Si igualó ir a paso 4, y si no, ir a paso 3

4. Poner nuevos valores a las variables PADRE, PROC, LLAMADA, AMB, RASTREO, COPIAS guardando los anteriores en la pila de armazones

Si hay exito en la LLAMADA ir al paso 3

3.7.3 Optimización en el caso de la recursión

En PROLOG hay un solo mecanismo para efectuar cálculos repetitivos: recursión. La recursión es más cara que la iteración, ya que no solamente se requiere tiempo sino también más almacenamiento, que crece linealmente con el número de llamadas. Debido a que PROLOG usa recursión en vez de iteración el problema es serio.

Existe una forma de reemplazar algunas formas de recursión con iteración. La idea general es suponer que lo que está a la izquierda del registro de activación previo no es relevante al nuevo y que entonces se puede sobre-escribir.

Las condiciones para aplicar este método son las siguientes:

1. La llamada a ser invocada es la última llamada en la cláusula a la cual pertenece.

2. En la misma cláusula, entre esta llamada y la primera llamada, inclusive, no deberá haber puntos de bactracking.

Al intérprete se le incluyó este método para lograr un manejo más eficiente de memoria.

3.7.4 Diagramas de Ferguson

Los diagramas de Ferguson son de gran ayuda porque muestran explicitamente la unificación efectuada en cada paso del programa.

Explicación de las componentes de estos diagramas:

Los semicirculos superiores representan las llamadas.

Los semicirculos inferiores representan la cabeza de la cláusula con la que se unificacó en ese paso.

Las casillas dentro del semicirculo inferior representan al armaón de las variables que serán utilizadas en la cabeza y cuerpo de esa cláusula.

Se denomina ambiente al alcance de un armazón de variables.

El alcance de cada ambiente (que se muestra como la

linea punteada en la figura 3.10) tiene validez a partir de la cabeza de una cláusula y hasta el momento de la unificación de cada una de las cláusulas que forman su cuerpo. Iniciándose a partir de ese momento otro ambiente para la cláusula que se está unificando. El nuevo ambiente y el anterior podrán estar ligados entre si debido a la unificación de los paramétros que aparecen tanto en la llamada como en la cabeza de la definición de la cláusula. Este hecho se muestra en el siguiente ejemplo durante la explicación de la forma en que son acotadas las variables en cada paso de unificación.

3.7.5 El análisis semántico

El análisis semántico fue realizado una parte durante el análisis sintáctico y la otra al tiempo de ejecución.

Durante el análisis sintáctico

Se verificarón:

a. Que la cabeza de una cáusula fuese un término que no sea número ni variable.

b. Que el conjunto de metas que forman la pregunta no fuese un término de tipo número o variable.

Durante el momento de ejecución

Se verifico:

a. En las operaciones aritméticas y de comparación que son de la forma operador(TERMINO,TERMINO,TERMINO) los parámetros 1 y 2 esten acotados a números o sean variables acotadas a números.

b. En la operacón call(TERMINO) se verificó que TERMINO se acote a un nombre de functor. si se acota a cualquier otro término es error.

c. En las operaciónes que manipulan la base de datos. En la operación assert cuyo sintaxis es assert(TERMINO,TERMINO,TERMINO) se verificó que el parámetro 1 y parámetro 2 se acoten a un término que no sea ni variable ni número y que el parámetro 3 sea un número o una variable acotada a número. En la operación retract(TERMINO,TERMINO,TERMINO) se verifica que el parámetro 1 sea un nombre de functor o una variable acotada a un nombre de functor y que el parámetro 2 y parámetro 3 sean números o variables acotadas a números.

3.7.6 Ejemplo de un programa ejecutado por el intérprete

Este programa hace una concatenación de dos listas en una tercera lista. Para esto se define el predicado "con" en forma recursiva. La cláusula 1 es la condición de salida de la recursión y dice que la lista vacia concatenada con una

lista L da como resultado la misma lista L. La cláusula 2 define el proceso de tomar el primer elemento de la primera lista y ponerlo como primer elemento de la tercera, y así sucesivamente tomar el primer elemento del resto de la primera lista hasta reducirla a la lista vacía.

```
1.      con([],L,L).
2.      con([U|X],Y,[U|Z]) :- con(X,Y,Z).
        ?- con([1,2],[3,4],L1) & escribe(L1).
```

En la figura 3.10 se muestra el diagrama de Ferguson para el ejemplo anterior.

El proceso de acotamiento de las variables de este ejemplo es como se muestra a continuación.

Primero se trata de igualar con la cláusula 1 pero esto falla, y entonces se iguala con la cláusula 2 exitosamente. Los valores obtenidos en este primer paso bajo el ambiente A1 son:

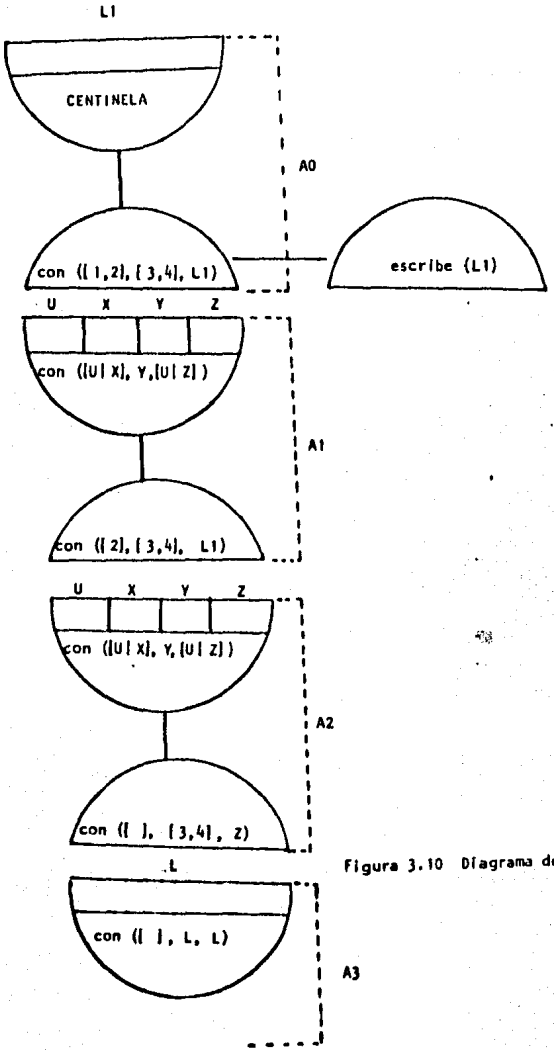

Figura 3.10 Diagrama de Ferguson

```
U/A1  se acota con  1/A0
X/A1  se acota con  [2]/A0
Y/A1  se acota con  [3,4]/A0
L1/A0 se acota con  [U/A1,Z/A1]
Z/A1  no se acota durante este paso
```

No unifica con la cláusula 1 pero sí con la cláusula 2.

