UNIVERSIDAD NACIONAL AUTONOMA
DE MEXICO

Unidad Académica de los Ciclos Profesional y
de Posgrado del C.C.H.

# MODULOS EN LENGUAJES DECLARATIVOS

**T E S I S**

PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS DE LA COMPUTACION

P R E S E N T A :

GUSTAVO ARTURO MARQUEZ FLORES

DIRECTORA DE TESIS,
Dra. Cristina Loyo Varela

MEXICO, D. F. ENERO DE 1993


**UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso**

C O N T E N I D O .

# I N T R O D U C C I O N .

El diseño y desarrollo de nuevos tipos de lenguajes de programación adecuados a las necesidades de los distintos problemas surgidos dentro de la computación, ha sido una preocupación central desde el nacimiento de la primera computadora. En los últimos años, esta preocupación ha sido motivada particularmente por la búsqueda, dentro de los lenguajes de programación, de características que permitan una mayor abstracción y expresividad. Así, se ha generado toda una línea de lenguajes que se agrupan bajo el nombre de *lenguajes declarativos*.

A diferencia de los llamados *lenguajes imperativos* que reflejan la arquitectura clásica de una computadora, los lenguajes declarativos poseen por lo general un modelo computacional abstracto y sólidamenente fundamentado. Los lenguajes declarativos permiten al programador abocarse más a la especificación del problema que a la descripción de su ejecución, expresan de manera natural las propiedades de comportamiento entre funciones y relaciones como entidades del lenguaje.

La mayoría de las características de los lenguages de tipo declarativo han sido motivadas por la problemática inherente a la inteligencia artificial, así han surgido los lenguajes *LISP* y *ML* como lenguajes funcionales; *PROLOG* como exponente de la programación lógica para el manejo de relaciones y de deducción, toda la línea de lenguajes orientados a objetos para expresar conocimiento y más recientemente ha surgido la necesidad de diseñar lenguajes híbridos que conjugan características de varios de ellos.

El presente trabjo se inscribe dentro del marco de un proyecto de investigación general sobre el diseño de un lenguaje lógico con manejo de tipos, objetos y módulos. Este proyecto se desarrolla conjuntamente entre el Laboratorio Nacional de Informática Avanzda, *LANIA* y la Facultad de Cibernética Matemática de la Universidad de la Habana.

El problema particular de la tesis se aboca a: a) el estudio

y análisis de las distintas propuestas de manejo de módulos que se han hecho para lenguajes declarativos, así como al análisis de las características de manejo de módulos en otro tipo de lenguajes que pueden ser interesantes para *PROLOG;* b) el diseño de una propuesta para la inclusión de módulos en *PROLOG* y 3) la programación de un prototipo de prueba que muestre la viabilidad de la propuesta.

El alcance de este trabajo no abarca la justificación lógica de la propuesta de manejo de módulos que se hace, ni la integración dentro del diseño del lenguaje de la investigación.
Este prototipo propuesto pretende básicamente, servir como un esquema de experimentación para explorar y usar los distintos conceptos de modularidad dentro de *PROLOG*.

El trabajo que aquí se presenta está organizado de la siguiente manera. En el primer capítulo, se introduce el concepto de módulo, su objetivo y características generales. Se hace también un análisis del tratamiento del concepto de módulo en varios lenguajes de programación, considerados como de mayor interés por su conceptualización y uso de los módulos. Estos lenguajes son *ML*, *OBJ3* y *ADA*.

En el segundo capítulo, se presentan las alternativas de diseño de sistemas de módulos que se han realizado para el lenguaje *PROLOG*.

En el tercer capítulo, se presenta el diseño del sistema de módulos propuesto para *PROLOG* con sus objetivos, características y elementos del sistema.

En el cuarto capítulo, se trata de dar una idea de la forma en que se llevó a cabo la programación del prototipo.

Finalmente, a manera de conclusiones se analizá las limitaciones, alcances y posibles mejoras del sistema de módulos desarrollado.

# C A P I T U L O  P R I M E R O

## El Concepto de Módulo.

### 1.1 Definición.

El concepto de módulo es usado para hacer referencia a un conjunto de entidades contiguas de un programa, las cuales reciben un nombre a través del cual se pueda hacer referencia a ellas. Estas entidades pueden ser desde simples declaraciones de tipos, variables y constantes, hasta procedimientos, funciones, código de programa o un mismo módulo. Este conjunto de entidades se caracteriza por realizar tareas específicas y poseer conexiones y límites bien definidos con el medio ambiente de trabajo.

### 1.2 Ventajas.

En la teoría general de sistemas, el concepto de módulo juega un papel central, al sugerir el desarrollo de sistemas estructurados o modulares, que permitan descomponer un sistema o problema de considerable complejidad en conjuntos de módulos. Con esto, se facilita la construcción de grandes sistemas en la que intervienen grupos de programadores. Esto da la ventaja de mejorar la organización del sistema, ocupándose un grupo de programadores en desarrollar su propio módulo sin tener que preocuparse por el resto del sistema.

La organización modular permite además, realizar inspecciones y correcciones a un programa o sistema, aislando las fallas más rápidamente y facilitando su depuración, sin afectar los otros módulos que están trabajando correctamente.

## 1.3 Objetivos de un Sistema de Módulos.

Los objetivos de un sistema de módulos en un lenguaje de programación, pueden resumirse como sigue:

- Descomponer código en bloques estructurados pequeños y simples.
- Proporcionar medios de abstracción.
- Facilitar la comunicación entre segmentos de código.
- Regular la visibilidad y la existencia de los elementos internos de un bloque.
- Permitir código reusable de programas.

Algunos de los objetivos que persigue un sistema de módulos, como la visibilidad, evitar la repetición de código, se han logrado en algunos lenguajes de programación, como *Pascal, C, ADA, ML, MODULA-2,* etc., a nivel de control de flujo mediante el empleo de procedimientos, funciones o macros.
Otros objetivos, como el manejo de la abstracción y comunicación entre los módulos, se han logrado a nivel de manejo de tipos de datos abstractos y de módulos mismos, introduciendo conceptos como implantaciones y declaraciones de programas parametrizadas, programación orientada a objetos, tipos genéricos, etc..

## 1.4 Características.

Desde hace algun tiempo, a partir de los años setenta aproximadamente, hay un concenso, más o menos general, de la estructura de un módulo.

Un módulo se presénta generalmente en dos partes, una llamada parte declarativa y otra llamada parte ejecutable. Otros autores como Drossopoulou *[ Drossopoulou 88 ]*, llaman a la parte declarativa interfaz e implantación a la parte de ejecución.

En la siguiente figura, se muestra en forma esquemática la estructura de un módulo.

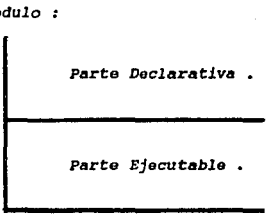



fig. I.1 Estructura general de un Módulo.

La parte declarativa consta de un conjunto de declaraciones que puede verse como la *declaración sintáctica* de las entidades que componen el módulo. La parte ejecutable corresponde al conjunto de instrucciones ejecutables de al menos, de las entidades computacionales especificadas en la parte declarativa. Estas definen la *especificación semántica* del módulo. En algunos casos, la parte declarativa contiene también información semántica.

La parte declarativa sirve también para describir la visibilidad, y el tiempo de existencia de las entidades del módulo, así como el quehacer del módulo. Además, cumple con dos objetivos, por un lado establece explícitamente las entidades que un módulo proporciona al medio exterior, lo que se conoce como exportación, y por otra parte, oculta los detalles irrelevantes al impedir el acceso a entidades del módulo no incluidas en la parte declarativa.

La parte de ejecución de un módulo define el cómo se va a realizar algo en el módulo. Contiene el medio ambiente de las entidades definidas en la parte declarativa, además de contener, posiblemente, la implantación de los tipos de datos definidos también.

En ambas partes de un módulo, la visibilidad y la existencia de entidades se logran de dos maneras: 1) el módulo puede elegir dentro del medio ambiente donde se encuentra, todas aquellas entidades que necesite, realizando dos mecanismos conocidos como

importación y parametrización; y 2) el módulo puede sólo dar a conocer al medio ambiente exterior, aquellas entidades que considere indispensables o necesarias, por medio también de un mecanismo conocido como exportación.

Idealmente, el número de entidades parametrizadas o importadas debe ser el mínimo requerido, y deben transmitir solamente la información considerada como indispensable para la comunicación del módulo con su medio exterior.

### 1.5 Ventajas de la Estructuración de un Módulo.

La ventaja que representa estructurar un módulo en dos partes, es que permite el desarrollo de programas estructurados y jerárquicos, en donde los módulos más generales asumen o importan a los de menor jerarquía. Esta organización facilita la solución de un problema abstrayéndolo por niveles.

Otra ventaja importante, es que es fácil modificar cualquier parte de ejecución que se desee del sistema, sin tener que modificar y recompilar los demás módulos, ni la correspondiente parte declarativa. Ya que es más frecuente modificar la parte de ejecución de un módulo que su parte declarativa. De esta forma, los programadores pueden diseñar y compilar el módulo principal del sistema y las partes declarativas, sin tener que preocuparse de cómo se va a escribir la parte de ejecución de los demás módulos.

El concepto de módulo, así como la idea de dividirlo en dos partes, surge en la programación con el lenguaje *SIMULA* en 1966 *[ Dahl 66 ]*. Este lenguaje, cuyo nombre es la abreviación de *SIMUlation LAnguage*, fué diseñado para describir formalmente las reglas de operación de sistemas definidos a través de eventos discretos. En *SIMULA*, un evento discreto, es definido mediante el concepto de proceso, el cual es un módulo. En general, un proceso tiene dos partes: una donde se describen mediante una secuencia de especificaciones y declaraciones (parte declarativa), los elementos que participan en un evento y otra parte donde se describen las acciones del evento mediante una secuencia de instrucciones, llamadas reglas de operación (parte ejecutable).

A partir del surgimiento del concepto de módulo en el lenguaje *SIMULA*, ha habido otros lenguajes y conceptos que lo han enriquecido aún más. A continuación se describirán los sistemas de módulos más sobresalientes, dentro de los lenguajes de programación.

## I.6 El Concepto de Módulo en Algunos Lenguajes de Programación.

### 1.6.1 El Concepto de Módulo en el Lenguaje *ADA*.

*ADA* es un lenguaje diseñado en 1979, cuyo nombre hace honor a *Augusta Ada Byron,* Condesa de Lovelace, hija del poeta *Lord Byron* y a quien se le considera la primera programadora del siglo XIX.

La idea de módulo en *ADA [ Ada 83 ]* surge de la necesidad de agrupar un conjunto de entidades reusables para varias aplicaciones. A esta colección de entidades relacionadas, se le denomina *paquete*. Estas entidades pueden ser además, subprogramas y declaración de exepciones. Un subprograma es un procedimiento o una función.

Un *paquete genérico* es un módulo parametrizado, es decir, un *paquete* en el cual se tiene la posibilidad de usarlo varias veces con distintos valores de sus parámetros, como si se tratara de una función o procedimiento. Los tipos de parámetros con los cuales se puede usar un *paquete genérico*, son variables, constantes, tipos, subtipos, procedimientos y funciones. Antes de usar un *paquete genérico* con los valores deseados, se debe indicar qué tipos tienen estos valores y cómo se corresponden con los parámetros del *paquete genérico*. Este proceso se conoce como instanciación del módulo. Así, un *paquete genérico* puede ser instanciado de diferentes maneras para resolver diferentes problemas.

Un *paquete genérico* consiste de una parte declarativa, llamada *especificación de paquete* y otra ejecutable, llamada el *cuerpo del paquete*. En la parte declarativa se especifican además los parámetros del *paquete*. La parte declarativa consta de la especificación de encabezados de procedimientos o funciones, así como de los tipos, variables y constantes que utiliza el módulo. Mientras que el *cuerpo del paquete*, consta de la declaración del cuerpo de esos procedimientos o funciones definidos en la especificación.

En seguida se muestra un ejemplo de la forma sintáctica de la *especificación* y el *cuerpo de un paquete*:

```
GENERIC

   Declaración de parámetros

   PACKAGE Nombre del Paquete Genérico  IS
      Encabezado de procedimientos y funciones
   PRIVATE

      Declaración de tipos Privados

END Nombre del Paquete Genérico .
```

fig. I.2 Parte declarativa de un Paquete Genérico.

```
PACKAGE BODY Nombre de Paquete Genérico  IS

   Declaración de procedimientos y funciones

END Nombre del Paquete Genérico
```

fig. I.3 Parte ejecutable de un Paquete Genérico.

Las dos partes de un *paquete genérico* pueden ser definidas en forma independiente, formando así unidades de compilación.

Un ejemplo de un *paquete genérico* que especifica el tipo de dato abstracto pila, en donde el tamaño y el tipo de elementos a manejar son parámetros del *paquete*, es el siguiente :

```
GENERIC
 Tam_Pila : IN Integer ;
 TYPE Tipo_Elemento is PRIVATE ;

 PACKAGE Pila_Generica IS

    TYPE Pila is PRIVATE ;
     PROCEDURE Mete( Elemento : IN Tipo_Elemento ;
                     laPila : IN OUT Pila ) ;

     PROCEDURE Saca( Elemento : OUT
                     Tipo_Elemento; laPila : IN OUT Pila ) ;
```

```
   PRIVATE
      TYPE Lista IS
       ARRAY( 1 .. Tam_Pila ) OF Tipo_Elemento ;

      TYPE Pila IS
       RECORD
          Apuntador : Integer RANGE 0 .. Tam_Pila := 0 ;
          Elements : Lista ;
       END RECORD ;
   END Pila_Generica ;


   PACKAGE BODY Pila_Generica IS

    PROCEDURE Mete( Elemento : IN Tipo_Elemento ;
                    laPila : IN OUT Pila ) IS
         BEGIN
            laPila.Apuntador := laPila.Apuntador + 1 ;
            laPila.Elements( laPila.Apuntador ) := Elemento ;
         END Mete ;

    PROCEDURE Saca( Elemento : OUT Tipo_Elemento ;
                    laPila : IN OUT Pila ) IS
         BEGIN
            Elemento := laPila.Elements( laPila.Apuntador ) ;
            laPila.Apuntador := laPila.Apuntador - 1 ;
         END Saca ;

   END Pila_Generica ;
```

Las siguientes instrucciones muestran tres posibles instanciaciones de este *paquete genérico*:

```
PACKAGE Pila_De_Booleanos IS
 NEW Pila_Generica( Tam_Pila => 36, Tipo_Elemento => Boolean ) ;

PACKAGE Pila_De_Caracteres IS
 NEW Pila_Generica( Tam_Pila => 255, Tipo_Elemento => Charac-
                                                   ter ) ;

PACKAGE Pila_De_Enteros IS
 NEW Pila_Generica( Tam_Pila => 10, Tipo_Elemento => Integer ) ;
```

En el primer caso, se tiene una pila de booleanos de 36 elementos, en el segundo una pila de caracteres con 255 elementos y en el último una pila de enteros con solo 10 elementos. Cada una de estas instanciaciones forma un *paquete* independiente, el cual puede ser compilado como una unidad, y cualquier otro puede accesar los procedimientos ahi definidos, que son *Saca* y *Mete*, se-

gun el tipo de elemento de la pila que se desee manejar.

Una instanciación puede ser declarada también en el cuerpo de un procedimiento o función, o en la *especificación* y *cuerpo de un paquete*. Esto da la facilidad de crear una instancia de un módulo que puede usarse en el mismo lugar que es creado y es local en ese contexto.

### 1.6.2 El Concepto de Módulo en el Lenguaje *ML*.

*ML* es un lenguaje funcional diseñado por *Robin Milner* en la Universidad de Edimburgo en 1978 *[ Harper 86 ]*. El nombre del lenguaje significa *MetaLanguaje*. Entre las principales características del lenguaje destacan las siguientes:

- Posee un sistema polimórfico de tipos. Es decir, dada una expresión del lenguaje, tiene asociada el tipo más general que se determina de modo único por el contexto en que se usa.
- Soporta la definición de tipos de datos abstractos. Este es un mecanismo que permite la abstracción y desarrollo de programas modulares, permitiendo ocultar información también.
- Posee un mecanismo para manejar excepciones cuando suceden condiciones anormales al ejecutar los programas.

Los elementos básicos que componen el sistema de módulos en *ML* son estruturas (*structure*), signaturas (*signature*) y funtores (*functor*).

Una signatura en *ML*, corresponde a lo que se llamó en la sección 1.4 de este capítulo, la parte declarativa de un módulo, mientras que una estructura corresponde a la parte ejecutable del módulo. *ML* introduce también el concepto de funtor, cuyo objetivo principal es servir como medio de abstracción en la construcción de módulos parametrizados, para permitir la generación de estructuras parametrizables de una manera más formal. Un funtor recibe como parámetros estructuras, las cuales definen la implantación (ejecución) de una abstracción que es especificada como una signatura, la cual deben satisfacer los parámetros del funtor. Así, empleando signaturas y estructuras se asegura la se-

paración entre abstracción e implantación, mientras que un funtor establece la conexión entre esa abstracción y la implantación definidas.