Los valores obtenidos bajo el ambiente A2 son:

```
U/A   se acota con   2/A1
X/A2  se acota con   []/A1
Y/A2  se acota con   Y/A1 = [3,4]/A0
Z/A2  no se acota en este paso
Z/A1  se acota con [U/A2,Z/A2]
```

Ahora ya unifica la meta a resolver con la cláusula 1 y los valores bajo el ambiente A3 son:

```
L/A3  se acota Y/A2 = Y/A1 = [3,4]/A0
Z/A2  se acota con L/A3 = Y/A1 = [3,4]/A0
```

Como ya se terminó esta trayectoria de solución, las variables quedaron como:

```
Z/A2 = [3,4]
Z/A1 = [U/A2,Z/A2]
```

Sustituyendo los valores que ya se tienen

```
Z/A1 es [2,[3,4]]
L1/A0 = [U/A1,Z/A1]
```

sustituyendo por los valores que se tienen

```
L1/A0 = [1,[2,[3,4]]] = [1,2,3,4]
```

CAPITULO 4

Discusión

En la bibliografía que se obtuvo de trabajos previos realizados en Marsella [COLM79], se menciona que se desarrolló un intérprete para un lenguaje intermedio de más bajo nivel que PROLOG, el cual era usado para escribir en él un programa que finalmente interpretaba PROLOG. Esta solución implicaba un doble proceso de interpretación, lo que hace más lento al intérprete. En este trabajo no se siguió esta filosofía sino que se hizo reconociendo directamente PROLOG, con lo cual se mejora la velocidad de ejecución, esto es importante para el tipo de aplicación en la que se va a utilizar el intérprete. El criterio para seleccionar la técnica con la cual se desarrollaría el intérprete dependió del campo de aplicación. Por ejemplo si el campo de aplicación requiere tiempos de respuesta interactiva o de tiempo real la filosofía con la que se desarrolló este trabajo será la más adecuada ya que al eliminar un paso intermedio se adquiere rapidez en el tiempo de respuesta. Si la aplicación es de otro tipo cualquiera de las dos filosofías podría utilizarse en el desarrollo del intérprete.

A continuación se presentan las decisiones tomadas durante el desarrollo del intérprete.

1. Definición de la gramática

Lo primero que se intentó fue tener una gramática de PROLOG completa pero los primeros problemas que se encontraron fueron que la gramática de PROLOG tenía ambigüedades, y este tipo de gramáticas no se pueden manejar con Yacc. Se decidió entonces seleccionar la gramática que puede ser reconocida directemente. Dicha gramática es la de PROLOG sin el manejo dinámico las precedencias de operadores.

2. Representación interna

Por razones de eficiencia se decidió usar una representación interna en vez de interpretar directamente usando el código fuente como fue planeado el desarrollo del intérprete. El diseño de la representación interna se organizó de tal forma que permitiera acelerar el proceso de interpretación y de retroceso.

3. Interpretación

Para manejar la ejecución de un programa en PROLOG se definierón estructuras de datos tipo pila y se hicieron las rutinas para llevar el control de ambientes de variables, puntos de retroceso, variables que se modifican en la unificación y copias en el caso de que una variable se acotaba a una estructura de tipo

lista o functor. El motivo de llevar una pila para hacer copias fue que se utilizó la técnica de copia de estructuras que es uno de los dos métodos que se tienen para la representación de términos durante la ejecución.

Finalmente se obtuvo un interprete con las siguientes características:

1. Un intérprete que puede ser usado en un ambiente Unix.

2. El manejo los operadores en forma funcional.

3. Facilidades para la depuración de programas ("trace").

4. Facilmente extendible debido a la modularidad con que fue diseñado, ya que sólo hay que modificar dos módulos para agregar nuevas rutinas de bajo nivel, escritas en lenguaje "C", a la biblioteca en forma permanente.

5. Permite que programas en escritos en el lenguaje "C" usen programas escritos en PROLOG.

6. Reusabilidad de programas ("software"). Permite que los nuevos programas de PROLOG usen programas previamente desarrollados.

CAPITULO 5

Conclusiones

Se trabajó ampliamente con estructuras de datos dinámicas y secuenciales viéndose que la mejor opción para el el almacenamiento interno de las cláusulas y de las pilas para el control de la interpretación y de las copias que se crean durante la ejecución es la secuencial porque permite que el módulo de interpretación sea más compacto y además permite llevar un mejor control de lugares de memoria están congelados en algún caso en que se efectúe retroceso.

Los objetivos del trabajo se cumplieron y tuvo características adicionales a las que inicialmente fueron planteadas. Porque inicialmente no se pretendiia utilizar una representación interna para las cláusulas sino que quería hacer la interpretación directamente sobre el código fuente, pero esto poco eficiente y poco elegante.

CAPITULO 6

Etapas futuras

Se puede plantear una serie de extensiones al trabajo realizado y al lenguaje, con el propósito de que resulte más práctico.

Las extensiones posibles al intérprete se pueden clasificar como extensiones a corto, mediano y largo plazo. Esta clasificación está hecha de acuerdo al grado de dificultad que implica realizar cada una de las extensiones.

Extensiones a corto plazo

1. Aumentar la biblioteca de rutinas del intérprete

Esto resultaría muy sencillo de realizar debido a que el código fuente está diseñado modularmente para que resulte fácilmente extendible. Los pasos a seguir para insertar una rutina son:

a. Incluir el nombre y aridad de la rutina básica en un archivo de carga inicial llamado "núcleo". El archivo núcleo contiene los nombres de las rutinas del sistema en el formato: nombre de la rutina aridad. Se utiliza para hacer una carga inicial de rutinas del sistema.

b. Escribir en lenguage "C" la nueva rutina e incluirla en el archivo "rutsis.c". El archivo

rutsis.c contiene el código fuente que se diseñó para cada una de las rutinas del sistema.

c. Insertar la llamada de la rutina en la función rutsis que se encuentra en el archivo rutsis.c.