Una signatura especifica los tipos que requiere una estructura y define las funciones con sus tipos que han de implantarse en una estructura. La sintaxis de una signatura es la siguiente:

```
signature Nombre de la signatura =

  sig
       Declaración de tipos

       Declaración del nombre y los tipos
       de los argumentos de las funciones

  end ;
```

Una estructura es un medio ambiente local en el que se asignan valores a los tipos o variables declarados en una signatura, siguiendo el mismo orden en que aparecen declarados.
Cuando existe esta correspondencia entre ellos, se dice que la estructura satisface la signatura dada. Dentro de una estructura, pueden también declararse nuevos tipos y sus valores correspondientes, así como establecer cual es la implantación de las funciones definidas en una signatura. La sintaxis de una estructura es la siguiente:

```
structure Nombre de la estructura : Nombre de una signatura

    struct

       Declaración de tipos

       Declaración de funciones

    end ;
```

Un funtor es una función que recibe una o varias estructuras como parámetros y regresa otra estructura. Ambas estructuras deben satisfacer una signatura definida previamente y que es especificada en los parámetros del funtor. La utilidad básica del funtor es implantar las funciones definidas en las signaturas especificadas como parámetros. Las estructuras que recibe como argumento, sirven para parametrizar los tipos de las funciones de la estructura que regresa el funtor. Los funtores no tienen una

signatura asociada, lo cual impide que la parametrización de funtores de orden superior sea posible. La sintaxis de un funtor es la siguiente:

```
functor Nombre del funtor ( Parámetro : Signatura ) :
                            Signatura =

 struct

    Declaración de tipos

    Declaración de funciones

 end ;
```

Para mostrar el empleo de signaturas, estructuras y funtores en *ML*, se describe a continuación un conjunto de módulos que especifican e implantan el tipo de dato abstracto cola para cualquier tipo de elemento. La siguiente signatura, llamada *COLA*, primeramente define este tipo de dato abstracto:

```
signature COLA =
   sig
      type elemento
      type cola

         val cola_vacia : cola
         val mete_ult : ( elemento * cola ) -> cola
         val resto : cola -> cola
         val primero : cola -> elemento
         val es_vacia : cola -> bool
end ;
```

La siguiente signatura especifica el tipo genérico con que puede usarse la cola:

```
signature ELEMENTO =
   sig
      type elemento
end ;
```

Finalmente, el siguiente funtor implanta el tipo de dato abstrac-

to cola :

```
functor Cola( Elem : ELEMENTO ) : COLA =
  struct
    type elemento = Elem.elemento
         datatype  cola = cola_vacia | mete_ult of
                          ( elemento * cola )
    exception Resto

    fun resto cola_vacia = raise Resto |
        resto( mete_ult( e, cola_vacia ) ) = cola_vacia |
        resto( mete_ult( e, c ) ) = mete_ult( e, resto( c ) )

    exception Primero

    fun primero cola_vacia = raise Primero |
        primero( mete_ult( e, cola_vacia ) ) = e |
        primero( mete_ult( e, c ) ) = primero( c )

    fun es_vacia cola_vacia = true |
        es_vacia _ = false

end ;
```

Si se desea ahora, por ejemplo, construir una cola de enteros, se debe primero definir el parámetro que recibe el funtor *Cola* para usarla. Este parámetro es una estructura que debe satisfacer, como en el encabezado del funtor se establece, la signatura *ELEMENTO*. Una estructura que satisface a *ELEMENTO*, es la siguiente:

```
structure Enteros : ELEMENTO =
    struct
        type elemento = int
end ;
```

Ahora se debe instanciar el funtor *Cola* con la estructura *Enteros*, para que la cola de enteros quede definida. La manera de hacer la instanciación es la siguiente:

```
structure Cola_Ent = Cola( Enteros )
```

De manera similar, si ahora se desea definir una cola de caracteres, se hace lo siguiente:

```
structure Caracteres : ELEMENTO =
    struct
        type elemento = string
end ;

structure Cola_Car = Cola( Caracteres )
```

Comparando el sistema de módulos de *ML* con el de *ADA*, *ML* introduce conceptos más abstractos como son los funtores . Este último concepto da al sistema de módulos una gran ventaja, permitiendo parametrizar además de tipos, variables o constantes también módulos. Este mecanismo de parametrizar el concepto de módulo, tiene la ventaja de que los módulos pueden así ser reutilizados por otros módulos.

### 1.6.3 El Concepto de Módulo en el Lenguaje *OBJ3*.

*OBJ* es un lenguaje declarativo que fue desarrollado en el Stanford Research Institute, por *Timothy Winkler, Jose Meseguer* y *Joshep Goguen* en 1988 *[ Goguen 88 ]*. Es un lenguaje que ha sido objeto de estudio desde 1976 con los primeros diseños de *Joshep Goguen*, como un lenguaje para intentar extender la teoría de tipos de datos abstractos y manejar funciones parciales. Desde entonces a la fecha, han surgido varias versiones del lenguaje, la más reciente es *OBJ3*.

La semántica denotativa de *OBJ3* está basada en la lógica ecuacional de primer orden o álgebra de tipos ordenados. Esta álgebra permite definir subtipos que dan la facilidad de manejar herencia múltiple, excepciones y definir funciones sobrecargadas, permitiendo así establecer funciones totales, que de otra manera serían parciales y restringidas a un solo tipo de dato.

La semántica operacional del lenguaje, está basada en ecuaciones que son escritas en forma declarativa mediante una signatura y que son interpretadas como reglas de reescritura, de manera similar a *ML*. A diferencia de *ML*, *OBJ3* permite definir estas ecuaciones con propiedades asociativas, conmutativas y con un elemento idéntico asociado. Por otro lado, da al usuario la libertad de elegir cualquier estrategia de evaluación respecto a

los argumentos de las ecuaciones, como evaluación retardada o anticipada y da también la facilidad de memorizar en una base de datos, los resultados de evaluaciones de ecuaciones hechas ya procesadas.

Desde su diseño inicial, *OBJ3* fue orientado especialmente hacia el manejo de módulos parametrizados de una manera fácil y eficiente. Para realizar y soportar eficientemente la programación parametrizada, introduce tres tipos de unidades de programación, llamados *objetos*, *teorías* y *vistas*. Cada uno agrupa a un conjunto de ecuaciones para un propósito específico.

La unidad *objeto* es usado para definir un tipo de dato abstracto, una relación entre tipos, subtipos o simplemente un conjunto de ecuaciones con un significado particular. Esta unidad puede estar parametrizada por otras unidades de este tipo a su vez. La sintaxis de un *objeto* es la siguiente:

```
obj Nombre del objeto[ Parámetros ] is tipo

    subsort tipo tipo tipo .....

    Definición de funciones:

       op nombre de función : tipos -> tipo .

    vars

      Declaración de variables:

        Nombre de variable : tipo .

      Declaración de ecuaciones:

         eq Nombre de función( argumentos ) =  valor .

endo .
```

En la misma unidad objeto se declara primero la sintaxis de las ecuaciones, como funciones y en seguida su semántica. La primera parte constituye la parte declarativa y la segunda la de ejecución, de acuerdo a la nomenclatura que se ha estado utilizando. A diferencia de los otros lenguajes mencionados, ambas partes están agrupadas en un mismo bloque.

La unidad llamada *teoría*, sirve para especificar y definir propiedades semánticas y sintácticas de las unidades requeridas como parámetros para una instanciación significativa, es decir, establece los requisitos que deben satisfacer los parámetros de las unidades *objeto*. Esta unidad puede también estar parametrizazada a su vez por otras unidades de *teorías* o de *objetos*. La sintaxis de una *teoría* es la siguiente:

```
th Nombre de la teoría is tipo

   sort tipo

   Definición de funciones:

   op nombre de función : tipos -> tipo .

   vars

     Declaración de variables:

       Nombre de variable : tipo .

     Declaración de ecuaciones:

       eq Nombre de función( argumentos ) = valor .

endth .
```

La unidad llamada *vistas*, asocia a una unidad que funge como parámetro de un *objeto*, una *teoría* dada que es el parámetro de un *objeto* parametrizado. Es decir, para instanciar un parámetro formal con un parámetro actual, es necesario proporcionar una *vista* para cada parámetro. La *vista* expresa que un cierto *objeto* satisface una cierta *teoría* en una cierta manera, ya que un *objeto* puede satisfacerla de más de una manera. La sintaxis de una *vista* es la siguiente:

```
view Nombre de la vista from Nombre de una teoría to
     Nombre de un objeto is

    sort tipo to tipo

    vars

    Declaración de variables :

       Nombre de variable : tipo .

    op Nombre de función to Nombre de función .

endw .
```

La instanciación de un *objeto* se realiza haciendo un mapeo de los tipos y funciones de la *teoría* dada, a los tipos y funciones del *objeto* que va a recibir como parámetro, preservando la relación de subtipos, aridad, atributos y tipos en las funciones.

Para la definición de las ecuaciones en las tres unidades, *OBJ3* ofrece gran flexibilidad sintáctica pudiéndolas describir en forma infija, prefija o posfija. Además, el tipo de los argumentos, así como el tipo que regresan las funciones y sus atributos o propiedades, se define al mismo tiempo que su sintaxis.
A diferencia de *ML*, las variables involucradas en la definición de ecuaciones en una unidad, se deben definir antes de usarlas, junto con el tipo al que pertenecen.

A continuación se escribirá en el lenguaje *OBJ3*, el ejemplo de colas descrito antes en *ML*. El siguiente *objeto* define el tipo cola, el cual está parametrizado por la *teoría* llamada *Elem*, que debe satisfacer la *teoría TipoElem*.

```
obj COLA[ Elem :: TipoElem ] is Cola
    subsort Elt < Cola .
    op cola_vacia : -> Cola .
    op mete_ult : Elt Cola -> Cola .
    op resto : Cola -> Cola .
    op primero : Cola -> Elt .
    op es_vacia : Cola -> Bool .

    Vars
       e : Elt .
       C : Cola .

    eq resto( mete_ult( e, cola_vacia ) ) = cola_vacia .
    eq resto( mete_ult( e, c ) ) = mete_ult( e, resto( c ) ).
    eq primero( mete_ult( e, c ) ) = primero( c ) .
    eq primero( mete_ult( e, cola_vacia ) ) = e .
    eq es_vacia cola_vacia = true .
    eq es_vacia mete_ult( e, c ) = false .

    endo
```

La *teoría TipoElem* simplemente especifica que el parámetro, que es un *objeto*, debe tener un tipo definido:

```
th TipoElem
   sort Elt .
endth
```

Las siguientes dos *vistas*, definen instancias de elementos que puede manejar una cola, la primera de naturales (*NAT*) y la segunda de booleanos (*BOOL*):

```
view Naturales from TipoElem to NAT is
     sort Elt to NAT
endv

view Booleanos from TipoElem to BOOLEAN is
     sort Elt to BOOL
endv
```

Ambos *objetos*, *NAT* y *BOOLEAN* están incorporados al lenguaje y definen, respectivamente, al tipo *NAT* y *BOOL*.

Para instanciar un *objeto* parametrizado, a través de una *vista*, se utiliza la instrucción *make*, como se muestra enseguida:

```
make Cola_Naturales is COLA[ Naturales ] endm .
make Cola_Booleanos is COLA[ Booleanos ] endm .
```

Otra de las características que posee *OBJ3* para el manejo de módulos, son las expresiones modulares o composición de módulos. Una expresión modular es:

- Un módulo no parametrizado, o
- Un módulo parametrizado cuyo parámetro es una expresión modular.

En *OBJ3*, las expresiones modulares se realizan mediante las *teorías*.

En *ML*, una signatura puede tener en un programa varias estructuras que la satisfagan. Cada uno de ellos especifica distintas formas de ejecutar lo declarado en la signatura, y se puede elegir a conveniencia la que se quiera. En *OBJ3* esto no es posible, porque la parte de declaración de ecuaciones (parte ejecutable de un *objeto*), debe venir inmediatamente a continuación de la declaración de las funciones y dentro de la misma unidad *objeto*.

*OBJ3* a diferencia de *ADA*, aparte de introducir el concepto

de módulos parametrizados, introduce el mecanismo de instanciación manual de parámetros de un módulo. Esto se realiza en *OBJ3* a través de la unidad llamada *vistas*.

También *OBJ3*, a diferencia de *ML* y *ADA*, permite la construcción de expresiones modulares.

Una aportación importante de estos dos últimos lenguajes, *ML* y *OBJ3*, al concepto de módulo, es que éste puede verse también como un tipo. Ya que un módulo parametrizado, requiere que se especifiquen las características o requerimientos que debe tener un parámetro, como si fuera un valor con un tipo asociados. En el lenguaje *ML*, las signaturas especifican estos requerimientos, mientras que en *OBJ3* son las *teorías* las encargadas de definirlos.

En el apéndice *A* aparece una tabla comparativa de algunos conceptos de módulos en los lenguajes *ADA*, *ML* y *OBJ3*.

A continuación se verá como se lleva a cabo la descripción de un tipo en un módulo.

## I.7 La Descripción de Tipos en Módulos.

La descripción de un tipo, según si se especifica en la parte de ejecución o declarativa de un módulo, puede ser protegida o pública, respectivamente. Puede también definirse en ambas partes del módulo. Si su declaración es protegida, el tipo puede ser usado en los módulos que lo importan o que lo reciben como parámetro, pero ninguna información acerca de su representación es dada a conocer fuera del módulo donde se describe. Esta forma de declaración se conoce también como opaca, mientras que la pública como transparente.

En general, se han distinguido tres modos para representar tipos: el hipotético, el abstracto y el transparente. Los dos primeros requieren que la descripción del tipo sea visible sólo dentro de la parte de ejecución de un módulo. El modo hipotético impone además la restricción de que si un tipo hace uso de otro tipo que está definido bajo este modo, él también deberá estar definido de la misma forma.

El modo transparente, permite que la representación del tipo sea visible fuera de la parte de ejecución de un módulo. La parte declarativa correspondiente, sólo permite dar a conocer al exterior del módulo, alguno de los identificadores involucrados en la representación . Este modo no tiene los problemas que tienen los otros dos mencionados, el modo hipotético y abstracto, sin embargo introduce otro: permite que un usuario al conocer la representación de un tipo, pueda alterarla, afectando así a otros módulos.

*ADA* permite la definición transparente del tipo, si se declara en la especificación de un paquete; es abstracta, si se declara en el cuerpo de un paquete, en donde recibe el nombre de tipo privado.

*ML* permite la representación del tipo de manera transparente y abstracta, dependiendo del contexto donde se defina y se usen los tipos.

Por ejemplo, en la definición de la siguiente estructura y funtor, el identificador de tipo *B.s* hace referencia de manera transparente al tipo *X.t* list, mientras que el tipo *X.t* es abstracto:

```
structure A : sigA =
   struct
      type t = int * int
   end ;
```

```
functor G( X : sigA ) : sigG =
   struct
      structure B : sigB =
      struct
         type s = X.t list
                  . . .
      end
        . . . B.s . . .
   end ;
```

Después de instanciar el funtor llamado *G*, la referencia *C.B.s* es transparente y denota *A.t* list, que es el tipo definido como ( int * int ) list:

```
structure C : sigG = G( A )
```

En general, en *ML* las referencias a través de una estructura son transparentes, mientras que las que se hacen a través de un parámetro de un funtor son abstractos.

El lenguaje *OBJ3* no posee mecanismos para definir tipos abstractos.

Como un intento por solucionar las desventajas que trae consigo cada una de las representaciones posibles de un tipo en un módulo, en *[ Drossopoulou 87 a y b ]* se sugiere una combinación de los distintos modos presentados, además de considerar las partes de un módulo como elementos parametrizables, como declaraciones parametrizadas por otras declaraciones, como se verá mas adelante.

## I.8 Problemas Abiertos en la Organización y Definición de Módulos .

Si se desea diseñar un sistema de módulos en el que se conciba a un módulo compuesto de dos partes, una parte declarativa y otra de ejecución, junto con algunas de sus características como la parametrización de entidades, así como alguna forma de expresar los tipos, el sistema aún no es muy poderoso. Ya que siguen existiendo todavía problemas de repetición de código; aún no es posible reutilizar código en algunos casos y aunque la abstracción ha ocupado un lugar, sigue habiendo situaciones en las que aún es ineficiente esta organización de un módulo.

Algunos de los problemas que hay que tomar en consideración para contar con un sistema de módulos útil, son los siguientes:

a) Múltiples partes de ejecución para una misma parte declarativa de un módulo .

Si se desea ordenar dos conjuntos de números, uno de los cuales casi está ordenado y el otro no, no sería eficiente tener dos módulos para ordenarlos. La diferencia entre cada uno de estos módulos, es que uno define un algoritmo eficiente para números casi ordenados y el otro para números completamente desordenados.

La solución propuesta aquí es que sea posible tener varias implantaciones (partes de ejecución de un módulo) que satisfagan a la vez una misma declaración, y no una para cada implantación y que además sea la misma.

b) Parametrización de partes de ejecución de un módulo.

¿ Por qué no considerar una generalización de la parametrización de entidades, es decir permitir la parametrización de partes de ejecución también ?. De esta forma se puede definir una o varias entidades en una parte ejecutable, las cuales pueden a su vez definirse de distinta manera, de acuerdo a la parte ejecutable que reciba como parámetro.

c) Subdeclaraciones.

Supóngase que se escribe en la parte declarativa de un módulo qué es una bicola. En el mismo programa se desea ahora también manejar, mediante un módulo, una cola de elementos. Sería provechoso utilizar de alguna manera la declaración del tipo bicola, sabiendo que una cola es un caso particular de ella. Además, la parte ejecutable del módulo que define el tipo bicola podría servir también para el tipo cola.

Esto sugiere considerar en un sistema de módulos la definición de subdeclaraciones, permitiendo de esta manera que la parte ejecutable de un módulo también pueda tener varias declaraciones.

d) Parametrización de declaraciones.

Supóngase que en un programa se define un módulo llamado *árbol binario*, el cual define qué es un árbol binario y sus operaciones conocidas. Más tarde, en el mismo programa, se desea definir en otro módulo, el algoritmo de ordenación *heapsort*. Este algoritmo para su funcionamiento, requiere de un árbol binario y de algunas de sus operaciones. Sería ineficiente volver a escribir en el módulo que define el algoritmo de *heapsort*, el contenido del módulo *árbol_binario*.

Se sugiere en este caso, como solución, considerar declaraciones también parametrizadas que requieran, para su definición completa, de otras declaraciones que son proporcionadas como parámetros.

A continuación se describen diversas soluciones que se han sugerido y desarrollado, para resolver los casos antes planteados.

a) Soluciones propuestas para múltiples partes de ejecución para una misma parte declarativa de un módulo.

Una solución consiste en permitir múltiples partes de ejcución que satisfagan una misma declaración. Así, la parte declarativa de un módulo, describe entonces una familia de implantaciones (partes de ejecución). Por ejemplo, el lenguaje ***NL*** permite que varias partes de ejecución sean definidas para una misma de-

claración, ya que una estructura puede satisfacer de varias maneras una signatura.

b) Soluciones propuestas a la parametrización de partes ejecutables de un módulo.

Aquí, puede considerarse que la parte ejecutable de un módulo, pueda estar parametrizada por una o varias partes ejecutables de otros módulos. El parámetro requerido se puede especificar en la parte declarativa o de ejecución. Si se especifica en la parte declarativa, debe tenerse cuidado de entenderse que no es un parámetro indispensable para ella, sino para la correspondiente parte ejecutable del módulo. La especificación del parámetro, independientemente de donde se haga, se hace indicando que se trata de la parte ejecutable de un módulo, y no de otro tipo de entidad. Se debe indicar también el nombre de la parte declarativa a la cual pertenece.

El lenguje *ML* permite realizar la parametrización de partes ejecutables de módulos, a través de los funtores. También el lenguaje OBJ3 permite realizarlas, a través de la unidad *teoría*.

c) Soluciones Propuestas para el manejo de Subdeclaraciones.

Aquí hay dos esquemas a considerar. Por una parte, plantea que la parte ejecutable de un módulo, pueda tener varias partes declarativas, y por otra, la existencia de subdeclaraciones. Con el primer esquema y con lo que se vio en la solución al inciso a), debe ahora considerarse que no es necesario que exista una correspondencia uno a uno entre la parte declarativa y de ejecución de un módulo.

En el caso de que la parte ejecutable de un módulo tenga varias partes declarativas, cada una de éstas difiere de las otras, en grado de generalidad y de restricción.
Este esquema es importante para expresar un tipo de dato abstracto, en varios niveles de abstracción y permitir un acceso controlado a dicho tipo.

Además, permite ver al tipo y manejarlo de distintas maneras, de acuerdo a como esté definido en cada parte declarativa. Por ejemplo, en el diseño de compiladores el módulo que corresponde a la fase de análisis, necesita conocer como representar un árbol,

crear e instanciar nodos dentro de él. Mientras que al módulo correspondiente a la fase de análisis sintáctico, sólo le interesaría saber cómo recorrer el árbol.

Por otra parte, en el esquema de existencia de subdeclaraciones, se dice que la parte declarativa *A* de un módulo, es una subdeclaración de otra parte declarativa *B*, de otro módulo, si para cada descripción hecha en *B* existe la misma en *A*. Tal descripción debe tener el mismo nombre o tipo, salvo por renombre. Se considera que la parte declarativa *A* exporta (posiblemente) más elementos que *B* y existen (potencialmente) más partes de ejecución de *B* que de *A*.

Por otra parte, cualquier parte ejecutable de un módulo que satisfaga *A*, satisface también *B*. La relación de subdeclaraciones es reflexiva y transitiva.

De los lenguajes presentados anteriormente, solo *ML* posee mecanismos para permitir que la parte ejecutable de un módulo se corresponda con varias partes declarativas de otros módulos. Pero ninguno permite expresar la relación de subdeclaraciones.

d) Soluciones propuestas para la Parametrización de declaraciones.

La parametrización de partes declarativas de un módulo, sirve básicamente para formar otras partes declarativas de otros módulos. Puede considerárseles como funciones que reciben como argumento una declaración de entidades y regresan (representan) otra declaración.

Los parámetros son partes declarativas o subdeclaraciones de módulos.

La ventaja de definir la parametrización de partes declarativas de un módulo, es que pueden verse extendidas en sus declaraciones por el parámetro que reciben. Estos parámetros pueden también agregar algunas limitaciones acerca del uso de alguna enentidad definida.

Tampoco ninguno de los lenguajes antes presentados posee el concepto de partes declarativas parametrizadas.

En general, las partes declarativas y de ejecución parametrizadas pueden combinarse para definir relaciones más complejas entre módulos, y con esto lograr un sistema de módulos mas poderoso.

En las soluciones que se han propuesto para cada uno de los casos anteriores, existen algunas características adicionales relacionadas con el lenguaje en el que van a ser implantadas y entre ellas mismas, las cuales se mencionan a continuación.

Los esquemas de múltiples partes de ejecución para un misma declaración, y el empleo de subdeclaraciones, tienen la ventaja de poder ser incorporados a casi cualquier lenguaje que no los posea, sin requerir de algun concepto adicional y sin alterar su filosofía, conceptos y estructuras básicas.

El esquema de partes ejecutables de un módulo parametrizadas por ellas mismas, requiere algunos mecanismos y conceptos previos para caracterizar y describir sus parámetros, como es el que deben de satisfacer la parte declarativa de un módulo dado. Estos mecanismos y conceptos son sólo proporcionados si se toma en cuenta también el esquema propuesto de múltiples partes de ejecución para la parte declarativa de un módulo.

# C A P I T U L O  S E G U N D O

## Extensiones Modulares para el Lenguaje *PROLOG*.

El objetivo de este capítulo es presentar las alternativas de diseño de sistemas de módulos que se han trabajado para el lenguaje *PROLOG*. Se discutirá el objetivo de introducir un sistema de módulos en *PROLOG*; los mecanismos básicos que posee *PROLOG* para la estructuración de programas; las distintas propuestas que se han realizado para extender el lenguaje *PROLOG*, hacia la programación estructurada y finalmente, se darán algunas recomendaciones y características básicas, que se han planteado algunos investigadores y las cuales deberá tener un sistema de módulos para *PROLOG*.

### 2.1 Objetivo de Introducir un Sistema de Módulos en *PROLOG*.

La finalidad de introducir módulos en *PROLOG* se resume en los siguientes puntos:

- Obtener las facilidades de la programación estructurada.

- Regular la visibilidad y la existencia de los elementos internos de un bloque.

- Obtener beneficios particulares debidos a la propia naturaleza del lenguaje y que no se lograrían en otro tipo de lenguajes con un sistema de módulos similar . Por ejemplo, la representación de conocimientos a través de módulos como marcos (*frames*), objetos, jerarquías, etc..

## 2.2 Mecanismos Básicos de Estructuración y Modularidad en *PROLOG*.

Los mecanismos de estructuración y modularidad que posee *PROLOG*, son muy elementales. Estos están dados básicamente por la separación entre conocimiento y control, mediante la distinción entre hechos y reglas. Como señalan algunos autores *[ Mello 89 ]*, el grado de modularidad de este medio es demasiado "fino". Otro de los mecanismos de estructuración que proporciona *PROLOG*, es el manejo de distintas bases de datos donde se halla declarado un conjunto de cláusulas. El inconveniente es que en todo momento sólo se permite manejar una base de datos. Ahora el grado de modularidad, como señalan los mismos autores, parece ser demasiado grande. Sin embargo, no existe en *PROLOG* un mecanismo para agrupar un conjunto de cláusulas y predicados.

Existen algunas implantaciones de *PROLOG [ Arity 91 ]*, que dan un mecanismo de estructuración, al permitir dividir una base de datos en bloques, llamados mundos. Un mundo puede contener hechos y reglas. En un momento dado solo puede existir un mundo activo, es decir, un conjunto de hechos y reglas a través del cual una meta dada es resuelta dentro de él, sin tener en cuenta la posibilidad de la existencia de otros mundos, en donde puede haber otras definiciones alternativas para resolver la meta.

Cada mundo, en estas versiones, recibe un nombre y existe además un predicado para crearlo. También existe un predicado para saber el nombre del mundo actual y cambiar a otro mundo existente, para considerar otras definiciones alternativas de predicados u otros predicados distintos. Los predicados que alteran la base de datos como *asserta*, *assertaz*, *record*, *recordz*, etc. afectan sólo al conjunto de predicados contenidos en el mundo activo.

El concepto de mundo es quizás, el mecanismo más cercano al concepto de módulo que posee *PROLOG*, aunque tiene algunos inconvenientes. Uno de ellos es que no proporciona un mecanismo claro para agrupar físicamente el conjunto de predicados que pertenecen a un mundo dado. Ya que en una parte de un programa se puede definir un predicado que cree un mundo, en otra parte activar ese mundo y definir unos cuantos predicados que le pertenecen y más adelante del programa otros. Además, un mundo no cuenta con mecanismos de exportación ni de importación, todo permanece oculto en el mundo, sin tener la posibilidad de tomar en cuenta otros mundos existentes. Otra desventaja es que no se permiten mundos parametrizados.

En cuanto a los mecanismos que permiten controlar la visibilidad y existencia de predicados y cláusulas, *PROLOG* cuenta con un conjunto de predicados extralógicos como *assert*, *assertz*, *re-*

*cord*, *recordz*, *retract*, etc. que alteran el contenido de una base de datos, insertando, eliminando o modificando predicados y cláusulas, así como alterando el orden en que ellos aparecen en la base de datos. Sin embargo, estos mecanismos no proporcionan los medios precisos para regular la existencia y la visibilidad de predicados y cláusulas como se desearía. Sólo controlan la visibilidad de predicados y cláusulas. Por ejemplo, el predicado *retract*, elimina totalmente una cláusula de la base de datos, mientras que el predicado *assert* agrega una cláusula, haciéndola totalmente ahora visible a toda la base de datos. Es deseable tener un mecanismo que permita ocultar una cláusula pero solo parcialmente, es decir, que solo uno o unos cuantos módulos supieran de su existencia y visibilidad en un momento dado.

En los últimos 10 años se han realizado varias investigaciones y esfuerzos por extender y enriquecer el lenguaje *PROLOG*, tratando de incorporar en él conceptos de modularidad para escribir programas más estructurados. En seguida se describen varias de las investigaciones que se han realizado.

## 2.3 Estado del Arte de Módulos en PROLOG.

Existen muchas investigaciones que se han realizado para incorporar módulos en *PROLOG*, incluso para agregarle otros conceptos. Muchas de ellas varían en puntos de vista y formalidad, esto sugiere que no ha habido un marco o estándar a considerar como punto de referencia de lo que debe ser un sistema de módulos. Lo que puede ayudar a explicar esto, como lo señala *Michel Dorochewsky [ Dorochewsky 91 ]*, es el hecho de que *PROLOG* por su propia naturaleza, de ser un lenguaje interpretado y poseer características de metaprogramación, agrege cierto grado de complejidad al momento de diseñar un sistema de módulos y que no se tiene en cambio con otro tipo de lenguajes que no poseen esa característica, como los imperativos.

Una de las investigaciones que se han realizado para incorporar módulos a *PROLOG*, es la Programación Lógica Contextual, la cual sigue como mecanismo de estructuración, el razonamiento contextual. Existen otras que introducen un sistema de módulos como mecanismo explícito de estructuración, es decir sin seguir una filosofía en particular. Ejemplos de estos últimos son:

- El sistema de módulos de *ML* en *PROLOG*.

- El sistema de módulos de algunas implantaciones comerciales de *PROLOG*, como son:

  - *BIM-Prolog*.

  - *QUINTUS-Prolog*.

  - *Sepia*.

*Michael Dorochewsky* sugiere otra clasificación que está más relacionada con los aspectos sintácticos y semánticos de un módulo. Para él, los sistemas de módulos existentes con los conceptos asociados, pueden clasificarse en:

- Modularidad Sintáctica
- Modularidad Semántica

### 2.3.1 Sistemas de Módulos Sintácticos.

En el esquema de Modularidad Sintáctica la estructura modular existe sólo a un nivel sintáctico. Una característica de este esquema, es que después de hecha la compilación y ensamblado de de un programa escrito con un sistema de módulos de este tipo, no queda reflejada, en el programa objeto resultante, ninguna estructura modular del programa fuente. El alcance de procedimientos, funciones y variables es determinado en tiempo de compilación.

Algunos lenguajes que tienen un sistema de módulos de este tipo son *ADA* y *Modula-2 [ Wirth 83 ]*. Aunque se han desarrollado algunos sistemas de módulos para *PROLOG* con este esquema, como son *DEC-10 PROLOG [ Bowen 81 ]* y *Siemens-PROLOG [ AG 88 ]*.

### 2.3.2 Sistemas de Módulos Semánticos.

En el esquema de modularidad semántica, la estructura modular del programa fuente queda reflejada fielmente en el programa objeto. Por lo que es más eficiente que el esquema anterior, pero es también más complejo y agrega cierto sobrepeso a los programas ejecutables.

*LISP [ Steele 84 ]* es un lenguaje funcional que posee un sistema de módulos similar al de este esquema semántico.

El esquema de modularidad sintáctica presenta algunas desventajas sobre el de modularidad semántica. Por ejemplo, al definir predicados privados en un módulo, ellos no pueden ser llamados a un nivel de metaprogramación aún desde dentro de ese mismo módulo. Incluso, predicados como *assert* o *clause* no pueden ser usados dentro de un módulo. Estas desventajas son debidas exclusivamente al sentido sintáctico que se les da a los módulos.

El esquema de modularidad semántica se divide a su vez en Modularidad Basada en Nombres y en Modularidad Basada en Predicados. A continuación se describe cada uno de ellos.

La característica básica de los sistemas de módulos Basados en Nombre, es que el concepto de módulo se aplica sólo a átomos o funciones. Cada uno de estos posee un identificador el cual representa el nombre del módulo al que pertenecen. Una ventaja importante que posee este tipo de sistema, es que permite la definición de tipos de datos abstractos, ya que permite definir funtores privados que oculten una estructura de datos.

Este esquema de módulos posee algunos inconvenientes, que lo hacen inapropiado, ya que presenta algunos problemas de flexibilidad para la metaprogramación y encapsulación. Por ejemplo, si un nombre de una función, predicado o átomo, es declarado como global, entonces las funciones, predicados y átomos con ese mismo nombre, declarados en un módulo como locales, son accesibles fuera de él; lo cual no siempre es deseable. También con este esquema, el algoritmo de unificación se comporta un poco diferente al de *PROLOG* normal. Por ejemplo, dos átomos con el mismo nombre, declarados como locales en dos módulos distintos, no pueden unificarse. Esto, sin embargo, no parece ser demasiado restrictivo.

Uno de los lenguajes que se ha desarrollado teniendo en cuenta el esquema basado en nombre es *BIM-Prolog [ BIM 89 ]*.

A diferencia del anterior, en los sistemas de módulos Basados en Predicados la modularidad se aplica sólo a predicados, los átomos, funciones y términos no son elementos que puedan pertenecer a un módulo. En este esquema existe un mecanismo de importación y exportación de predicados. La desventaja que posee este sistema, es que no da las mismas facilidades que el otro esquema para implantar tipos de datos abstractos. Además, las implantaciociones comerciales que se han realizado con este sistema, como es *QUINTUS-Prolog [ Quintus 87 ]*, no satisfacen los requerimientos de encapsulación y flexibilidad de metaprogramación al mismo

tiempo.

A continuación se describe el concepto de módulo en la Programación Lógica Contextual y el sistema de módulos de *ML* en *PROLOG*.

### 2.3.3 Programación Lógica Contextual.

La Programación Lógica Contextual surge como una propuesta por extender la programación lógica hacia el razonamiento contextual, sugerida por *Luis Monteiro* y *Antonio Porto [ Monteiro 89 ]* en la Universidad de Nova de Lisboa, Portugal.

El propósito de este paradigma es, por una parte, proporcionar módulos de programas, llamados unidades, que ofrezcan las facilidades de un sistema de módulos tradicional. Por otra parte, pretende dar un modelo computacional de razonamiento contextual, el cual es comúnmente necesitado en algunas áreas de la inteligencia artificial, como el procesamiento de lenguaje natural, los sistemas expertos, el aprendizaje, el diseño de sistemas para prototipos, etc..

El concepto de contexto puede ser pensado como una asociación de ambientes, es decir, asignar un nombre a un valor durante un período en tiempo de ejecución de un programa. Este concepto es común y existe también en otros lenguajes de programación. En este caso, el contexto es un medio para asociar llamadas de predicados con definiciones.

Las unidades consisten de agrupaciones de un conjunto de cláusulas. Cada unidad recibe un nombre. Estas cláusulas pueden aparecer también en otras unidades, proporcionando así distintos medios ambientes de evaluación de predicados. Para proporcionar un mecanismo de visibilidad y ocultamiento de hechos y reglas, se proporciona un predicado llamado *visible( x/n )*, el cual indica que el predicado *x* con aridad *n*, definido dentro de una unidad, es dado a conocer al exterior. La siguiente unidad, llamada *operacioneslistas*, realiza algunas operaciones sobre listas :

```
unit( operacioneslistas ) .

      visible( long/2 ) .
      visible( pertenece/2 ) .
      visible( invierte/2 ) .
```

```
long( [], 0 ) .
long( [ X | Xs ], N ) :- long( Xs, M ), N is M + 1 .

pertenece( X, [ X | Xs ] ) .
pertenece( X, [ Y | Ys ] ) :- pertenece( X , Ys ) .

invierte( [], [] ) .
invierte( [ X | Xs ], Ys ) :- invierte( Xs, Zs ),
                              une( Zs, [ X ], Ys ) .

cabeza( [ X | Xs ], X ) .
cabeza( [], [] ) .
```

El predicado *une* no está definido en esta unidad, por lo que aún no puede usarse. La siguiente unidad, llamada *unelista* define este predicado:

```
unit( unelista ) .

    visible( une/2 ) .

    une( [ X | Xs ], Ys, [ X, Zs] ) :- une( Xs, Ys, Zs ) .
    une( [], Xs, Xs ) .
    une( Xs, [], Xs ) .
```

Para utilizar el predicado *invierte* es necesario proporcionar las dos unidades: *operacioneslistas* y *unelista*.
Es importante notar la presencia del predicado *visible( une/2 )* en la unidad *unelista*, ya que sin él, no sería posible usar el predicado *invierte* aunque estén las dos unidades presentes .