La necesidad de incluir una nueva rutina puede darse si se desea definir una nueva operación que requiera de alguna rutina básica no disponible actualmente. Para esto se tendría que incluir dicha rutina en el conjunto de la manera antes indicada y finalmente se podrían definir las operaciones que la van a usar.

2. Optimizar la representación interna de los programas

Para hacer esto se podría almacenar en un solo arreglo el área de memoria donde se guardan los esqueletos de las cláusulas, la pila de copias y la pila de variables. Con esto se podría hacer más sencillo y pequeño el módulo del intérprete. Además, si a cualquiera de estas áreas se le acabara el espacio se podrían aplicar técnicas como la de Garwick [STAT80] para la reubicación de pilas, agregando la posibilidad de que el arreglo crezca si la reubicación no es posible por falta de espacio libre entre las tres áreas.

Extensiones a mediano plazo

1. Incluir manejo dinámico de operadores

Otra dirección en la que se puede trabajar es en

incluir el manejo dinámico de operadores y sus precedencias, realizando una etapa de preprocesamiento antes del análisis sintáctico, durante la cual se convertirán las expresiones infijas a cláusulas de PROLOG y el programa resultante se pasará como entrada al intérprete de PROLOG. Este nuevo módulo será un preprocesador para las expresiones de tipo aritméticas, lógicas y de comparación.

Extensiones a largo plazo

Se pueden hacer extensiones al lenguaje para que PROLOG resulte un lenguaje de programación más apropiado para ser utilizado en el manejo de grandes bases de datos. El propósito de estas extensiones sería facilitar el desarrollo de aplicaciones en las que es necesario el acceso y modificación concurrente de los datos por múltiples usuarios. En el estado actual del lenguaje no es posible desarrollar aplicaciones de este tipo lo que limita su campo de aplicación. Las nuevas áreas de aplicación pueden ir desde bases de datos para CAD hasta bases de conocimiento.

1. Usar PROLOG en un contexto de bases de datos multiusuario.

2. Incluir las operaciones necesarias para que se facilite la definición de las transacciones dentro de PROLOG.

3. Extender el intérprete para que maneje bases de datos en disco.

4. Tratar de optimizar el tiempo de acceso y actualización a los datos en disco.

Aplicaciones futuras

Como se mencionó en la introducción, se tratará de ver la factibilidad de usar PROLOG en la parte de manejo de restricciones para el "Sistema Manejador de Bases de Datos para aplicaciones de la computación aplicada al diseño (CAD)". En este momento se está en la etapa de identificación y análisis de las clases de restricciones y excepciones que podrían ser manejadas desde PROLOG. Hasta ahora se ha llegado a la conclusión de que PROLOG podría manejar perfectamente esta aplicación y que si se hacen las extensiones necesarias para el manejo de bases de datos muy bien podría ser usado no sólo para las restricciones sino para desarrollar en él las aplicaciones de CAD.

APENDICE 1

PRUEBAS DEL INTERPRETE

Ejemplo 1

MIEMBRO DE UNA LISTA

```
m(X,[X|_]).
m(X,[_|Y]):-m(X,Y).
?- m(2,[1,3,7,2,5]) & escribe("si lo encontre").
PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO

 si lo encontre
--- TRUE---

F I N    D E    S E S I O N
```

El programa verifica que un elemento "X" esté en una lista. Para esto se define el predicado m que tiene dos argumentos: el elemento a buscar ("X"), y la lista en la cual se realiza la búsqueda.

Primero se determina si "X" es la cabeza de la lista, mediante la cláusula 1. En este caso la cola de la lista se puso como variable anónima porque no necesitamos conocer la cola de la lista. Luego se busca si "X" es elemento de la cola de la lista, con la cáusula 2. En este caso no interesa el valor de la cabeza de la lista razón por la cual se usó una variable anónima.

Este ejemplo fue seleccionado para mostrar la recursión y el uso de listas en PROLOG.

APENDICE 1

PRUEBAS DEL INTERPRETE

Ejemplo 1

MIEMBRO DE UNA LISTA

```
m(X,[X|_]).
m(X,[_|Y]):-m(X,Y).
?- m(2,[1,3,7,2,5]) & escribe("si lo encontre").
PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO

 si lo encontre
--- TRUE---

F I N    D E    S E S I O N
```

El programa verifica que un elemento "X" esté en una lista. Para esto se define el predicado m que tiene dos argumentos: el elemento a buscar ("X"), y la lista en la cual se realiza la búsqueda.

Primero se determina si "X" es la cabeza de la lista, mediante la cláusula 1. En este caso la cola de la lista se puso como variable anónima porque no necesitamos conocer la cola de la lista. Luego se busca si "X" es elemento de la cola de la lista, con la cáusula 2. En este caso no interesa el valor de la cabeza de la lista razón por la cual se usó una variable anónima.

Este ejemplo fue seleccionado para mostrar la recursión y el uso de listas en PROLOG.

Ejemplo 2

MIEMBRO DE UNA LISTA, CON RASTREO

```
m(X,[X|_]).
m(X,[_|Y]):-m(X,Y).
?-trace & m(2,[1,3,7,2,5]) & escribe("si lo encontre").


PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO

  t r u e
  m(2, .(1, .(3, .(7, .(2, .(5,[])))))) f a l s e
  m(2, .(1, .(3, .(7, .(2, .(5, [])))))) t r u e
  m(2, .(3, .(7, .(2, .(5, [])))))) f a l s e
  m(2, .(3, .(7, .(2, .(5, [])))))) t r u e
  m(2, .(7, .(2, .(5, [])))) f a l s e
  m(2, .(7, .(2, .(5, [])))) t r u e
  m(2, .(2, .(5, []))) t r u e
  escribe(si lo encontre) si lo encontre t r u e
--- TRUE---

F I N   D E   S E S I O N
```

Este es el ejemplo 1 con la bandera de rastreo prendida para que se imprima la llamada y el valor booleano que se tiene en cada paso. Como puede observarse, hay dos llamadas con la misma lista: una con valor falso y la otra verdadero. Esto es porque al llamar recursivamente a m se verifica la primera regla de m (y para esos valores es falso), y porque al fracasar la regla 1 se pasa a la regla 2, con la cual se tiene éxito.