Dada una meta y un conjunto de unidades, el programador puede elegir cualquier combinación de unidades para evaluar dicha meta . Por lo tanto, la ejecución de un programa consiste en la evaluación de metas en una colección ordenada de unidades, llamada contexto . De esta forma un contexto representa lo que puede llamarse una línea de razonamiento . Un contexto es representado por una lista de nombres de unidades, de la forma *[ un,...,u0 ]*, con *un* el nombre de la última unidad agregada al contexto y *u0* la primera unidad agregada. Los contextos pueden ser dinámicamente extendidos, usando el operador llamado de extención >>. Este operador utilizado en una expresión como la siguiente:

```
u >> m
```

significa que la meta *m* debe ser probada en el nuevo contexto formado por la unidad *u*, y el contexto existente. Por ejemplo, considerando las unidades dadas anteriormente, para evaluar la meta *invierte( [ 1, 2, 3 ], Xs )*, se utiliza la siguiente expresión:

*unelista >> operacioneslistas >> invierte( [ 1, 2, 3 ], Xs )*

De esta forma, la meta *invierte( [ 1, 2, 3 ], Xs )* será resuelta en el contexto *[ operacioneslistas, unelista ]*.

La programación lógica contextual tiene la característica de permitir cambiar dinámicamente de ambientes, cuando una nueva información del propio sistema es generada. Proporciona también distintos puntos de vista de un programa de acuerdo a la presencia o ausencia de datos.

Con los conceptos básicos de la programación contextual presentados hasta aquí, es natural que ahora surjan algunas preguntas, como las siguientes: Dado un contexto ¿ Cuál es la definición de un predicado cuando es llamado ?, ¿ Qué parte del contexto tomar en cuenta cuando se propone una meta a resolver en dicho contexto ? y ¿ Cómo determinar el contexto actual cuando una extención tiene lugar ?.

Existen varias respuestas para estas preguntas que conforman diferentes políticas de diseño del lenguaje en cuanto a:

Definición de Predicados,

Tipos de asociación y

Alcance de Predicados.

Respecto a la definición de predicados, como existe la posibilidad de que un predicado pueda ser definido en varias unidades, en el mismo contexto, existen dos alternativas a considerar. Una de ellas se comporta de la siguiente manera. Cuando una extensión de contexto tiene lugar, la unidad agregada a ella extiende la definición de algunos de los predicados existentes en el contexto previo. Otra alternativa es que esta unidad invalide las definiciones de predicados contenidos en el contexto previo e introduzca nuevas definiciones de predicados.

Cualquiera de estas dos alternativas o ambas, puede ser utilizada en la implantación. Si se elige ambas alternativas, una de ellas puede adoptarse como definición por omisión. Por ejemplo, si se pone al principio de una unidad el predicado *extends( p / n )*, indica que el predicado *p* puede tener otras definiciones, cuando una extención de contexto tiene lugar. Pero si este predicado no es colocado, indica que su definición puede ser invalidada por otras unidades, cuando también una extención de contexto tiene lugar.

En cuanto a la asociación, existen también dos alternativas. Supóngase que se tiene el contexto *U = [ un, un-1,..., ui,...,u1, u0 ]* y una meta *m*, que al ser probada en dicho contexto falla en la unidad *ui*. Para evaluar esta meta, una alternativa es buscar definiciones de los predicados involucrados sólo en las unidades *ui-1,...,u1,u0* y si la meta aún falla, entonces la meta falla en el contexto *U*. Otra alternativa es buscar definiciones en todo el contexto a partir de la unidad *un*.

La primera solución es llamada evaluación de predicado en forma anticipada *(eager)* y la segunda retardada *(lazy)*. Se asume por convención que cada predicado de una unidad al ser llamado, debe ser evaluado en forma anticipada, pero si es precedido por el operador *#*, indica que tiene que ser evaluada en forma retardada. Por ejemplo, supóngase que se tienen las siguientes unidades:

```
unit( u1 ) .          unit( u2 ) .                    unit( u3 ) .
  a( 1 ) .              c( X, Y ) :- a( X ), #b( y ) .    a( 3 ) .
  b( 1 ) .                                               b( 3 ) .
```

y se desea ver el resultado de la expresión:

*u1 >> u2 >> u3 >> p( X, Y ) .*

Es decir, el resultado de la meta *p( X, Y )* en el contexto *[ u3, u2, u1 ]*. El predicado *a/1* es evaluado bajo el contexto *[ u2, u1]* y el predicado *b/1*, por tener asociado una forma de evaluación retardada, en *[ u3, u2, u1 ]*. El resultado de la evaluación es entonces *X = 1* y *Y = 3*.

Aunque ambos tipos de llamadas dan flexibilidad a los mecanismos de composición de unidades, la asociación de tipo anticipada no es recomendada y es ineficiente para desarrollar unidades que deben ser creadas dinámicamente y que deben ser tomadas en cuenta para la definición de un predicado. Ya que sólo toma en

cuenta los predicados de las unidades *[ ui,...u1 ]* y no las otras unidades *un, ... , ui-1,* que podría tener una implantación más eficiente de un concepto expresado a través de predicados.

En lo que al alcance de los predicados se refiere, veamos la relación que guarda una unidad u con un contexto actual *U,* cuando una extensión de contexto, *u >> m* tiene lugar. Esta relación o combinación de *u* con *U,* se conoce como el alcance de la unidad *u* respecto a *U.* Existen dos formas en que se puede definir el alcance de una unidad. Una puede ser de tipo léxico y otro dinámico.

En el alcance léxico, la relación de una unidad con el contexto es determinado cuando la unidad es creada. En el alcance dinámico, su relación es establecida cuando una extensión tiene lugar, es decir, cuando el operador >> es usado.

El alcance léxico tiene la propiedad de "congelar" el contexto actual, ya que la extensión de un contexto *U* con una unidad *u* definida con alcance léxico, forma un nuevo contexto *U'*, que es el contexto asociado con *u* en su tiempo de creación. Este nuevo contexto *U'* será usado por la unidad *u* para evaluar predicados sin tomar en cuanta el contexto previo *U*. Cuando haya finalizado la evaluación del predicado en la unidad *u* y contexto *U'*, el contexto actual vuelve a ser *U*.

Las unidades definidas con un alcance dinámico se pueden combinar de varias maneras para lograr diferentes contextos de prueba, obteniendo así diferentes puntos de vista. Por ejemplo, la meta *g(X)* puede ser resuelta en dos diferentes contextos en la siguiente manera:

*u >> v >> g( X )*
*r >> v >> g( X )*

con *u, v* y *r* unidades definidas con alcance dinámico . En el primer caso, la meta *g(x)* es probada en el contexto *[ v, u ]*, mientras que en el segundo en *[ v, r ]*. El comportamiento de *g( x )* en estas dos unidades puede ser distinto dependiendo de los predicados definidos en *u* y *r*.

Para definir el tipo de alcance de una unidad se usan los siguientes predicados:

*define( nombre de la unidad, localización de la unidad )*
*create( nombre de la unidad, localización de la unidad )*

Ambos predicados reciben como primer parámetro, el nombre de una unidad y como segundo el lugar donde localizar la unidad, por ejemplo, el dispositivo físico. El primer predicado especifica que la unidad definida tiene un alcance léxico, mientras que la segunda un alcance dinámico. La manera de indicar que la unidad U tiene asociado un alcance léxico con el contexto *[ un, .. ul ]*, es a través de la siguiente expresión:

*ul >> :.. >> un >> define( U, localización de la unidad )*

## Extensiones e Implantación.

Aunque la Programación Lógica Contextual sigue siendo objeto de investigacion, ya se han desarrollado algunas implantaciones y aplicaciones de él. Además, se han sugerido otras posibles extenciones del lenguaje.

Una de estas extensiones consiste en permitir definir unidades parametrizadas por otras unidades, haciendo de esta forma más explícita la dependencia de un predicado con el exterior. Una extensión más que se sugiere es la de unidades asumidas, que en ocasiones es útil para evitar escribir demasiado, permitiendo la existencia de relaciones predefinidas entre unidades. Esto significa que, por ejemplo, en lugar de escribir una expresión como *x >> y >> z >> M*, basta con solo escribir *z >> M*, asumiendo que la unidad *z* requiere la unidad *y*, y ésta la unidad *x*.

Una extensión más que se sugiere, es la de contar con un mecanismo o estructura como la de una pila, la cual contenga un historial de los contextos o unidades usadas recientemente para probar una fórmula. Esto serviría para recordar cual fue el contexto que se usó finalmente para demostrar una meta, ya que durante este proceso generalmente hay cambios de contexto o unidades.

Respecto a las aplicaciones, la Programación Contextual ha sido usado en el diseño de sistemas de prototipos. Uno de ellos es el llamado *ALPES-PROLOG*, el cual es un medio ambiente de desarrollo de prototipos basado en el uso de la programación lógica. Otra de sus aplicaciones, es un sistema de conocimiento con una interfaz en lenguaje natural.

Y en cuanto a la implantación, la cual se ha desarrollado sobre el mismo *PROLOG*, los autores sugieren que se desarrolle a nivel de la máquina abstracta de *PROLOG*, en lo cual no se ha investigado mucho.

### 2.3.4 El Sistema de Módulos de ML en *PROLOG*.

Como una alternativa para incorporar módulos a *PROLOG* de una manera particular, *D. T. Sanella* y *L. A. Wallen [ Sanella 92 ]* sugieren un sistema de módulos similar al de *ML*. El sistema de módulos que presentan, pretende ser fácil e independiente al lenguaje *PROLOG*, para lograr así una implantación también sencilla. El objetivo de su trabajo, es introducir el concepto de módulo en *PROLOG* pero desde un punto de vista funcional, muy similar al de *ML*.

El sistema de módulos que proponen, soporta la construcción de unidades de código parametrizadas y la definición de tipos de datos abstractos.

Los elementos que participan en el sistema de módulos del lenguaje son los mismos que los de *ML*: signaturas, estructuras y funtores. Para soportar el manejo de tipos de datos abstractos, los autores presentan una unidad de programación especial llamada abstracción (*abstraction*). El lenguaje *ML* cuenta también con este elemento en su sistema de módulos.

Signaturas.

Las signaturas describen el contenido del código ejecutable de un módulo, es decir, el quéhacer de las estructuras. El contenido de una signatura se compone de un conjunto de funciones y predicados junto con su aridad. La sintaxis de una signatura es la siguiente:

sig *Nombre de la signatura* =

fun *Declaración de funciones con aridad*

pred *Declaración de predicados con aridad*

end .

Las palabras sig y end agrupan los componentes de una signatura .

Por ejemplo, la siguiente signatura especifica el tipo de dato abstracto árbol binario y algunas de sus operaciones. El árbol binario es definido a través de las funciones hoja y árbol.

```
sig ArbolBin =

    fun hoja : 0, arbol : 3 .

    pred  hoja : 1, nodo : 1, unahoja : 1,
                  unnodo : 4, raiz : 2, arbizq : 2,
                  arbder : 2 .

end .
```

Estructuras.

Las estructuras definen un conjunto de funciones, predicados y cláusulas. Estas definen el cuerpo de los predicados definidos en una signatura, es decir, implantan lo que es requerido en una signatura. La sintaxis de una estructura es la siguiente:

```
structure Nombre de la estructura :  Nombre de una signatura

   struct

    fun Declaración de funciones con aridad

        Declaración de cláusulas

end .
```

Las palabras structure y end, agrupan los componentes de una estructura. El nombre de la signatura definida después de los dos puntos en una estructura, indica la signatura que implanta la estructura. Esta declaración es opcional.

La siguiente estructura implanta el árbol binario definido en la signatura ArbolBin definida anteriormente:

```
structure ImplArbolBin : ArbolBin =
   struct

      hoja( hoja ) .

      nodo( arbol( _, _, _ ) ) .

      unahoja( hoja ) .

      unnodo( Raiz, ArbIzq, ArbDer,
              arbol( Raiz, ArbIzq,  ArbDer ) ) .

      raiz( arbol( Raiz, ArbIzq, ArbDer ), Raiz ) .

      arbizq( arbol( _, ArbIzq, _ ), ArbIzq ) .

      arbder( arbol( _, _, Arbder ), Arbder ) .

   end .
```

Para permitir la construcción de programas con una estructura jerárquica, este sistema de módulos permite la declaración de estructuras dentro de una misma estructura. Este mecanismo sirve como un medio de importación y enriquecimiento de una estructura. Por ejemplo, la siguiente estructura define la cláusula pertenece, la cual verifica si un elemento pertenece o no a un árbol.

Esta estructura hace uso a su vez de la estructura ImplArbolBin:

```
structure PertArbolBin =
   struct

      structure X = ImplArbolBin .

      pertenece( Elem, X / arbol( Elem, _, _ ) ) .

      pertenece( Elem, X / arbol( _, ArbIzq, _ ) :-
          pertenece( Elem, ArbIzq ) .

      pertenece( Elem, X / arbol( _, _, ArbDer ) :-
          pertenece( Elem, ArbDer ) .

   end .
```

Esta estructura puede hacer referencia a todos los elementos definidos en la estructura *ImplArbolBin*. La manera de hacer referencia a cada uno de ellos, es a través del nombre de la estructura al que pertenecen seguido de una barra diagonal y el elemento deseado.

También en este sistema de módulos existe el operador open de *ML*. Este operador hace accesibles al exterior todos los elementos de una estructura.

Los mecanismos de ocultación de entidades son similares a los de *ML*. Esto es, los elementos que se especifican en una estructura, y que no aparecen definidos en la signatura correspondiente que satisface, son locales a esa estructura, y por consecuencia no es posible accesarlos fuera de ella.

Abstracciones.

Las abstracciones son un tipo especial de estructuras que permiten la definición de tipos de datos abstractos. Estas estructuras permiten ocultar la implantación de estos tipos, los cuales son definidos como predicados de *PROLOG*. La ventaja que tiene utilizar este tipo de estructuras, es que cualquier estructura puede hacer referencia a los tipos definidos en una abstracción, sin conocer su implantación. Esto tiene la ventaja de que al alterar la definición de un tipo de dato abstracto, no se modifican las estructuras que lo utilizan. Por ejemplo, si se modifica la definición del tipo árbol en la estructura *ImplArbolBin*, también será necesario hacer la misma modificación en la estructura *PertArbolBin*, ya que hace referencia directa a la implantación de este tipo. Para definir el tipo árbol en forma oculta, primero se define su signatura. Esta consiste en sólo definir las operaciones que lo caracterizan, pero no los predicados que lo definen, como son: *hoja : 0 y arbol : 3* en la signatura *ArbolBin*. Esta signatura es la siguiente:

```
sig ArbolBinOcult =

   pred  hoja : 1, nodo : 1, unahoja : 1,
         unnodo : 4, raiz : 2, arbizq : 2,
         arbder : 2 .

end .
```

La estructura (abstracción) que satisface esta signatura y define al tipo árbol es la siguiente:

```
abstraction ImplArbolBinOcult : ArbolBinOcult =
   struct
      fun hoja : 0, arbol : 3 .
      hoja( hoja ) .
      nodo( arbol( _, _, _ ) ) .
      unahoja( hoja ) .
      unnodo( Raiz, ArbIzq, ArbDer,
              arbol( Raiz, ArbIzq, ArbDer ) ) .
      raiz( arbol( Raiz, ArbIzq, ArbDer ), Raiz ) .
      arbizq( arbol( _, ArbIzq, _ ), ArbIzq ) .
      arbder( arbol( _, _, Arbder ), Arbder ) .
end .
```

La estructura *PertArbolBin* descrita antes, considerando ahora la implantación oculta del tipo árbol es la siguiente:

```
structure PertArbolBin =
   struct
      structure X = ImplArbolBinOcult .
      pertenece( Elem, Arbol ) :- X / raiz( Arbol, Elem ) .
      pertenece( Elem, Arbol ) :-
         X / arbizq( Arbol, ArbIzq ),
         pertenece( Elem, ArbIzq ) .
      pertenece( Elem, Arbol ) :-
         X / arbder( Arbol, ArbDer ),
         pertenece( Elem, ArbDer ) .
end .
```

Como se observa, esta estructura hace ahora referencia al tipo árbol en forma indirecta a través de los predicados *raiz*, *arbizq*, *arbder*, sin saber la representacón del árbol, ya que es desconocida.

Funtores.

Los funtores son estructuras parametrizadas por otras estructuras, similares a las de *ML*. Este tipo de estructuras permiten definir cláusulas que dependen de otras que son recibidas como parámetros. De esta forma se tiene la ventaja de elegir distintos medios ambientes de evaluación de una meta, de acuerdo al parámetro que reciba. Las estructuras que recibe como parámetro, deben satisfacer también las signaturas establecidas en el funtor. La sintaxis del funtor es la siguiente:

```
functor Nombre del funtor
        ( Nombre del parámetro : signatura ) : signatura
    struc

        cláusulas

end .
```

Lo que representa un funtor al instanciarlo con uno o más estructuras, es otra estructura que puede satisfacer una signatura dada. Esto se especifica en el funtor al final de los dos puntos.

La estructura *PertArbolBin* tiene la desventaja de que depende de la estructura *ImplArbolBinOcult*. Si se deseara elegir otra implantación del tipo árbol, pero que posea los mismos predicados *raiz*, *arbizq* y *arbder*, se tendría que escribir otra estructura muy similar a la de *PertArbolBin*, habiendo así duplicidad de código. La solución ideal es que la estructura *PertArbolBin* recibiera como parámetro la estructura *ImplArbolBinOcult*. De esta forma, el predicado *pertenece* funciona para cualquier representación del tipo árbol.

El siguiente funtor resuelve este problema:

```
functor PertArbolBinGral( Y : ArbolBinOcult ) =
   struct

      structure X = Y .

      pertenece( Elem, Arbol ) :- X / raiz( Arbol, Elem ) .

      pertenece( Elem, Arbol ) :-
          X / arbIzq( Arbol, ArbIzq ),
          pertenece( Elem, ArbIzq ) .

      pertenece( Elem, Arbol ) :-
          X / arbDer( Arbol, ArbDer ),
          pertenece( Elem, ArbDer ) .

end .
```

Esta estructura recibe como parámetro cualquier estructura que satisfaga la signatura *ArbolBinOcult*. Por ejemplo, la estructura *ImplArbolBinOcult*, sirve como parámetro a este funtor, y la forma de indicarlo es a través de la siguiente expresión:

```
PertArbolBinGral( ImplArbolBinOcult )
```

Esta estructura contiene la cláusula *pertenece*, descrita anteriormente, pero considerando la implantación de árboles definida en la estructura *ImplArbolBinOcult*.

Los autores de este sistema de módulos introducen también en el lenguaje, un conjunto de predicados extralógicos para ser utilizados en las signaturas, estructuras y funtores. El más importante de ellos es *call*, el cual permite demostrar una meta en un funtor instanciado. Pero para poder llamar a través de este predicado los definidos en una estructura o funtor, es necesario establecer primero mediante un valor, una referencia de las estructuras y funtores al que pertenecen, ya que ese valor es necesitado como argumento por el predicado *call*. Por ejemplo, para saber cuál es el valor de referencia de la estructura *PertArbolBinGral( ImplArbolBinOult )* se ejecuta la siguiente meta:

```
structure( Ref, PertArbolBinGral( ImplArbolBinOult ) )
```

el valor de referencia de la estructura está contenido en la variable *Ref*. Ahora por ejemplo, si se desea saber si la letra *a*, pertenece al siguiente árbol:

*arbol( arbol( hoja, z, hoja ), c, arbol( hoja, a, hoja ) )*

utilizando la estructura *PertArbolBinGral( ImplArbolBinOcult )*, se prueba la siguiente meta:

```
call( pertenece( a, arbol( arbol( hoja, z, hoja ), c,
                           arbol( hoja, a, hoja ) ), Ref ) .
```

Otros predicados que existen son *assert* y *retract*, que, respectivamente, agregan y eliminan una cláusula de una signatura o estructura que reciben como parámetro.

En el apéndice *B* aparece una tabla comparativa de algunos conceptos de módulos en la *Programación Lógica Contextual*, *ML-PROLOG*, *BIM-PROLOG* y *SEPIA*.

## 2.4 Características Básicas y Recomendaciones para un Sistema de Módulos en *PROLOG*.

Cualesquiera que sean las características y elementos del sistema de módulos propuesto para *PROLOG*, existen algunos requerimientos mínimos que debe satisfacer el diseño. Estos requerimientos han sido propuestos como básicos por algunos autores como *[ Dorochewsky 91 ], [ O' Kee 85 ]* y otros se consideran también como elementales y se describiran en seguida.

### 2.4.1 Facilidad de Uso e Implantación.

Los elementos del sistema de módulos deben de estar incorporados en el lenguaje *PROLOG* de una manera natural, sin tener que cambiar los conceptos básicos del lenguaje, como el proceso de unificación, los conceptos de metaprogramación, uso y definición de cláusulas de *Horn* y su carácter interpretativo. Así como, no debe ser necesario extender demasiado el lenguaje, de manera que se tengan que desarrollar compiladores e intérpretes difíciles o hacer modificaciones costosas a otras implantaciones que existan, si se va a desarrollar el sistema sobre ellos.

Los módulos del sistema se deben de usar y definir sin que el programador con experiencia se le dificulte. Incluso el programador no necesita saber otro lenguaje por la similitud que tuviera el sistema de módulos con él. También debe ser fácil transcribir, salvo algunas pequeñas modificaciones, un programa hecho sin usar módulos al sistema.

### 2.4.2 Manejo de Encapsulación.

Esta característica es una de las más importantes y es soportada por la mayoría de los sistemas de módulos existentes, por lo que se considera como un requisito indispensable. La encapsulación consiste en la agrupación de un conjunto de entidades, donde los detalles de una representación concreta, como la disposición de los datos, puede no ser visible desde afuera.

### 2.4.3 Facilidad de Alterar un Módulo.

El sistema de módulos debe ser suficientemente flexible como para permitir alterar los elementos de un módulo durante el desarrollo o funcionamiento de un programa. Estos elementos pueden ser las conexiones que tiene el módulo con el medio exterior y las partes que lo componen. Para alterar cada uno de estos elementos, deben existir predicados específicos para agregar o eliminar cada uno de ellos, así como también para eliminar o crear un módulo. Estos requisitos son indispensables si se desea conservar en el sistema de módulos, la característica propia que tiene *PROLOG* de diseño de prototipos y de metaprogramación.

### 2.4.4 Soporte de Metaprogramación.

Esta es una de las características más específicas que debe satisfacer como requisito el sistema de módulos. Ya que aunque a *PROLOG* se le haya incorporado un sistema de módulos, sus características propias de metaprogramación que lo distingue de otros lenguajes, deben de conbinarse bien con el sistema de módulos. No sería correcto que se perdiera la característica de metaprogramación, o que no se hayan tomado en cuenta en el diseño del sistema y que finalmente, no trabajen como se espera con el sistema de módulos.

### 2.4.5 Construido con Bases Sólidas.

Otro de los requisitos específicos que debe cumplir el sistema de módulos, como señala *Richard O' Keefe [ O' Kee 85 ]*, es que tenga una base semántica basada en la programación lógica y no sólo una base sintáctica. Deben existir algunas reglas básicas que establezcan cómo demostrar una meta a partir de un conjunto de módulos, dependiendo cómo se usen éstos. Si es posible, debe existir también una semántica formal que establezca cómo convertir o traducir un programa escrito con ese sistema de módulos al lenguaje de *PROLOG* normal, estableciendo una equivalencia entre ambos lenguajes. Como señala *O' Keefe*, esta semántica enfatiza el carácter metalógico del sistema de módulos, además de que puede guiar fácilmente a construir una implantación.

Según *O' Keefe* existen muchas implantaciones de sistemas de módulos para *PROLOG* que no cumplen con este requisito, como *LM-PROLOG* y *MODULE.PL*. Esto hace que al programar en estos lenguajes se tenga que adoptar una manera similar a otros lenguajes declarativos, ya que no existe conexión con la lógica. El sistema de módulos de estos lenguajes se ve como incorporado a la fuerza.

### 2.4.6 Predicados Básicos que Debe Tener el Sistema.

Existen algunos predicados que se considera que deben estar también en el sistema de módulos, porque son una extensión de los que existen en *PROLOG*. Por ejemplo, así como existen los predicados *assert* y *retract* para alterar el contenido de la base de datos, deben existir predicados similares que afecten el contenido de un módulo dado. Otros predicados útiles, son por ejemplo *clause*, *functor*, *abolish*, *listing*, *keys*, etc.. Todos ellos con el mismo propósito para el cual fueron hechos, pero ahora considerando que un módulo específico es pasado como argumento también. Un predicado muy importante a considerar es *call*, el cual permitirá demostrar una meta bajo uno o mas módulos, en lugar de la base de datos global como lo hace el *call* convencional.

En este capítulo se han discutido varias investigaciones importantes que se han realizado por incorporar módulos a *PROLOG*. En el siguiente capítulo se propondrá un sistema de módulos para *PROLOG*, tomando en cuenta los conceptos presentados en el primer capítulo y las investigaciones mostradas en este capítulo.

# C A P I T U L O  T E R C E R O

## Un Modelo para el Manejo de Módulos en *PROLOG*.

En este capítulo se presenta el diseño de un sistema de módulos para el lenguaje *PROLOG*, objeto de este trabajo. Primero se describen los objetivos y alcances del sistema que se propone. Después, se enuncia los elementos o conceptos de los sistemas de módulos que fueron considerados para construir el sistema propuesto, descritos en el primero y segundo capítulos, Finalmente se presenta el modelo del sistema de módulos desarrollado.

### 3.1 Objetivos del sistema de módulos.

El objetivo básico del sistema de módulos que se propone, es contar con un modelo experimental en el cual se puedan explorar y usar fácilmente los conceptos de modularidad, en el lenguaje *PROLOG*. Siendo éste uno de sus objetivos básicos, la implantación del sistema de módulos no intenta ser eficiente, sino un prototipo sobre el cual evaluar el diseño del modelo. El sistema pretende recoger las ideas que parecieron más interesantes de las implantaciones de *PROLOG* existentes, enunciados en el capítulo anterior.

## 3.2 Consideraciones generales.

Los conceptos de módulo considerados como más importantes de otros lenguajes fueron los siguientes:

Del Lenguaje *OBJ* se tomó fundamentalmente el concepto de:

1) Construcción de módulos parametrizados y expresiones modulares.

Se consideró esta caracerística por las facilidades que ofrece al programador al permitir construir nuevos módulos a partir de otros dados, que a su vez pueden servir como módulos reutilizables para diversas aplicaciones, según los módulos que reciban como parámetros.

Del lenguaje *ADA* se consideró la propiedad de:

1) Importar módulos desde otros módulos y desde un archivo.

El permitir que un módulo pueda importar a otros módulos, permite reducir el código del módulo, así como también evita la repetición de código en otros módulos que lo llegaran a usar. Por otra parte, esta importación evita también que se pasen demasiados parámetros al módulo. El usuario del módulo no tiene por qué dar aquellos parámetros, ya que no contribuyen en forma directa con el objetivo básico del módulo.

Por otra parte, contar con la importación de módulos desde un archivo, evita que el archivo fuente de un programa sea muy grande; haciendo posible reutilizar los módulos que han sido escritos anteriormente en otros archivos. Esta facilidad permite también construir bibliotecas de módulos reusables.

Del lenguaje *ML* se tomaron en consideración las siguientes características:

1) La separación de un módulo en dos partes.

2) Satisfatibilidad de una estructura con una o más signaturas.

3) Permitir el acceso de todos los elementos de un módulo al exterior.

Como se mencionó en el primer capítulo, el dividir un módulo en dos partes: declaración y ejecución, tiene grandes ventajas en un sistema de módulos, por las características ahí descritas.

En cuanto a la satisfatibilidad, permite que la parte declarativa de un módulo, sea satisfecha, en el sentido de *ML*, por varias partes de ejecución de un módulo. Por otra parte, la más importante, la satisfatibilidad establece los requerimientos que deben satisfacer los parámetros de un módulo.

Se decidió considerar la facilidad de hacer accesibles los elementos de un módulo hacia el exterior, por tener la ventaja de poder usar el contenido de un módulo en el exterior, como si no existiera el módulo.

Otros conceptos considerados de algunas otras implantaciones y modelos de *PROLOG*, como la Programación Lógica Contextual fue:

1) La extensión e invalidación de cláusulas al llamar un módulo parametrizado o importado.

Se consideró la propiedad de extensión e invalidación de cláusulas, porque permite considerar otras alternativas de definición, provenientes de otros módulos recibidos como parámetros o importados dentro del módulo.

Y en cuanto a la naturaleza de los módulos en el sistema, se consideró:

2) Los sistemas de módulos basados en sintaxis.

La característica de ser el sistema basado en sintaxis, se decidió simplemente por facilidad, ya que el sistema de módulos propuesto intenta ser sólo un prototipo.

## 3.3 Presentación del Modelo.

### 3.3.1 Componentes Básicos.

Los componentes básicos de un módulo en el modelo que se va a describir son dos: la unidad de definición y la unidad de ejecución. Ambos corresponden, respectivamente, a la parte declarativa y de ejecución de un módulo, como se definió en el primer capítulo.

Un módulo en este sistema queda conformado por dos cláusulas de segundo orden de tipo hecho y de aridad tres. La sintaxis de la unidad de definición de un módulo es la siguiente:

```
mod_defi( Nombre del Módulo,
    impo( Lista de predicados o módulos a importar ),
    pred_expo( Lista de predicados )
    ) .
```

La unidad de definición del módulo, tiene como nombre *mod_defi*; como primer argumento el nombre del módulo; como segundo argumento un funtor cuyo nombre es *impo* y argumento una lista de nombres de predicados o módulos que importa el módulo. Y como tercer argumento posee otro funtor de nombre *pred_expo* y argumento una lista de predicados que exporta el módulo al exterior.

La otra cláusula de tipo hecho que conforma la definición de un módulo, es la unidad de ejecución del módulo, el cual tiene la siguiente sintaxis:

```
módulo( Nombre del Módulo,
        parms( Lista de parámetros ),
        cuerpo( Lista de cláusulas )
      ) .
```

El nombre del predicado de esta unidad es *módulo*. El primer argumento es el nombre del módulo, su segundo argumento es un fun-

tor de nombre *parms* cuyo argumento es una lista de módulos o cláusulas que recibe como parámetros el módulo. Como tercer argumento posee otro funtor de nombre *cuerpo*, cuyo argumento es una lista de cláusulas, que definen el cuerpo de los predicados que conforman el módulo.

En caso de que un módulo no reciba algun parámetro, la palabra *null* debe colocarse como argumento al funtor *parms*.

La ventaja de definir un módulo como una cláusula, radica en la posibilidad de ejecutar predicados extralógicos como son *retract*, *assert*, *abolish*, *arg*, etc.. Ellos permiten alterar las cláusulas que contiene un módulo, así como los parámetros que recibe y los módulos que importa. De hecho, se puede también agregar o quitar unidades o módulos completos en tiempo de ejecución, cumpliendo de esta forma con el tipo de propiedades de metaprogramación, recomendadas en el capítulo dos. Otra ventaja de definir módulos como cláusulas, es que evita la introducción de operadores o estructuras nuevas al lenguaje, que lo harían más complejo y difícil de entender.

Un programa en este sistema de módulos, consiste en un conjunto de módulos y cláusulas que hacen referencia a los módulos para probar una o varias metas. De esta forma, el concepto de módulo es utilizado para agrupar y estructurar un conjunto de cláusulas y hechos, que puedan usarse como medio ambiente local para probar una meta.

El orden de declaración de los módulos e incluso de las unidades que lo componen, es irrelevante. Los módulos incluso pueden definirse en otros archivos, formando así bibliotecas de módulos, donde otros archivos pueden hacer referencia a ellos.

### 3.3.2 Predicados Extralógicos.

Para probar una meta, dentro del medio ambiente de un módulo, se usa el predicado *call* que extiende el sistema. El primer argumento que recibe este predicado, es el nombre del módulo y como segundo argumento la meta a probar. En caso de que el módulo reciba parámetros, el primer argumento del predicado *call* al llamarlo, deberá ser un funtor cuyo nombre es el del módulo y sus argumentos los parámetros correspondientes.

En seguida se presenta un ejemplo para mostrar los conceptos del sistema de módulos antes definidos. El siguiente módulo ordena una secuencia de objetos, utilizando el algoritmo de la burbuja:

```
mod_defi( ordena,
          pred_expo( ord( Xs, Ys ) )

        ) .


módulo( ordena, parms( null ),
        cuerpo( ord( Xs, Ys ) :- cambia( Xs, Zs ), !,
                                 ord( Zs, Ys )
                ord( Ys, Ys )

                cambia( [], [] ) :- fail .
                cambia( [ X, Y | Rest ], [ Y, X | Rest ] ) :-
                                                    X > Y, !
                cambia( [ Z | Rest ], [ Z | Rest1 ] ) :-
                        cambia( Rest, Rest1 ), !
              )
      ) .
```

El predicado llamado *ord*, definido en la unidad de definición del módulo anterior, ordena una lista que recibe como primer argumento y la regresa en el segundo argumento. El cuerpo de este predicado, está definido en la unidad de ejecución del módulo. El predicado *cambia*, por no estar definido en la unidad de definición, se considera local al módulo.

Si ahora se desea ordenar, por ejemplo, la lista: *[ 4, 2, 1, 3 ]* en el módulo anterior, se ejecuta la siguiente meta:

```
call( ordena, ord( [ 4, 2, 1, 3 ], X ) ) .

X = [ 1, 2, 3, 4 ] .
```

Una característica de la unidad de definición, es que permite validar los argumentos de los predicados que exporta un módulo. La validación se hace mediante predicados que identifican el tipo de los argumentos. Estos predicados se colocan después de cada argumento seguido de dos puntos.

Por ejemplo, en caso de que el módulo anterior se usara para ordenar números, la unidad de definición es la siguiente:

```
mod_defi( ordena,
          pred_expo( ord( Xs : list , Ys : var )

                     list( [] )
                     list( [ Cab : number | Rest ] ) :-
                           list( Rest )
          ) .
```

Esta unidad de definición, a diferencia de la anterior, utiliza los predicados *list*, *var* y *number* para validar los argumentos del predicado *ord*. El predicado *list*, cuyas cláusulas que lo definen aparecen en la misma unidad, regresa el valor de verdadero si su argumento es una lista de números. El predicado *var* regresa el valor de verdadero, si su argumento es una variable no instanciada. El predicado *number* toma el valor de verdadero si recibe como argumento un número real o entero. Ambos regresan falsp en cualquier otro caso. Estos predicados se consideran ya incorporados al sistema de *PROLOG*, y como otros que existen también, se pueden usar libremente en cualquier unidad de un módulo.

Otros predicados extralógicos del sistema de módulos son los siguientes:

*assert( M, C )* : Agrega al módulo *M* la cláusula *C*.

*retract( M, C )* : Elimina del módulo *M* la cláusula *C*.

*assertm( C )* : Agrega al módulo donde está presente esta cláusula, la cláusula *C*.

*retractm( C )* : Elimina del módulo donde está presente esta cláusula, la cláusula *C*.

*accexprmod( M )* : Hace visibles al exterior todas las cláusulas definidas en la lista de módulos *M*, los cuales pueden estar parametrizados.

*accexprmodarch( M, A )* : Hace visibles al exterior todas las cláusulas definidas en la lista de módulos *M* del archivo A. Los módulos pueden estar también parametrizados.