Ejemplo 3

```
DENSIDAD DE POBLACION
pob (eu,203).
pob(india,548).
area(eu,3).
area(india,1).
densidad(X,Y) :-  pob(X,P)
                & area(X,A)
                & div(P,A,Y).
?- densidad(Z,X) & escribe(X)  & escribe(" es la densidad de ") &
escribe(Z) & nl & fail.

PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO

 67  es la densidad de  eu
 548  es la densidad de  india
--- FALSE---
```

El programa encuentra la densidad de población a partir de la población y el área de una ciudad.

El predicado pob relaciona una ciudad X con Y millones de personas, y el predicado área denota relaciones entre la ciudad Y y su área en millones de millas cuadradas.

Este ejemplo fue seleccionado para probar rutinas del sistema y ver cómo las variables se instancian a constantes.

Como puede notarse, al final el intérprete responde **FALSE** y esto es debido a que se utilizó el predicado interconstruido fail cuyo efecto es hacer que la cláusula fracase y efectuar un retroceso para tratar de encontrar otra solución, en este problema ya no puede encontrar otra y por eso responde **FALSE**.

Ejemplo 4

DENSIDAD DE POBLACION, CON RASTREO

```
pob (eu,203).
pob(india,548).
area(eu,3).
area(india,1).
densidad(X,Y) :-  pob(X,P)

                & area(X,A)

                & div(P,A,Y).
```

```
?-trace & densidad(Z,X) & escribe("X=") & escribe(X) &nl & fail.

PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO

t r u e
 densidad( V1, V2) t r u e
 pob(V1, V3) t r u e
 area(eu, V4) t r u e
 div( 203, 3, V2) t r u e
 escribe(X=) X= t r u e
 escribe(67) 67 t r u e
 nl
t r u e
 fail f a l s e
 area(eu, V4) f a l s e
 pob(V1, V3) t r u e
 area(india, V4) f a l s e
 area(india, V4) t r u e
 div(548, 1, V2) t r u e
 escribe(X=) X= t r u e
 escribe(548) 548 t r u e
 nl
t r u e
 fail f a l s e
--- FALSE---

F I N   D E   S E S I O N
```

Ejemplo 5

CONCATENAR DOS LISTAS

```
 concatena([],L1,L1).
 concatena([X|L1],L2,[X|L3]) :- conactena(L1,L2,L3).

 ?-concatena([1,2,3],[4,5,6],X) & escribe("X=") & escribe(X)
& nl.

 PROLOG 1.1  D.R. 5/17/87  por:
 Ma. S. Vargas
 iimas-unam, MEXICO

 X=  .(1, .(2, .(3, .(4, .(5, .(6, [])))))
 --- TRUE---

F I N   D E   S E S I O N
```

Este programa une dos listas, L1 y L2, en una nueva lista L3. El primer elemento de la lista L1 siempre será el primer elemento de la tercera lista (L3). El proceso consiste en ir tomando el primer elemento del resto de la lista L1 hasta reducirla a la lista vacia.

Ejemplo 6

```
CONCATENAR DOS LISTAS, CON RASTREO
?-trace.
concatena([],L1,L1).
concatena([X|L1],L2,[X|L3]) :- concatena(L1,L2,L3).
?- concatena([1,2,3],[4,5,6],X) & escribe("X=") & escribe(X) & nl.

PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO

 t r u e
  concatena(.(1, .(2, .(3, []))), .(4, .(5, .(6,[]))), V1) f a l s e
  concatena(.(1, .(2, .(3, []))), .(4, .(5, .(6, []))), V1) t r u e
  concatena(.(2, .(3, [])), .(4, .(5, .(6, []))), V4) f a l s e
  concatena(.(2, .(3, [])), .(4, .(5, .(6, []))), V4) t r u e
  concatena(.(3, []), .(4, .(5, .(6, []))), V4) f a l s e
  concatena(.(3, []), .(4, .(5, .(6, []))), V4) t r u e
  concatena([], .(4, .(5, .(6, []))), V4) t r u e
  escribe(X=) X= t r u e
  escribe(.( 1, .( 2, .( 3, .( 4, .(5, .(6, [])))))))
      .(1, .(2, .(3, .(4, .(5, .(6, [])))))) t r u e
  nl
 t r u e
 t r u e
--- TRUE---

F I N   D E   S E S I O N
```

Este es el mismo programa del ejemplo 5 sólo que tiene prendida la bandera de rastreo para seguir el comportamiento del programa.

Ejemplo 7

OBTENCION DE RELACIONES FAMILIARES

```
h(luis).
h(pablo).
h(rolando).
h(samuel).
h(francisco).
h(carlos).
h(juan).
h(gerardo).
m(otilia).
m(abelem).
m(belem).
m(esperanza).
m(clara).
m(guadalupe).
m(lourdes).
m(leticia).
p(samuel,rolando,belem).
p(rolando,luis,otilia).
p(belem,pablo,abelem).
p(francisco,rolando,belem).
p(carlos,rolando,belem).
p(esperanza,rolando,belem).
p(clara,rolando,belem).
p(guadalupe,rolando,belem).
p(lourdes,rolando,belem).
p(leticia,rolando,belem).
p(juan,gerardo,guadalupe).

hno(X,Y):-h(X) & p(X,W,Z) &
       h(Y) & dif(X,Y) & p(Y,W,Z). X es hermano de Y y Y es h
hno(X,Y):-h(X) & p(X,W,Z) & m(Y) & p(Y,W,Z). X es hermano de Y
      y Y es m
hno(X,Y):- ! & escribe("no hay mas hermanos") & nl & halt.

dif(X,X):- ! & fail.
dif(X,Y).
?- hno(X,Y) & escribe(hmno(X,Y)) & nl &  fail.
madre(X,Y):- p(X, _ ,Y) & m(X).
madre(X,Y):-_! & halt.

?- madre(X,Y) & escribe(madre(X,Y)) & nl & fail.

padre(X,Y):-p(X,Y,_) & h(Y).
padre(X,Y):- ! & halt.

?- padre(X,Y) & escribe(padre(X,Y)) & nl & fail.
```

```
PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO

  hmno(samuel, francisco)
  hmno(samuel, carlos)
  hmno(francisco, samuel)
  hmno(francisco, carlos)
  hmno(carlos, samuel)
  hmno(carlos, francisco)
  hmno(samuel, esperanza)
  hmno(samuel, clara)
  hmno(samuel, guadalupe)
  hmno(samuel, lourdes)
  hmno(samuel, leticia)
  hmno(francisco, esperanza)
  hmno(francisco, clara)
  hmno(francisco, guadalupe)
  hmno(francisco, lourdes)
  hmno(francisco, leticia)
  hmno(carlos, esperanza)
  hmno(carlos, clara)
  hmno(carlos, guadalupe)
  hmno(carlos, lourdes)
  hmno(carlos, leticia)
 no hay mas hermanos
---TRUE---
  madre(samuel, belem)
  madre(rolando, otilia)
  madre(belem, abelem)
  madre(francisco, belem)
  madre(carlos, belem)
  madre(esperanza, belem)
  madre(clara, belem)
  madre(guadalupe, belem)
  madre(lourdes, belem)
  madre(leticia, belem)
  madre(juan, guadalupe)
--- TRUE---

  padre(samuel, rolando)
  padre(rolando, luis)
  padre(belem, pablo)
  padre(francisco, rolando)
  padre(carlos, rolando)
  padre(esperanza, rolando)
  padre(clara, rolando)
  padre(guadalupe, rolando)
  padre(lourdes, rolando)
  padre(leticia, rolando)
  padre(juan, gerardo)
--- TRUE---
```

F I N D E S E S I O N

Este programa encuentra las relaciones familiares tipo hermano a partir de especificar las relaciones padre, mujer y hombre. También obtiene quien es la madre y padre para cada uno de los miembros de la familia.

Este ejemplo fue seleccionado para probar el uso del CUT y como se instancian o acotan variables a términos compuestos (functores con parámetros).

Ejemplo 8

```
FUNCTORES COMO PARAMETROS DE FUNCTORES

o(j,b(w,a(e,b))).
?- o(j,b(X,a(Y,b))) & escribe(X) & nl & escribe(Y).
?- o(j,X) & escribe (X).

PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO

   w
   e
--- TRUE---

  b(w,a(e,b))
--- TRUE---

F I N  D E  S E S I O N
```

Ejemplo 9

FUNCTORES COMO PARAMETROS DE FUNCTORS, CON RASTREO

```
?- trace.
o(j,b(w,a(e,b))).
?- o(j,b(X,a(Y,b))) & escribe(X) &nl&escribe(Y).
?- o(j,X) & escribe (X).

PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO

 t r u e
--- TRUE---
  o(j, b(V1,a(V2,b))) t r u e
  escribe(w) w  t r u e
  nl
 t r u e
  escribe(e) e t r u e
--- TRUE---

  o(j,V1) t r u e
  escribe(b(w,a(e,b)))  b(w,a(e,b))  t r u e
--- TRUE---

F I N    D E    S E S I O N
```

Ejemplo 10

SUMATORIA DE LOS NATURALES

```
suma(1,1):- !.
suma(N,Res) :-
          res(N,1,N1) &
           suma(N1,Res1) &
           +(Res1,N,Res).

?-suma(3,X) & escribe("X=") & escribe(X) & nl.

PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO

 X= 6
--- TRUE---
F I N    D E    S E S I O N
```

Este programa suma los N primeros naturales y deja el resultado en el segundo parámetro.

Este ejemplo fue seleccionado para probar la recursión, el uso del cut y las rutinas del sistema.

Ejemplo 11

SUMATORIA DE LOS NATURALES, CON RASTREO

```
?-trace.
suma(1,1):- !.
suma(N,Res) :-
          res(N,1,N1) &
           suma(N1,Res1) &
           +(Res1,N,Res).
?-suma(3,X) & escribe("X=") & escribe(X) & nl.

PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO
 t r u e
--- TRUE---
  suma(3,V1) f a l s e
  suma(3,V1) t r u e
  res(3,1,V3) t r u e
  suma(2,V4) f a l s e
  suma(2,V4) t r u e
  res(2,1,V3) t r u e
  suma(1,V4) t r u e
  ! t r u e
  +(1,2,V2) t r u e
  +(3,3,V2) t r u e
  escribe(X=) X= t r u e
  escribe(6) 6 t r u e
  nl
 t r u e
--- TRUE ---
F I N    D E    S E S I O N
```

Este es el mismo programa del ejemplo 10 pero con rastreo.

Ejemplo 12

```
EFECTO DEL RETRACT EN RUTINAS DEL SISTEMA

a(perro).
?-trace & retract(escribe,1,1).

PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO

 INTENTO DE REDEFINICION DE RUTINA DEL SISTEMA IGNORADO  escribe/1
--- FALSE--

F I N    D E    S E S I O N
```

Este ejemplo fue seleccionado para mostrar el uso de retract[11]. En esta versión del intérprete no se permite la redefinición de rutinas del sistema. Esto se hizo para proteger al programador de posibles errores.

---

[11] retract es un predicado interconstruido que permite borrar cláusulas (en la base de datos del programa) en el momento de ejecución de los programas.

Ejemplo 13

RETRACT Y ASSERT DE CLAUSULAS DEFINIDAS POR EL USUARIO

```
a(1).
a(perro).
a(3).
?-a(X) & escribe(X) & nl & fail.
?-retract(a,1,1).
?- a(X) & escribe(X) & nl & fail.
?-assert(a(gato),[],2).
?- a(X) & escribe(X) & nl & fail.

PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO

 X=1
 X=perro
 X=3
--- FALSE---
--- TRUE---
 X=perro
 X=3
--- FALSE---
--- TRUE---
 X=perro
 X=3
 X=gato
--- FALSE---

F I N   D E   S E S I O N
```

Este ejemplo fue seleccionado para mostrar el uso de assert[2] y retract de cláusulas definidas por el usuario.

---

[2] assert es un predicado interconstruido que permite insertar nuevas cláusulas a la base de datos, en el momento de ejecución de los programas.