### 3.3.3 Importación de Módulos y Predicados.

El sistema de módulos permite que un módulo importe uno o más cláusulas o predicados, así como también uno o más módulos. En caso de que se quiera importar un conjunto de cláusulas o predicados a un módulo, se debe colocar el nombre de cada uno de ellos y su aridad, como argumento al funtor *impo* en la unidad de definición del módulo. Por ejemplo, el módulo *ordena* definido anteriormente puede importar un predicado llamado *compara*, el cual compara dos objetos del tipo de elementos de la lista que se quiere ordenar. Este módulo se define como sigue:

```
mod_defi( ordena,
          impo( compara / 2 ),
          pred_expo( ord( Xs, Ys ) )

        ) .


módulo( ordena, parms( null ),
        cuerpo( ord( Xs, Ys ) :- cambia( Xs, Zs ), !,
                                 ord( Zs, Ys )
                ord( Ys, Ys )

                cambia( [], [] ) :- fail
                cambia( [ X, Y | Rest ], [ Y, X | Rest ] ) :-
                        compara( X, Y ), !
                cambia( [ Z | Rest ], [ Z | Rest1 ] ) :-
                        cambia( Rest, Rest1 ), ! )
      ) .
```

Cuando lo que se desea importar es un conjunto de módulos, el nombre de cada uno de ellos también se define como argumento del funtor *impo*. Los módulos importados pueden estar parametrizados. Por ejemplo, supóngase que se definen dos módulos llamados *listas* y *operlista*. El primero contiene, entre otros predicados, la definición del predicado *list* utilizado en el módulo *ordena* definido anteriormente. El módulo *operlista*, contiene, también entre otros predicados, la definición del predicado *compara*, utilizado también en el módulo *ordena*. Entonces, el módulo *ordena*

definido anteriormente, puede importar los módulos *listas* y *operlista* como se muestra en el siguiente módulo:

```
mod_defi( ordena,
          impo( listas, compara ),
          pred_expo( ord( Xs : list , Ys : var )

        ) .

módulo( ordena, parms( null ),
        cuerpo( ord( Xs, Ys ) :- cambia( Xs, Zs ), !,
                                 ord( Zs, Ys )
                ord( Ys, Ys )

                cambia( [], [] ) :- fail .
                cambia( [ X, Y | Rest ], [ Y, X | Rest ] ) :-
                        compara( X, Y ), !
                cambia( [ Z | Rest ], [ Z | Rest1 ] ) :-
                        cambia( Rest, Rest1 ), !
              )
      ) .
```

Las cláusulas que recibe un módulo a través de importaciones, son colocadas al principio del módulo y después de aquellas cláusulas que se reciben como parámetros.

### 3.3.4 Instanciación Manual de Módulos.

Otro elemento básico del sistema que participa en la construcción de módulos, es el predicado llamado *mapea*.
Existen dos razones básicas por las cuales se decidió incorporar este elemento al sistema. La primera surge al permitir que una unidad de definición de un módulo, tenga más de una unidad de ejecución que la satisfaga. Por ejemplo, si se desea definir en un programa un módulo llamado *inserción*, el cual ordene números por el algoritmo de inserción, aparte del descrito anteriormente llamado *ordena*, se tendría que definir el siguiente módulo:

```
mod_defi( inserción,
          pred_expo( inser( Xs, Ys ) )

        ) .
```

```
módulo( inserción, parms( null ),
        cuerpo( inser( [ X ], [ X ] )
                inser( [ X | Xs ], ListOrd ) :-
                inser( Xs, RestListOrd ),
                inserta( X, RestListOrd, ListOrd )

                inserta( X, [], [ X ] )
                inserta( X , [ Y | Ys ], [ X, Y | Ys ] ) :-
                         X < Y, !
                inserta( X, [ Y | Ys ], [ Y | Ws ] ) :-
                inserta( X, Ys, Ws )
              )
      ) .
```

Como se observa, el contenido de la unidad de definición de este módulo es la misma que la del módulo *ordena*. Para evitar esta duplicidad de código, se incorporó la declaración llamada *mapea*. El objetivo básico de esta declaración, es establecer cuáles de las cláusulas definidas en la parte de ejecución de un módulo, sirven como cuerpo para los predicados definidos en una unidad de definición. La sintaxis de esta declaración, la cual se considerará también como una unidad, es la siguiente:

```
mapea( Nombre de una unidad : Nombre de una unidad
          de ejecución              de definición,

 map( [ Nombre de un predicado / Aridad del Predicado :

        Nombre de un predicado / Aridad del Predicado

      ] ) ) .
```

La definición de esta declaración es nuevamente una cláusula de tipo hecho de aridad dos y cuyo nombre es *mapea*. El primer argumento que recibe, es el nombre de una unidad de ejecución de un módulo seguido de dos puntos y del nombre de una unidad de definición de otro módulo. Esta unidad de ejecución indica de cual se van tomar las cláusulas que van a servir como cuerpo para los predicados definidos en la unidad de definición que se especifica. El segundo argumento es un funtor llamado *map*, cuyo argumento es una lista de predicados y su aridad, relacionados de la siguiente forma:

```
predicado / aridad : predicado / aridad
```

Esta relación establece que el cuerpo del predicado especificado a la izquierda de los dos puntos y definido en la unidad de ejecución que se especificó en la unidad *mapea*, sirve como cuerpo para el predicado especificado a la derecha de los dos puntos y definido en la unidad de definición también especificada en la unidad *mapea*. De esta forma, la unidad de definición del módulo *inserción*, se puede sustituir por la siguiente declaración:

```
mapea( inserción : ordena,
       map( [ inser / 2 : ord / 2 ] ) ) .
```

La segunda razón para incorporar la unidad *mapea*, es que permite establecer cuáles módulos sirven como parámetros a otro módulo. En seguida se describirá primero cómo es la parametrización en el sistema de módulos y luego se explicará el uso de esta declaración en ese contexto.

### 3.3.5 Módulos Parametrizados.

Un módulo puede estar parametrizado por uno o más módulos, así como también por una o más cláusulas. Los módulos que recibe como parámetros pueden a su vez estar parametrizados también. Primero se describirá cómo es la parametrización usando módulos y después la parametrización usando cláusulas o predicados.

La manera de hacer un módulo parametrizado, es indicando en la unidad de ejecución del módulo, en el argumento del funtor *parms*, cuáles son los nombres, separados por comas, de las unidades de definición que un módulo debe satifacer para ser parámetro. Es decir, los parámetros formales de un módulo, que son unidades de definición, establecen qué es lo que debe hacer el parámetro que recibe, el cual es un módulo.

Por ejemplo, el siguiente módulo, llamado *busca*, tiene como fin buscar en una lista de elementos, un elemento dado por el algoritmo de búsqueda binaria.

```
mod_defi( busca,
          pred_expo( busca( Xs, Y ) )

        ) .
```

```
módulo( busca, parms( ordenación ),
        cuerpo( busca( ListElem, Elem) :-
                  ordena( ListElem, ListOrd ),
                  long( ListOrd, Long ),
                  busqbin( ListOrd, Elem, 1, Long  )

                busqbin( List, Elem, Inic, Fin ) :-
                  IndMedR is ( Inic + Fin ) /  2,
                  IndMed is integer( IndMedR ),
                  elemmed( List, IndMed, 1, ElemMed ),
                  ifthenelse( Elem > ElemMed,
                    ( IndMedSig is IndMed + 1,
                      ifthen( IndMedSig > Fin, fail ),
                      busqbin( List, Elem, IndMedSig, Fin  )
                    ),
                    ( ifthenelse( Elem < ElemMed,
                        ( IndMedAnt is IndMed - 1,
                          busqbin( List, Elem, Inic,
                                   IndMedAnt )
                        ),
                        true )
                    )
                  )

                elemmed( [ Elem | RestList ], IndElem,
                         IndElem, Elem )

                elemmed( [ Elem | RestList ], IndElem, Cont,
                         ElemMed) :-
                    ContSig is Cont + 1,
                    elemmed( RestList, IndElem, ContSig,
                             ElemMed )

                long( [ E ], 1 )

                long( [ C | R ], L ) :- long( R, LR ),
                                         L is 1 + LR
              )
      ) .
```

En el módulo, la operación de búsqueda la realiza a través del predicado *busca*, el cual regresa verdadero si un elemento, que es el que recibe como segundo argumento, está en la lista que recibe como primer argumento. Este predicado, antes de aplicar el algoritmo de búsqueda binaria, tiene que primero ordenar la lista de elementos, ya que el algoritmo lo requiere así. El módulo recibe como parámetro, un módulo que define el algoritmo por el cual se va a ordenar la lista.

La siguiente unidad de definición es el parámetro formal del módulo *busca*:

```
mod_defi( ordenación,
          pred_expo( ordena( Xs, Ys ) )

        ) .
```

De acuerdo a la definición de esta unidad, los módulos *ordena* e *inserción* definidos antes, sirven como parámetros al módulo *busca*. Antes de instanciar el módulo *busca* con alguno de estos módulos, se debe indicar que ambos son módulos que ejecutan lo requerido en el parámetro formal, es decir, satisfacen las solicitudes de ejecución establecidas en la unidad de definición *ordenación*. La forma de indicar ésto para ambos módulos, es usando la declaración *mapea* de la siguiente manera:

```
mapea( ordena : ordenación,
       map( [ ord / 2 : ordena / 2 ] ) ) .

mapea( inserción : ordenación,
       map( [ inser / 2 : ordena / 2 ] ) ) .
```

Una vez establecida esta relación entre la unidad de definición *ordenación* (parámetros formales) y los módulos *ordena* e inserción (parámetros actuales) se puede usar el módulo busca con cualquiera de ellos. Por ejemplo, para saber si el número *45* está en la lista *[ 23, 67, 34, 21, 55, 67, 88 ]*, usando el algoritmo de ordenación de la burbuja, se utiliza la siguiente meta:

```
call( busca( ordena ),
      busca( [ 23, 67, 34, 21, 55, 67, 88 ], 45 ) .
```

Si ahora se desea usar el algoritmo de inserción, se usa la siguiente meta:

```
call( busca( inserción ),
      busca( [ 23, 67, 34, 21, 55, 67, 88 ], 45 ) .
```

A continuación de muestran algunas otras metas válidas que hacen uso del módulo busca:

```
buscaelemalg( Elem, Algorit ) :-
   call( busca( Algorit ),
         busca( [ 34, 22, 1, 88, 6, 7, 44 ], Elem ) ) .

buscaelemalg( Lista, Elem ) :-
   call( busca( inserción ), busca( Lista, Elem ) ) .

buscaelemalg( Lista, Elem, Algorit ) :-
   call( busca( Algorit ), busca( Lista, Elem ) ) .
```

Puede suceder en algún caso particular, que un módulo satisfaga los requerimientos de la unidad de definición del parámetro formal de un módulo, pero que contenga más cláusulas que los solicitados en esa unidad. En este caso se elige aquellas cláusulas requeridas en la unidad de definición, mediante la lista de asociación de predicados que recibe el predicado *map* en la declaración de *mapea*.

La manera de hacer un módulo parametrizado por una o más cláusulas o predicados, es indicando nuevamente en el argumento del funtor *parms*, de la unidad de ejecución, cuáles son los nombres y aridad de las cláusulas o predicados, que recibe como parámetros el módulo. Si el parámetro es un conjunto de cláusulas o predicados, cada uno de ellos va separado por una coma, y el nombre y aridad de cada uno de ellos va separado por una diagonal. Por ejemplo, si el módulo anterior es ahora parametrizado por el predicado *ordena*, su definición es la siguiente:

```
mod_defi( busca,
          pred_expo( busca( Xs, Ys ) )

          ) .


módulo( busca, parms( ordena / 2 ),
        cuerpo( busca( ListElem, Elem) :-
                    ordena( ListElem, ListOrd ),
                    long( ListOrd, Long ),
                    busqbin( ListOrd, Elem, 1, Long  )
```

```
                busqbin( List, Elem, Inic, Fin ) :-
                  IndMedR is ( Inic + Fin ) / 2,
                  IndMed is integer( IndMedR ),
                  elemmed( List, IndMed, 1, ElemMed ),
                  ifthenelse( Elem > ElemMed,
                    ( IndMedSig is IndMed + 1,
                      ifthen( IndMedSig > Fin, fail ),
                      busqbin( List, Elem, IndMedSig, Fin  )
                    ),
                    ( ifthenelse( Elem < ElemMed,
                        ( IndMedAnt is IndMed - 1,
                          busqbin( List, Elem, Inic,
                                   IndMedAnt )
                        ),
                        true )
                    )
                  )

                elemmed( [ Elem | RestList ], IndElem,
                         IndElem, Elem )

                elemmed( [ Elem | RestList ], IndElem, Cont,
                         ElemMed) :-
                   ContSig is Cont + 1,
                   elemmed( RestList, IndElem, ContSig,
                            ElemMed )

                long( [], 0 ) .

                long( [ C | R ], L ) :- long( R, LR ),
                                         L is 1 + LR

            )
        ) .
```

El predicado *ordena*, que recibe como parámetro el módulo anterior, ordena una lista que a su vez recibe como primer argumento y la regresa ordenada en el segundo. La instanciación de un módulo con un predicado o cláusula que va a recibir como parámetro, se hace colocando como argumento los nombres de estos predicados o cláusulas, junto con su aridad, al nombre del módulo cuando es llamado. Por ejemplo, suponiendo que las cláusulas *ord* e *inser*, declaradas, respectivamente en los módulos *ordena* e *inserción*, están definidas fuera del módulo *busca*, entonces para saber nuevamente si el número *45* está en la lista *[ 23, 67, 34, 21, 55, 67, 88 ]*, usando el algoritmo de ordenación de la burbuja, se utiliza la siguiente meta:

```
call( busca( ord / 2 ),
      busca( [ 23, 67, 34, 21, 55, 67, 88 ], 45 ) .
```

Si ahora se desea también usar el algoritmo de inserción, se usa la siguiente meta:

```
call( busca( inser / 2 ),
      busca( [ 23, 67, 34, 21, 55, 67, 88 ], 45 ) .
```

A continuación de muestran algunas otras metas también válidas que hacen uso del módulo *busca* parametrizado por una cláusula:

```
buscaelemalg( Elem, Algorit ) :-
   call( busca( Algorit / 2 ),
         busca( [ 34, 22, 1, 88, 6, 7, 44 ], Elem ) ) .
```

```
buscaelemalg( Lista, Elem ) :-
   call( busca( inser / 2 ), busca( Lista, Elem ) ) .
```

```
buscaelemalg( Lista, Elem, Algorit ) :-
   call( busca( Algorit / 2 ), busca( Lista, Elem ) ) .
```

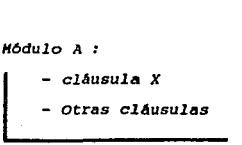
### 3.3.6 Mecanismos de Ocultamiento.

Los mecanismos de ocultamiento que tiene el sistema son muy similares a las de los lenguajes *SML* y *ADA*. En un módulo, todos los predicados que aparecen en la unidad de definición, pueden ser llamados fuera del módulo. Las cláusulas declaradas en la unidad de ejecución del módulo, y no en cualquiera de las unidades de declaración que satisfaga, son locales a él y no pueden ser accesadas fuera del módulo. No se permite dentro de un módulo, usar el predicado *call*.

### 3.3.7 Atributos de cláusulas definidas en un módulo.

Al hacer la instanciación de módulos parametrizados, es común que alguno de los parámetros que son recibidos, traiga nuevas definiciones de cláusulas, al módulo que las recibe. Sin embargo, por alguna razón se desea tomar ahora en cuenta esas cláusulas e ignorar las que están en el módulo que las recibe. Esto, porque por ejemplo, son más eficientes o involucran otras cláusulas a considerar.

Permitir la posibilidad de usar cláusulas y módulos de esta forma, puede causar la duplicidad de código de módulos. Por ejemplo, considérese la siguiente situación. Supóngase que existe un módulo llamado *A*, el cual define una cláusula llamada *X* y algunas otras:




Supóngase ahora que se definen dos módulos, llamados *B* y *C*, parametrizados por el módulo *A*. Se desea usar *B* y *C* de la forma *B( A )* y *C( A )*, pero con la siguiente restricción. En la expresión *C( A )* el módulo *C* necesita a la cláusula *X* y las demás cláusulas definidas allí. Pero en la expresión *B( A )* el módulo *B* no requiere extender la definición de la cláusula *X* definida también en él, pero sí necesita de las otras cláusulas definidas en el módulo *A*.

Una alternativa para resolver este problema es construir un nuevo módulo, el cual contenga las cláusulas definidas en el módulo *A*, excepto la cláusula *X*. Este nuevo módulo servirá ahora de parámetro al módulo *B*.

El inconveniente de esta solución es que por una parte existe duplicidad de código, el definido en el módulo *A* como otras cláusulas. Por otra parte, en algunos casos, no es fácil dividir un módulo de esta forma, porque ello afectaría también a otros módulos o porque el módulo forma parte de una biblioteca y no se proporciona el código de la unidad de ejecución.

Para resolver este tipo de problemas y otros muy similares, que se describirán más adelante, el sistema de módulos introduce el concepto de atributos de cláusulas, las cuales tienen un propósito específico. Los atributos se colocan al principio de la definición de las cláusulas especificadas en la unidad de ejecución de un módulo seguidas de dos puntos.

Uno de ellos es el atributo llamado *pri*, el cual resuelve precisamente el problema planteado anteriormente. Este atributo al colocarlo al principio de una cláusula, significa que si exis-

ten otras con el mismo nombre y aridad que provengan de otros módulos, ya sea parametrizados o importados, no son tomadas en consideración.

La solución al problema descrito anteriormente usando este atributo, consiste en colocarlo al principio de la cláusula llamada *X* en el módulo *B*. Los módulos *A* y *C* permanecen igual. Así, al formar la expresión modular *B( A )*, la cláusula *X* del módulo *A* no existe, solo la de *B*.

Otro de los los atributos que existen en el sistema es *inv*. Este tiene un propósito básicamente opuesto al anterior. Es decir, colocado al principio de una cláusula, significa que si existen otras cláusulas con el mismo nombre y aridad que provengan de otros módulos, ya sean parametrizados o importados, son tomados ahora en cuenta, mientras que la cláusula que se definió con el atributo no se tomará en cuenta.

Este atributo resolvería problemas similares al siguiente. Supóngase que se tiene nuevamente el módulo *A*, descrito en el caso anterior, pero que ahora recibe como parámetros un módulo *B* y *C*, ambos distintos a los anteriores. El módulo *C* define un conjunto de cláusulas excepto a la cláusula *X*. El módulo *B* define también un conjunto de cláusulas y también la cláusula *X*.

*módulo C :*

- *cláusulas*

*módulo B :*

- *cláusula X*
- *Otras cláusulas*

Ahora los módulos se requieren usar de la forma *A( C )* y *A( B )*, pero con la siguiente restricción. En la expresión *A( B )*, se requiere que ahora sea considerada la cláusula *X* de *B* y no la de *A*. Nuevamente una alternativa para solucionar este problema es construir otro módulo que contenga las cláusulas definidas en el de *A*, excepto la cláusula *X*. Este nuevo módulo serviría ahora para formar la expresión *A( B )*, con la restricción señalada antes. Los inconvenientes de esta solución son iguales a los planteados en el caso anterior.

Usando el atributo *inv* en la cláusula *X* del módulo *A*, se soluciona este problema, sin los inconvenientes antes descritos.

Un atributo más que existe en el sistema es *ext*. El propósito que éste tiene es considerar, aparte de la definición de las cláusulas que poseen este atributo, otras con el mismo nombre y aridad que provengan de otros módulos, ya sea parametrizados o importados también.

Ya que en el desarrollo de programas, con el uso de módulos, es más común usar el atributo *ext* que *pri* e *inv*, se decidió, en el diseño del sistema, que éste fuese el atributo usado por defecto y de manera opcional, en la declaración de las cláusulas de un módulo.

### 3.3.8 Importación de Módulos desde otros archivos.

En el sistema de módulos, es posible que un programa importe un conjunto de módulos de otro programa. La manera de hacer la importación de estos módulos, es utilizando el predicado *include* que proporciona el sistema. Este predicado recibe como argumento el nombre del archivo donde están el conjunto de módulos a importar. El nombre del archivo es especificado como una secuencia de caracteres, en la manera usual de PROLOG. Este predicado debe ser colocado al principio del programa que lo va a utilizar, precedido por el símbolo ":-" y finalizando con un punto.

En el apéndice *C* aparece la misma tabla del apéndice *B*, pero incluido como lenguaje el prototipo diseñado en este capítulo.

# C A P I T U L O  C U A R T O

## Implantación del Sistema de Módulos Propuesto.

### 4.1 Generalidades de la implantación.

En este último capítulo, se explicará la implantación del modelo diseñado para el manejo de módulos en *PROLOG*, descrito en el capítulo tercero.

El sistema fue escrito totalmente en *PROLOG*. Se decidió hacerlo así, por la ventaja que tiene *PROLOG* de poder diseñar rápidamente prototipos y verlos funcionar. Además, porque una vez que funciona, es fácil incorporarlo al lenguaje *PROLOG* o desarrollar una implantación más eficiente y formal en algun otro lenguaje. En particular se desarrolló el sistema de módulos en *Arity Prolog* versión 6.00 1990-1991, la cual es una implantación comercial que existe para *PROLOG [ Arity 91 ]*. Esta implantación tiene la ventaja de trabajar en un ambiente de computadora personal y ser bastante estándar. Esta última característica hace que el sistema sea fácil de transportar a algunas otras implantaciones comerciales de *PROLOG*, como son *BIM-Prolog*, *Quintus-Prolog*, etc.. Esto permite la utilización del sistema en computadoras grandes y estaciones de trabajo con ambiente *UNIX*.

Una vez desarrollada la implantación del sistema, ésta se incorporó al compilador de *Arity Prolog*, siendo ésta una de las facilidades que proporciona. Esta característica permite agregar cláusulas a la base de datos del compilador, las cuales el usuapuede llamar libremente después, sin necesidad de escribirlas cada vez que las necesite de nuevo en la base de datos. De esta forma, el sistema de módulos funciona más rápido y da la apariencia de formar parte del sistema original de *Arity Prolog*.

## 4.2 Funcionamiento de la Implantación.

El funcionamiento del sistema de módulos es el siguiente. Cuando el intérprete de *Arity Prolog* intenta probar una meta que es *call*, en donde el primer argumento es una expresión modular, el sistema de módulos entra en operación. Lo que realiza este predicado, en esencia, es primero agregar a la base de datos del intérprete, cada una de las cláusulas y términos que componen la expresión módular. Es decir, agrega el contenido del módulo referido, incluso los módulos importados y los recibidos como parámemetros. Y segundo, lo que realiza es probar la meta o las metas que tiene como segundo argumento el predicado *call*.

La diferencia entre cada una de las cláusulas y términos agregados a la base de datos, y las que aparecen en los módulos a los que se hacen referencia en el primer argumento del predicado *call*, es que el sistema de módulos, al irlos agregando a la base de datos del intérprete, les coloca un argumento más a cada uno de ellos. Este argumento es una clave que asocia a cada una de las cláusulas y términos de un módulo, el nombre de módulo al cual pertenecen. Esta clave es desconocida por el usuario del sistema de módulos. De esta forma se construye, a través de un argumento que es común a un conjunto de cláusulas que pertenecen a un módulo, un medio ambiente local en la base de datos en el cual se va a probar la meta propuesta, que aparece como segundo argumento del predicado *call*. El resto de la base de datos no es tomada en cuenta, por no tener esa clave como argumento. Una clave es también agregada como argumento a cada una de las metas a probar, que aparecen como segundo argumento del predicado *call*. Esta clave, en particular, es el nombre del módulo en el cual se va a probar la meta, incluyendo los parámetros con que fue llamado.

La definición del predicado *call* es la siguiente:

```
call( ExprMod, Claus ) :-
    sepmodparm( ExprMod, Módulo, LlaveDef, LlaveMod ),
    agrdefmod( LlaveDef, LlaveMod, Módulo ),
    agrmódulo( LlaveMod, ExprMod ),
    calls( LlaveDef, Claus ) .                    ..... ( 1 )
```

En seguida se describirá cada uno de los predicados que componen el cuerpo de esta cláusula.

El primer predicado que llama la cláusula que define el predicado *call*, es *sepmodparm*, y su definición es la siguiente:

```
sepmodparm( ExprMod, Módulo, LlaveDef, LlaveMod ) :-
    string_term( LlaveMod, ExprMod ),
    functor( ExprMod, Módulo, AridExprMod ),
    string_term( LlaveDef, Módulo ) .          ..... ( 2 )
```

El predicado *sepmodparm* tiene dos objetivos, primero separar el nombre del módulo que fue llamado en el predicado *call*, de sus parámetros y segundo, construir las claves que asocian el nombre de un módulo con las cláusulas que contiene. El predicado *sepmodparm* recibe como primer parámetro, el módulo llamado en el predicado *call*, y regresa, en el segundo argumento, el nombre del módulo sin los parámetros con que fue llamado. En caso de que el módulo sea llamado sin parámetros, regresa de nuevo el nombre del módulo. El tercer y cuarto argumento que regresa el predicado *sepmodparm*, son las claves que asocian a las cláusulas definidas en la unidad de definición y de ejecución de un módulo, con el nombre de la unidad a la cual pertenecen. Ambas claves serán llamadas, de ahora en adelante y respectivamente, la clave de asociación de unidad de definición y clave de asociación de unidad de ejecución. Estas claves son una cadena de caracteres que representan, por una parte, el nombre del módulo que se llama y por otra el nombre del módulo junto con los parámetros con que fue llamado en el predicado *call*.

En la cláusula (2), que define al predicado *sepmodparm*, las cadenas de caracteres son formadas a través del predicado *string_term* que está incorporado en el intérprete de *Arity Prolog*. Este predicado convierte un término en una cadena de caracteres y viceversa. El predicado *functor* es otro predicado incorporado al intérprete. Este predicado permite conocer el nombre y la aridad del funtor, que recibe como primer argumento. Este predicado es usado en la cláusula (2) para regresar la aridad y el nombre del funtor que recibe como primer argumento, el cual es el nombre del módulo junto con sus parámetros que se llama en el predicado *call*.

El segundo predicado que llama la cláusula que define el predicado *call*, es *agrdefmod*. La finalidad de este predicado, es agregar a la base de datos, el contenido de la unidad de definición del módulo que se llama en el predicado *call*.

Las cláusulas que componen la definición del predicado *agrdefmod* son las siguientes:

```
agrdefmod( LlaveDef, LlaveMod, Módulo ) :-
   mod_defi( Módulo, impo( ModPred ), pred_expo( ListPred ) ),
   agrImpo( LlaveDef, LlaveMod, ModPred ),
   agrpredmod( LlaveDef, LlaveMod, ListPred ) . ..... ( 3 )

agrdefmod( LlaveDef, LlaveMod, Módulo ) :-
   mod_defi( Módulo, pred_expo( ListPred ) ),
   agrpredmod( LlaveDef, LlaveMod, ListPred ) . ..... ( 4 )
```

El primer argumento que reciben estas cláusulas, *LlaveDef*, es la llave de asociación de unidad de definición, el segundo argumento, *LlaveMod*, es la llave de asociación de unidad de ejecución, y el tercer argumento, *Módulo*, es el nombre del módulo que se llama en el predicado *call*. A continuación se describirá cada una de estas cláusulas.

La cláusula (3), considera el caso cuando el módulo que recibe como tercer argumento, importa cláusulas o módulos, mientras que la cláusula (4) no. Lo primero que realizan las cláusulas (3) y (4), es buscar la unidad de definición del módulo que reciben como tercer argumento, mediante la unificación del predicado *mod_defi*, con aquel que exista en la base de datos y que coincida, en el primer argumento, con el nombre del módulo. En caso de que el módulo importe cláusulas u otros módulos, el predicado *agrimpo*, que llama la cláusula (3), se encarga de agregarlos a la base de datos.

Las cláusulas que componen la definición del predicado *agrimpo*, son las siguientes:

```
agrimpo( LlaveDef, LlaveMod, [ ExprMod | RestImpo ] ) :-
   functor( ExprMod, Módulo, AridExprMod ),
   agrdefmod( LlaveDef, LlaveMod, Módulo ),
   agrmódulo( LlaveMod, ExprMod ),
   agrimpo( LlaveDef, LlaveMod, RestImpo ) . ..... ( 5 )

agrimpo( LlaveDef, LLaveMod,
         [ NomPred / AridPred | RestImpo ] ) :-
   functor( PredImp, NomPred, AridPred ),
   PredImp =.. [ NomPred | ArgPred ],
   Pred =.. [ NomPred | LlaveMod, ArgPred ],
   assert( ( Pred :- PredImp ) ) .          ..... ( 6 )
```

La primera de estas cláusulas considera el caso cuando el módulo que se llama en el predicado *call*, importa módulos y la segunda, predicados. En caso de que el módulo importe módulos, la cláusula (5) agrega a la base de datos el contenido de los módulos importados, a través de los predicados *agrdefmod* y *agrmódulo*, los cuales agregan, respectivamente, a la base de datos el contenido de la unidad de definición y de ejecución del módulo importado. Antes de agregar en la cláusula (5), el cuerpo de un módulo, esta cláusula determina cual es el nombre del módulo y que parámetros tiene, lo cual lo realiza a través del predicado *functor*. Si el módulo importa predicados, la cláusula (6) establece, mediante una cláusula, una liga entre el nombre del predicado importado, llamado *NomPred / AridPred* en la cláusula (6) y el que existe fuera del módulo y que se va a importar. De acuerdo a los nombres de las variables que aparecen en la cláusula (6), esta cláusula es la siguiente:

*NomPred( LlaveMod, Argumentos ) :- NomPred( Argumentos )*

en donde:

*NomPred* : es el nombre del predicado importado;

*LlaveMod* : es la llave de asociación de unidad de ejecución;

*Argumentos* : son los argumentos de la cabeza y cuerpo de la cláusula, los cuales deben coincidir.

Lo que hace entonces la cláusula (6), es construir la cláusula *NomPred*. El primer predicado que llama la cláusula (6) es *functor( PredImp, NomPred, AridPred )*, el cual construye el predicado *NomPred( Argumentos )*. La siguiente instrucción, *PredImp =..[ NomPred | ArgPred ]*, obtiene los argumentos del predicado *NomPred( Argumentos )*. Luego la instrucción *Pred =.. [ NomPred | LlaveMod, ArgPred ]* construye el predicado *NomPred( LlaveMod, Argumentos )*. Finalmente el predicado *assert( ( Pred :- PredImp ) )* construye la cláusula *Pred* y la agrega a la base de datos del intérprete.

Una vez agregados a la base de datos del intérprete los predicados o módulos que importa el módulo llamado en el predicado *call*, el último predicado que llama la cláusula *agrdefmod*, *agrpredmod*, agrega los predicados de la unidad de definición de este módulo.

La cláusula que define el predicado *agrpredmod* es la siguiente:

```
agrpredmod( LlaveDef, LlaveMod, [ Pred | RestPred ] ) :-
   constrclaus( LlaveDef, LlaveMod, Pred, ( Cab :- Cuerp ) ),
   assert( ( Cab :- Cuerp ) ),
   agrpredmod( LlaveDef, LlaveMod, RestClaus ) .    .... ( 7 )
```

Este predicado recibe como primer argumento la clave de asociación de unidad de definición, como segundo argumento la clave de asociación de unidad de ejecución y como tercer argumento, una lista de predicados que conforman el cuerpo de la unidad de definición del módulo llamado en el predicado *call*. La cláusula *agrpredmod*, llama primero al predicado *constrclaus*, el cual construye una cláusula con el primer predicado de la lista que recibe como tercer argumento. El predicado *constrclaus* regresa en el tercer argumento, la cláusula que construye, que en la cláusula (7) es *Cab :- Cuerp*. Una vez construida esta cláusula, la cláusula (7) la agrega a la base de datos mediante el predicado *assert( ( Cab :- Cuerp ) )*. Finalmente, la cláusula *agrpredmod* agrega a la base de datos, el resto de la lista que tiene como terecer argumento, llamándose recursivamente con el resto de esta lista.

El tercer predicado que llama la cláusula *call* es *agrmódulo*. Este predicado tiene como fin, agregar a la base de datos la unidad de ejecución del módulo que fue llamado en el predicado *call*, así como los módulos que fueron puestos como parámetros. La cláusula que define este predicado es la siguiente:

```
agrmódulo( LlaveMod, ExpMod ) :-
   ExpMod =.. [ Módulo | ParmsAct ],
   módulo( Módulo, parms( ParmsForm ), cuerpo( Claus ) ),
   parmact( ParmsAct, ParmsActPrinc ),
   agrinterfs( LlaveMod, ParmsActPrinc, ParmForm ),
   agrlistmod( LlaveMod, ParmsAct ),
   agrclamod( LlaveMod, Claus ) .              ..... ( 9 )
```

Esta cláusula recibe como primer argumento la llave de asociación de unidad de ejecución, y como segundo argumento el nombre del módulo junto con sus parámetros llamado en el predicado *call*. Lo primero que hace esta cláusula, es separar el nombre del módulo de sus parámetros, mediante la instrucción *ExpMod =..[ Módulo | ParmsAct ]*, en donde la variable *Módulo* es el nombre del módulo llamado en el predicado *call* y *ParmsAct* su lista de parámetros. La cláusula *agrmódulo* busca después la unidad de ejecución del módulo llamado, mediante la unificación del predicado *módulo* con

aquel que exista en la base de datos, y que coincida, en el primer argumento, con el nombre del módulo llamado en el predicado *call*. Al hacer esta unificación del predicado *módulo*, se obtienen cuales son los parámetros formales y el cuerpo del módulo llamado, los cuales están guardados en forma de lista en los predicados *parms( ParmsForm )* y *cuerp( Claus )*, respectivamente. El siguiente predicado que llama la cláusula *agrmodulo* es el predicado *parmact( ParmsAct, ParmsActPrinc )*. En caso de que el módulo haya sido llamado con parámetros, los cuales pueden ser módulos parametrizados también, el predicado *parmact* separa los nombres de los módulos de cada uno de ellos de los parámetros que a su vez tienen y coloca los nombres de los módulos en una lista, que es la que regresa como segundo argumento el predicado *parmact*. Si los parámetros con que fue llamado un módulo en el predicado *call* son predicados, el predicado *parmact* los devuelve sin ninguna modificación en la lista. En caso de que el módulo no reciba parámetros, el predicado *parmact* regresa una lista vacía.

Las cláusulas que definen al predicado *parmact* son las siguientes:

```
parmact( [], [] ) .                                    ..... ( 10 )

parmact( [ Pred / Arid | RestParm ],
         [ Pred / Arid | RestNomParm ] ) :-
    parmact( RestParm, RestNomParm ) .                 ..... ( 11 )

parmact( [ Módulo | RestParm ],
         [ NomMódulo | RestNomParm ] ) :-
    functor( Módulo, NomMódulo, AridMódulo ),
    parmact( RestParm, RestNomParm ) .                 ..... ( 12 )
```

La cláusula (12), separa los nombres de los módulos de cada uno de sus parámetros mediante el predicado *functor*, y se llama luego recursivamente para hacer lo mismo con el resto de los módulos que son parámetros del módulo llamado en el predicado *call*.

Una vez que se obtienen con el predicado *parmact* los parámetros con que fue llamado el módulo en el predicado *call*, y se tienen los parámetros formales de este módulo en el predicado *parms( ParmsForm )*, el siguiente predicado que llama la cláusula *agrmódulo*, es *agrinterfs*, el cual establece la comunicación entre ambos parámetros.

Las definiciones de este predicado son las siguientes:

```
agrinterfs( LlaveMod, [ NomPredAct / AridPredAct |
                        RestParmAct ],
          [ NomPredForm / AridPredForm | RestParmForm ] ) :-
   agrinterf( LlaveMod, NomPredAct / AridPredAct,
   NomPredForm / AridPredForm ),
   agrinterfs( LlaveMod, RestParmAct, RestParmForm ) . ..( 12 )


agrinterfs( LlaveMod, [ ModParmAct | RestParmAct ],
          [ DefForm | RestParmForm ] ) :-
   agrinterf( LlaveMod, ModParmAct, DefForm ),
   agrinterfs( LlaveMod, RestParmAct, RestParmForm ) . ..( 13 )
```

Ambas cláusulas reciben como primer argumento la llave de asociación de unidad de ejecución, como segundo argumento la lista de parámetros con los cuales fue llamado el módulo en el predicado *call* (parámetros actuales) y como tercer argumento, reciben una lista de parámetros los cuales hace referencia el cuerpo del módulo llamado en el predicado *call* (parámetros formales).