Ejemplo 14

RETRACT Y ASSERT DE CLAUSULAS DEFINIDAS POR EL USUARIO, CON RASTREO

```
a(1).
a(perro).
a(3).
?-trace.
?-a(X) & escribe(X) & nl & fail.
?-retract(a,1,1).
?- a(X) & escribe(X) & nl & fail.
?-assert(a(gato),[],2).
?- a(X) & escribe(X) & nl & fail.

PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO

 t r u e
--- TRUE---
  a(V1) t r u e
  escribe(X=) X= t r u e
  escribe(1) 1 t r u e
  nl
 t r u e
  fail f a l s e
  a(V1) t r u e
  escribe(X=) X= t r u e
  escribe(perro) perro t r u e
  nl
 t r u e
  fail f a l s e
  a(V1) t r u e
  escribe(X=) X= t r u e
  escribe (3) 3 t r u e
  nl
 t r u e
  fail f a l s e
--- FALSE---
  retract(a,1,1) t r u e
--- TRUE---
  a(V1) t r u e
  escribe(X=) X= t r u e
  escribe(perro) perro t r u e
  nl
 t r u e
  fail f a l s e
  a(V1) t r u e
  escribe(X=) X= t r u e
  escribe(3) 3 t r u e
  nl
```

```
 t r u e
  fail f a l s e
--- FALSE---
  assert(a(gato),[],2) t r u e
--- TRUE---
  a(V1) t r u e
  escribe(X=) X= t r u e
  escribe(perro) perro t r u e
  nl
 t r u e
  fail f a l s e
  a(V1) t r u e
  escribe(X=) X= t r u e
  escribe(3) 3 t r u e
  nl
 t r u e
  fail f a l s e
  a(V1) t r u e
  escribe(X=) X= t r u e
  escribe(gato) gato t r u e
  nl
 t r u e
  fail f a l s e
--- FALSE---
F I N   D E   S E S I O N
```

Ejemplo 15

ASSERT DE RUTINAS CON CUERPO

```
a(1).
a(perro).
a(3).
?-a(X) & escribe("X=") & escribe(X) & nl & fail.
?-retract(a,1,1).
?-a(X) & escribe("X=") & escribe(X) & nl & fail.
?-assert(a(G),[is(G,25),escribe("entre al cuerpo")],2).
?-a(X) & escribe("X=") & escribe(X) & nl & fail.

 PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO
 X= 1
 X= perro
 X= 3
--- FALSE---
--- TRUE---
 X= perro
 X= 3
--- FALSE---
--- TRUE---
 X= perro
 X= 3
 entre al cuerpo X= 25
--- FALSE---
F I N    D E    S E S I O N
```

Ejemplo 16

ASSERT DE RUTINAS CON CUERPO, CON RASTREO

```
a(1).
a(perro).
a(3).
?-trace & a(X) & escribe("X=") & escribe(X) & nl & fail.
?-retract(a,1,1).
?-a(X) & escribe("X=") & escribe(X) & nl & fail.
?-assert(a(G),[is(G,25),escribe("entre al cuerpo")],2).
?-a(X) & escribe("X=") & escribe(X) & nl & fail.

PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO
 t r u e
  a(V1) t r u e
  escribe(X=) X= t r u e
  escribe(1) 1 t r u e
  nl
 t r u e
  fail f a l s e
  a(V1) t r u e
  escribe(X=) X= t r u e
  escribe(perro) perro t r u e
  nl
 t r u e
  fail f a l s e
  a(V1) t r u e
  escribe(X=) X= t r u e
  escribe(3) 3 t r u e
  nl
 t r u e
  fail f a l s e
--- FALSE---
  retract(a,1,1) t r u e
--- TRUE---
  a(V1) t r u e
  escribe(X=) X= t r u e
  escribe(perro) perro t r u e
  nl
 t r u e
  fail f a l s e
  a(V1) t r u e
  escribe(X=) X= t r u e
  escribe(3) 3 t r u e
  nl
 t r u e
  fail f a l s e
--- FALSE---
```

```
  assert(a(V1),.(is(V1,25),.(escribe(entre al cuerpo),[])),2)
        t r u e
--- TRUE---
  a(V1) t r u e
  escribe(X=) X= t r u e
  escribe(perro) perro t r u e
  nl
 t r u e
  fail f a l s e
  a(V1) t r u e
  escribe(X=) X= t r u e
  escribe(3) 3 t r u e
  nl
 t r u e
  fail f a l s e
  a(V1) t r u e
  is(V1,25) t r u e
  escribe(entre al cuerpo) entre al cuerpo t r u e
  escribe(X=) X= t r u e
  escribe( 25) 25 t r u e
  nl
 t r u e
  fail f a l s e
--- FALSE---
F I N   D E   S E S I O N
```

APENDICE 2

MANUAL PARA EL USUARIO

El intérprete está escrito en C. Actualmente se ejecuta en una computadora Onyx bajo el sistema operativo Unix, versión 7, y en una PC compatible con IBM bajo Unix, sistema III.

El intérprete puede usarse de dos formas:

1. Interactiva

2. No interactiva

Forma interactiva

Para usar el intérprete en forma interactiva llamar a PROLOG tecleando g, con lo cual PROLOG responderá

```
PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO
```

PROLOG queda a la espera de la definición de los axiomas y teoremas a ser probados.

Se puede añadir nuevas cláusulas que serán compiladas dinámicamente y sumadas a la base de datos de PROLOG.

La forma se salir del modo interactivo es mediante dos Control/C con lo cual se terminará la sesión de PROLOG y se mandará un mensaje:

F I N D E S E S I O N

Forma no interactiva

En este segundo método se usan archivos creados con el editor de Unix. La forma de llamar a PROLOG es

```
g ‹nom_arch›
 ó
mp ‹nom_arch›
```

a lo cual PROLOG responderá con

```
PROLOG 1.1  D.R. 5/17/87  por:
Ma. S. Vargas
iimas-unam, MEXICO



Operaciones válidas:

predicados aritméticos: ‹  = ‹  ›=  =
operadores aritméticos: +  -  *  / mod

predicados interconstruidos:
lee
escribe
nl
assert
retract
call
halt
debug
nodebug
!
fail
stop
```

Para información de la sintaxis de cada uno de los

predicados interconstruidos (véase apartado 3.7) y para ver la forma de construir programas en PROLOG (véase el apartado 2.1.3).

APENDICE 3

ESTRUCTURA DEL INTERPRETE

El intérprete de PROLOG fue hecho utilizando herramientas de Unix como Yacc que es un generador de parser, Make que permite controlar la compilación de los módulos del intérprete y SCCS que permite llevar un control de versiones de los programas.

El intérprete fue desarrollado en una computadora Onyx bajo el sistema operativo Unix versión 7. Tiene 6500 lineas de código escrito en el lenguaje "C" y tirne aproximadamente 100 rutinas. El programa objeto es de 51 Kb.

Del total de rutinas del intérprete se hizo el árbol de rutinas hasta el nivel 3 (véase figura A3.1).

Lista de rutinas en orden alfabético

abre: Abre un archivo de carga inicial.

back: Rutina que ejecuta backtracking.

checabc: Checa que las metas a resolver esten bien construidas esto es que una meta no sea variable o entero.

crgainal: Carga las rutinas del sistema desde un archivo llamado "nucleo", les construye su entrada en

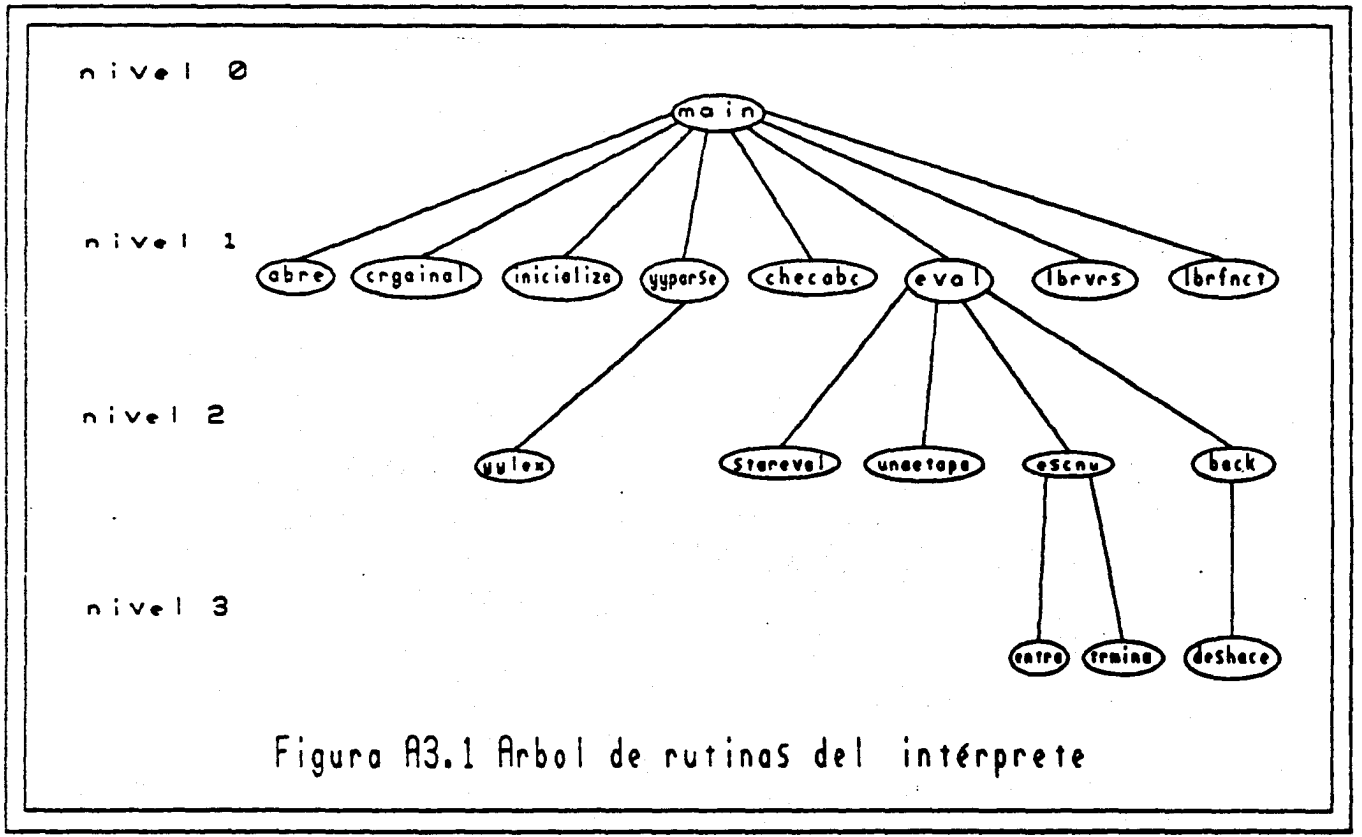

Figura A3.1 Arbol de rutinas del intérprete

el diccionario y su descriptor como rutina del sistema.

deshace: En caso de backtracking deshace las referencias.

escnu: Checa si es una cláusula no unitaria o cláusula unitaria esto es determina si es regla o hecho.

eval: Se encarga de evaluar una pregunta de PROLOG.

inicializa Inicializa variables globales.

lbrvrs: Libera el espacio usado para las variables una vez que se termina la ejecución o si un error ocurre.

lbrfnct: Libera el espacio usado por los esqueletos de de los functores.

stareval: Hace la iniciación de las pilas para la ejecución de una pregunta.

unaetapa: Realiza la unificación de la cabeza de la cláusula.

yylex: Rutina que obtiene los tokens.

yyparse: Rutina que reconoce la gramática de PROLOG.

A continuación se presentan el archivo makefile, main del programa, rutina spacio, protype y eval.

```
# @(#)makefile  1.1-5/16/87 D.R. Mexico M.S.VARGAS

g:y.tab.o lex.o dicc.o carga.o heap.o mt.o stack.o memoria.o
       eval.o checa.o unif.o trace.o rutsis.o rut2sis.o
       time cc -i -O  -o g y.tab.o lex.o dicc.o carga.o heap.o
       mt.o stack.o memoria.o eval.o checa.o unif.o trace.o rutsis.
rut2sis.o y.tab.c :gb.y def.h
       time yacc -vd gb.y
y.tab.o :def.h
       time cc -c -i -O   y.tab.c
lex.o   :def.h
       time cc -c -i -O   lex.c
dicc.o  :def.h
       time cc -c -i -O    dicc.c
heap.o  :def.h
       time cc -c -i -O   heap.c
mt.o    :def.h
       time cc -c -i -O   mt.c
stack.o : def.h
       time cc -c -i -O  DDBG stack.c
carga.o:carga.c
       time cc -c -i -O  carga.c
memoria.o:memoria.c
       time cc -c -i -O   memoria.c
eval.o   :eval.c
       time cc -c -i -O    eval.c
checa.o  :def.h
       time cc -c -i -O   checa.c
unif.o  :unif.c
       time cc -c -i -O    unif.c
trace.o :trace.c
       time cc -c -i -O   trace.c
rutsis.o :rutsis.c
       time cc -c -i -O   rutsis.c
rut2sis.o :rut2sis.c
       time cc -c -i -O   rut2sis.c
```

```
/*     Main del Programa        */
main()
{
   abre("nucleo");
   crgainal();
   ioptr=stdin;
   if (setjmp(eof)) {
        printf("F I N    D E    S E S I O N");
    }
   else {
         while (TRUE) {

            while ( ! meta ) {
               inicializa();
                yyparse();
                if (niv0.cpo != NULL) {
                    checabc();
                    if ( !meta ){
                       lbrvrs(niv0.ptlv);
                       niv0.ptlv = NULL;
                    }else ;
                 }
                 else ;
             }  /* termina while */
   /* ya encontro pregunta y ahora hay que resolverla */
   eval(aproto,niv0.nvar);
   meta = FALSE;
   lbrvrs(niv0.ptlv);
   lbrfnct(niv0.cpo);
         }   /* termina while TRUE */
   } /* termina setjmp */
}   /* termina el main */
```

```
/* da espacio para construir el prototipo  */
hyr *spacio(tam)
int tam;
{
  hyr *ap;
  if ((ap=(hyr *)maloc(sizeof(hyr)*tam))==NULL) {
     er(3,"ovf en memoria no puedo construir  prototipo");
     return(NULL);
  }
  return(ap);
} /* termina spacio */



/* construye el prototipo del tama&o que se le indica en aridad más
   uno y llena los campos de ese prototipo */
protype(aridad,dir)
int aridad;
int dir;
{

 int tam;
 int i;
 hyr *ap;

    tam=(aridad)+1;
    if (tam >1) {
         /* pido espacio para el prototipo */
        if((ap=spacio(tam))==NULL)  return(OVFMEM);
        else ;
    ap->pdic = dir;  /* lleno nombre del functor */
        /* lleno los argumentos para ese nombre de functor */
    for(i=aridad;i>=1;i--) {
      top=top-1;
      ap->a[i].tipo=params[top].tpo;
      ap->a[i].args=params[top].campo;
    } /* termina for */
    }
    else { /* el nombre del functor no tiene parametros */
       /* pido espacio para el prototipo */
    if((ap=spacio(tam))==NULL) return(OVFMEM);
    else ;
    ap->pdic = dir;  /* lleno nombre del functor */
    }
aproto=ap;

} /* termina protype */
```

```
/* eval ejecuta una pregunta */

eval(goalseq,nvargoal)
 union hyr *goalseq;
 int nvargoal;
{
   stareval(goalseq,nvargoal);
   deten=FALSE;
   exito=TRUE; /* falsa solo cuando hay fracaso */
   do {
      unaetapa(&exito,&deten);
      if (! deten) {
           if (! exito) {
             back(&deten);
           }
           else {
            if (escnu( cproc))  entra();
             else
                  trmina(&deten);
           }
       }
       else ;

   } while (! deten);
if (exito)
  printf("--- TRUE---");
else
  printf("--- FALSE---");

} /* termina eval */
```

APENDICE 4

GLOSARIO DE TERMINOS

META. Recibe el nombre de meta la llamada a un procedimiento dentro de la definición de una cláusula. Ejemplo, si definimos la siguiente cláusula A :- B1,B2,...,BN, las cláusulas Bi son conocidas como metas o llamadas a procedimientos.

OBJETO. Todos los objetos de PROLOG son términos y los términos pueden ser:

Cláusulas

Estructuras o términos compuestos

Variables

Constantes

INSTANCIACION. Es un término usado cuando nos referimos a una variable. Una variable es instanciada cuando existe un objeto al que se encuentra acotada. Si no es asi, se le denomina no instanciada.

PREDICADO. Se llama predicado al nombre de un procedimiento.

BIBLIOGRAFIA

[AHOA77]

Aho, A., J.D., Ullman, Principles of Compiler Design, Addison Wesley, 1979.

[BATD84]

Batory D. S., A. P., Buchmann, "Molecular Objects, Abstract Data Types and Data Models a Framework", 10th. International Conference on Very Large Data Bases, Singapore, 1984.

[BUCA86]

Buchmann A P, R.S, Carrera, M.A., Vazquez-Galindo, "A generalized constraint and exception handler for an object oriented CAD-DBMS", International workshop on object-oriented database systems, Pacific Grove Califonia, 1986.

[BRUM82]

Brugnooghe, M., "The Memory management of Prolog Implementation", Logic Programming, Academic Press, 1982.

[CHRK83]

Crhistian, Kaare, The UNIX Operating System, John Wiley & Sons, 1983.

[COEH82]

Coelho H., J.C., Cotta, L.M., Pereira, How to solve it with Prolog Laboratorio Nacional de Engenharia Civil, 1982.

[CLOW81]

Clocksin, W. F., C.S, Mellish, Programming in Prolog, Springer Verlag, 1981.

[COLM79]

Colmerauer, A., Kanoui, H., Van Caneghem M., Etude et réalisation d'un systema Prolog, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, 1979.

[DEYL84]

Deyi, L.,A Prolog database system,John Wiley & Sons, 1984.

[EMDM82]

Van Emden, M.H., "An Interpreting Algorithm for Prolog programs", First International Logic Programming Conference, Marseille, 1982.

[HOPJ69]

Hopcroft, J., J., Ullman,Formal languages and their relation to automata, Addison Wesley, 1969.

[JHOS81]

Jhonson, S.C., "Yacc: Yet Another Compiler-Compiler", Unix programmers manual, 1978.

[KOWR74]

Kowalski R. A., "Predicate Logic as Programming Language",Proceedings of the IFIP Congress, North-Holland, Amsterdam, 1974.

[MELC82]

Mellish C. S., "An Alternative to Structure-sharing in the Implementation of a Prolog Interpreter, Logic Programming, Academic Press, 1982.

[MONL78]

Moniz, Luis, Fernando C., Pereira, David H, Warren, User's guide to Decsystem-10 Prolog, Reporte interno de la universidad de Edinburgo, 1978.

[ROBJ65]

Robinson J. A., "A machine-oriented logic based on the resolution principle", Journal of the Association for Computing Machinery Vol. 12, No. 1., 1965.

[SANE82]

Santana-Toth E., " Prolog applications in Hungary", Logic Programming, Academic Press, 1982.

[STAT80]

Standish Thomas; Data Structure Techniques, Addison Wesley, 1980.