La cláusula (12) considera el caso cuando el módulo llamado en el predicado *call*, está parametrizado por un predicado, mientras que la cláusula (13) cuando el parámetro es un módulo. Cada una de estas dos cláusulas establece la comunicación entre cada parámetro actual con su correspondiente parámetro formal. Lo que hace cada una estas cláusulas, es llamar primero al predicado *agrinterf*, quien se encarga de establecer la comunicación entre el primer parámememetro de la lista de parámetros actuales, con el primer parámetro de la lista de parámetros formales, y luego él mismo se llama recursivamente para establecer la comunicación con el resto de los parámetros en ambas listas. Cuando el módulo llamado en el predicado *call*, tiene como parámetro un predicado, la cláusula que define al predicado *agrinterf* es la siguiente:

```
agrinterf( LlaveMod, NomPredAct / AridPredAct,
           NomPredForm / AridPredForm ) :-
   functor( PredAct, NomPredAct, AridPredAct ),
   PredAct =.. [ NomPredAct | ArgsPredAct ],
   PredForm =.. [ NomPredForm, LlaveMod | ArgsPredAct ],
   nuevcab( LlaveMod, PredAct, PredActL1 ),
   assert( ( PredForm :- PredActL1 ) ) .        ..... ( 14 )
```

Esta cláusula recibe como primer argumento la llave de asociación de unidad de ejecución, como segundo argumento, el nombre y aridad del predicado, que funge como parámetro en el módulo llamado en el predicado *call*. Y como tercer argumento recibe el predicado junto con su aridad, al cual se hace referencia como parámetro en el cuerpo del módulo llamado en la cláusula *call*. El objetivo de la cláusula *agrinterf*, es construir una cláusula para establecer una comunicación entre ambos parámetros del módulo llamado en la cláusula *call*. La cabeza de la cláusula que construye *agrinterf* es el predicado al que se hace referencia como parámetro en el cuerpo del módulo que se llama en el predicado *call* y el cuerpo de la cláusula es el predicado que funge como parámetro en la llamada del mismo módulo en el predicado *call*.
De esta forma, de acuerdo a los nombres de los argumentos que recibe la cláusula (14), la cláusula que construye es la siguiente:

*NomPredForm( LlaveMod, Argumentos ) :-*
*NomPredAct( LlaveMod, Argumentos ) .* ..... ( 15 )

en donde:

*NomPredForm* : es el nombre del predicado que se hace referencia como parámetro en el cuerpo del módulo que se llama en el predicado *call*;

*LlaveMod* : es la llave de asociación de unidad de ejecución;

*NomPredAct* : es el nombre del predicado que funge como parámetro en la llamada del módulo en el predicado call y

*Argumentos* : son los argumentos de ambos predicados en la cláusula.

De esta forma, cuando en el cuerpo del módulo que fue llamado en el predicado *call*, se hace referencia al predicado parametrizado, esta cláusula liga su definición con la del predicado que funge como parámetro en el módulo llamado en el predicado *call*.

El primer predicado que llama la cláusula *agrinterf*, es el predicado *functor( PredAct, NomPredAct, AridPredAct )*, el cual construye el predicado *NomPredAct( argumentos )*, a partir de su nombre y aridad, que son recibidas como argumentos. Este predicado es llamado por *functor* como *PredAct*. La siguiente instrucción que llama la cláusula (14) es *PredAct =..[NomPredAct | ArgsPred-*

*Act ]*, la cual obtiene los argumentos del predicado *PredAct*. La siguiente instrucción que llama es *PredForm =.. [ NomPredForm, Lla-veMod | ArgsPredAct ]*, la cual construye el predicado *Nom-PredPredForm( LlaveMod, argumentos )* a partir del nombre del predicado parametrizado, su aridad, los argumentos, que son los mismos a los del predicado *PredAct* y la clave de asociación de unidad de ejecución. El siguiente predicado *nuevcab( LlaveMod, Pred-Act, PredActLl )*, tiene como objetivo agregar la clave de asociación de unidad de ejecución, llamada *LlaveMod*, como primer argumento, al predicado *PreddAct* y regresa este predicado modificado como tercer argumento (*PredActLl*).

La definición del predicado *nuevcab* es la siguiente:

```
nuevcab( LlaveMod, Term, NuevCab ) :-
   Term =.. [ CabListTerm | RestListTerm ],
   NuevCab =.. [ CabListTerm, LlaveMod |
                 RestListTerm ] .              ..... ( 16 )
```

Esta cláusula recibe como primer argumento la clave de asociación de unidad de ejecución, como segundo argumento el predicado al cual se quiere agregar y como tercero, el predicado modificado. La cláusula *nuevcab* es utilizada para construir a partir del predicado *NomPredAct( argumentos )*, el predicado *NomPredAct( Llave-Mod, argumentos )*. Una vez construida la cabeza y el cuerpo de la cláusula (16), el último predicado que llama la cláusula *agrinterf*, forma la cláusula y la agrega a la base de datos mediante el prdicado *assert( ( PredForm :- PredActLl ) )*.

Cuando el módulo llamado en el predicado *call* tiene como parámetros módulos, la cláusula que define al predicado *agrinterf* es la siguiente:

```
agrinterf( LlaveMod, ModParmAct, DefForm ) :-
   mapea( ModParmAct : DefForm, map( ListMap ) ),
   mod_defi( DefForm, pred_expo( Pred ) ),
   agrclausinterf( LlaveMod, ListMap, Pred ) .   ..... ( 17 )
```

El objetivo de esta cláusula es establecer la comunicación entre los predicados a los que se hace referencia como parámetros, a través de una unidad de definición, en el cuerpo del módulo llamado en el predicado *call* y los predicados contenidos en un módulo, el cual funge como parámetro de un módulo . El primer argumento que recibe la cláusula (17), es la llave de asociación

de unidad de ejecución, el segundo argumento que recibe es el nombre del módulo puesto como parámetro en el módulo llamado en el predicado *call* y como tercer argumento recibe el nombre de una unidad de definición, el cual contiene los predicados que debe satisfacer el módulo dado como parámetro.

Lo primero que hace la cláusula (17), es buscar el predicado *mapea* dentro del programa que se está ejecutando, para establecer la correspondencia entre los predicados requeridos como parámetros en un módulo y los que la satisfacen. Esto se realiza mediante la unificación del predicado *mapea*, que llama la cláusula *agrinterf*, con aquel que exista en el programa y con la condición de que coincida en el primer y segundo argumentos, con el nombre del módulo que funge como parámetro y la unidad de definición que contiene los parámetros requeridos. Al hacer la unificación del predicado *mapea*, se instancia, mediante el funtor *map( ListMap )*, la lista de correspondencia entre los predicados requeridos como parámetros definidos en la unidad de definición llamada *DefForm*, y los predicados del módulo pasado como parámetro que las satisfacen. Esta lista de correspondencia entre predicados es guardada en la variable *ListMap*. Lo que hace después la cláusula *agrinterf*, es buscar la unidad de definición que contiene los parámetros requeridos, mediante la unificación del predicado *mod_defi* con aquel que exista en el programa y que coincida, en el primer argumento, con el nombre de la unidad de definición, que es el tercer parámetro de la cláusula *agrinterf*.
Al hacer la unificación del predicado *mod_defi*, se instancia también, mediante el funtor *pred_expo( Pred )*, los predicados, y tipos de los argumentos que debe satisfacer un módulo como parámetro. Los predicados están colocados en una lista y guardados en la variable *Pred*.

Una vez que se tienen identificados en la cláusula *agrinterf* la unidad de definición y el predicado *mapea*, para establecer la comunicación entre los módulos requeridos como parámetros de un módulo, y los que se envían como parámetros en el módulo llamado en el predicado *call*, la cláusula llama finalmente al predicado *agrclausinterf*. La cláusula que define este predicado es la siguiente:

```
agrclausinterf( LlaveMod, [ NomPredAct / AridPredAct :
                NomPredForm / AridPredForm | RestMap ],
                [ PredDef | RestPredDefs ] ) :-
   constrclaus( LlaveMod, LlaveMod, PredDef,
                ( PredForm :- CuerpForm ) ),
   PredForm =.. [ NomPredForm | ArgsForm ],
   PredActL1 =.. [ NomPredAct, LlaveMod | ArgsForm ],
   nuevcab( LlaveMod, PredForm, PredFormL1 ),
   nuevcuer( LlaveMod, CuerpForm, CuerpFormL1 ),
   assert( ( PredFormL1 :- PredActL1, CuerpFormL1 ) ),
   agrclausinterf( LlaveMod, RestMap, RestPredDefs ) . ..( 18 )
```

El objetivo de esta cláusula, es realizar algo similar al predicado *agrinterf* definido anteriormente, construye una cláusula para establecer la comunicación entre la llamada de un predicado parametrizado mediante un módulo y el que se recibe como parámetro, en el módulo llamado en el predicado *call*. El predicado *agrclausinterf*, recibe como primer argumento la clave de asociación de unidad de ejecución, como segundo argumento la lista de correspondencia entre los predicados definidos como parámetros y los predicados que los satisfacen. Y recibe como tercer argumento, el conjunto de predicados definidos en la unidad de definición que determina los requerimientos de los predicados parametrizados. Este conjunto de predicados es recibido como una lista de predicados de la siguiente forma:

```
NomPred1( Variable1 / Predicado1,
          Variable2 / Predicado2, ... ),

NomPred2( Variable1 / Predicado1,
          Variable2 / Predicado2, ... ), .....
```

en donde :

*NomPred1*, *NomPred2*, etc. son los nombres de los predicados definidos en la unidad de definción, definido en el módulo llamado en el predicado *call* como parámetro formal;

*Predicado1*, *Predicado2*, etc. son los nombres de los predicados que validan los tipos de las variables *Variable1*, *Variable2*, etc. especificados en los predicados.

La cláusula que construye *agrclausinterf*, a diferencia de la que construye también el predicado *agrinterf*, contiene como cuerpo los predicados que validan los argumentos del predicado parametrizado, y que son especificados en la unidad de definición definida como parámetro formal, del módulo al cual se llama en el predicado *call*. De acuerdo a los nombres de los argumentos que recibe la cláusula *agrclausinterf*, la cláusula que construye es la siguiente:

```
NomPredForm( LlaveMod, Argumentos ) :-
   NomPredAct( LlaveMod, Argumentos ), CuerpFormL1 . ... ( 19 )
```

en donde:

*NomPredForm* : es el nombre del predicado parametrizado definido en la lista de correspondencia y en la unidad de definición, la cual establece los requerimientos de los predicados parametrizados;

*NomPredAct* : es el nombre del predicado que contiene el cuerpo del predicado parametrizado y que está definido en la lista de correspondencia también;

*LLaveMod* : es la clave de asociación de unidad de ejecución;

*Argumentos* : es el conjunto de los argumentos que posee la cláusula y que son los mismos en ambos predicados de la cláusula *NomPredForm* y *NomPredAct*.

*CuerpFormL1* : es el conjunto de predicados que validan los argumentos del predicado parametrizado, y que están definidos en la unidad de definición que establece los requerimientos de los predicados parametrizados.

Se describirá en seguida el funcionamiento de la cláusula *agrclausinterf*. Lo primero que realiza esta cláusula, es construir una cláusula que valide cada uno de los argumentos del predicado parametrizado, es decir, del primer predicado de la lista que recibe como tercer argumento. Esto lo realiza llamando al predicado *constrclaus*, explicado anteriormente. Los argumentos que se envían son: *LlaveMod, LlaveMod, PredDef* y *( PredForm :- CuerpForm ))*, respectivamente. Los dos primeros argumentos son la clave de asociación de unidad de ejecución, el tercer argumento es el primer predicado de la lista del tercer argumento del predicado *agrclausinterf*. El predicado *constrclaus*, de acuerdo a los nombres de los argumentos de la cláusula *agrclausinterf*, regresa en el tercer argumento la siguiente cláusula:

*PredForm( Argumentos ) :- CuerpForm* ..... ( 20 )

en donde:

*PredForm* : es el nombre del predicado parametrizado;

*Argumentos* : es el conjunto de los argumentos del predicado y

*CuerpForm* : es el conjunto de predicados que validan los tipos de los argumentos del predicado parametrizado.

Lo que realiza después la cláusula *agrclausinterf* es obtener los argumentos de la cabeza de la cláusula (20). Esto lo realiza mediante la instrucción *Pred =.. [ NomPredForm | ArgsForm ]*. Después mediante la instrucción *PredActLl =..[ NomPredAct, LlaveMod | ArgsForm ]*, construye el predicado *NomPredAct( LlaveMod, Argumentos )* que es el primero que aparece en el cuerpo de la cláusula (19). Luego mediante el predicado *nuevcab( LlaveMod, PredForm, PredFormLl )*, agrega la llave de asociación de unidad de ejecución a la cabeza de la cláusula (19), para así formar el siguiente predicado:

*NomPredForm( LlaveMod, argumentos )*

que es el que regresa el predicado *nuevcab* como tercer argumento.

El siguiente predicado que llama la cláusula *agrclausinterf* es *nuevcuer( LlaveMod, CuerpForm, CuerpFormLl )*. Las cláusulas que definen este predicado son las siguientes:

```
nuevcuer( LlaveMod, ( CabCuerp, RestCuerp ),
      ( NuevCabCuerp, NuevRestCuerp ) ) :-
   nuevcuer( LlaveMod, CabCuerp, NuevCabCuerp ),
   nuevcuer( LlaveMod, RestCuerp, NuevRestCuerp ) . ... ( 21 )

nuevcuer( LlaveMod, ( X ), NuevClaus ) :-
      ifthenelse( termnoprim( X ),
                  nuevcab( LlaveMod, X, NuevClaus ),
                  NuevClaus = X ) .              ... ( 22 )
```

Este predicado tiene como fin agregar a cada uno de los predicados que recibe como una lista en el segundo argumento, la llave de asociación de unidad de ejecución que recibe como primer argumento. La llave es agregada como primer argumento, a cada predicado de esta lista. La lista de estos predicados modificados es regresada en el tercer argumento.

En la cláusula *agrclausinterf*, el predicado *nuevcuer( LlaveMod, CuerpForm, CuerpFormLl )*, recibe como segundo argumento el cuerpo de la cláusula (20) llamada *CuerpForm* y regresa como tercer argumento mediante una lista, el mismo conjunto de predicados, pero en cada predicado aparece como primer argumento la llave de asociación de unidad de ejecución. Esta lista es guardada en la variable *CuerpFormLl*, la cual constituye la otra parte del cuerpo de la cláusula (19). Finalmente, una vez formada la cláusula (19), el siguiente predicado que llama la cláusula *agrclausinterf* es *assert( ( PredLl :- PredActLl, CuerpLl ) )*, el cual la agrega a la base de datos. La cláusula *agrclausinterf* termina llamándose recursivamente para establecer de igual menera, mediante una cláusula, la comunicación con el resto de los parámetros formales y actuales del módulo llamado en el predicado *call*.

El siguiente predicado que llama la cláusula *agrmódulo* es *agrlistmod*. Este predicado tiene como fin, agregar a la base de datos, el contenido de los módulos que son puestos como parámetros en el módulo llamado en el predicado *call*. Las cláusulas que definen este predicado son las siguientes:

```
agrlistmod( LlaveMod, [ Pred / Arid | RestParm ] ) :-
   agrlistmod( LlaveMod, RestParm ) .          ..... ( 23 )

agrlistmod( LlaveMod, [ Mod | RestParm ] ) :-
   agrmódulo( LlaveMod, Mod ),
   agrlistmod( LlaveMod, RestParm ) .          ..... ( 24 )
```

La primera definición del predicado *agrlistmod* considera el caso cuando el módulo está parametrizado por un predicado, mientras que la segunda por un módulo. Ambas cláusulas reciben como primer argumento, la clave de asociación de unidad de ejecución, y como segundo argumento una lista que contiene, respectivamente para cada una de las cláusulas anteriores, los nombres de los predicados, y los módulos que fueron puestos como parámetros al módulo llamado en el predicado *call*. A continuación se describirá cada una de estas cláusulas.

Lo que hace la cláusula (23), es agregar el resto de la lista de parámetros que recibe, llamándose recursivamente, ya que no hace nada con el nombre del predicado y aridad que recibe como primer elemento de esta lista, porque supone que existe la cláusula de este predicado en el programa que se está ejecutando. Lo que hace la cláusula (24), es llamar primero al predicado *agrmódulo*, para agregar el contenido del módulo que está como primer elemento de la lista que recibe. Este predicado *agrmódulo* fue definido anteriormente. Después la cláusula *agrlistmod* agrega

también el resto de la lista de parámetros con que fue llamado el módulo en el predicado *call*, llamándose recursivamente.

El último predicado que llama la cláusula *agrmódulo*, es *agrclamod*. La cláusula que define este predicado es la siguiente:

```
agrclamod( LlaveMod, [ Claus | RestClaus ] ) :-
    agrclaus( LlaveMod, Claus ),
    agrclamod( LlaveMod, RestClaus ) .          ..... ( 25 )
```

El objetivo de este predicado, es agregar cada una de las cláusulas que componen el cuerpo de la unidad de ejecución del módulo llamado en el predicado *call*. El primer argumento que recibe la cláusula *agrclamod* es la clave de asociación de unidad de ejecución y el segundo argumento que recibe es una lista de cláusulas que componen el cuerpo de la unidad de ejecución del módulo llamado en el predicado *call*. Lo que hace la cláusula *agrclamod* es llamar primero al predicado *agrclaus*, el cual agrega a la base de datos, la primera cláusula de la lista que recibe *agrclamod*. Después la cláusula *agrclamod* agrega el resto de las cláusulas de la lista llamándose recursivamente. En seguida se describirá la cláusula que define al predicado *agrclaus*.

Antes de agregar una cláusula del cuerpo de la unidad de ejecución del módulo llamado en el predicado *call*, la cláusula *agrclaus* debe revisar con que atributos fue definida esta cláusula. Estos atributos, como se mencionó en el capítulo anterior, son: *pri*, *inv* y *ext*. Por ejemplo, si una cláusula está definida con el atributo *pri*, la cláusula *agrclaus* debe verificar en la base de datos, si existen otras definiciones de la cláusula que pretende agregar, si es así, entonces debe primero eliminarlas, y después agregar la cláusula. Pero si no existen otras definiciones de la cláusula a agregar, debe agregar la cláusula misma. De la misma forma, cuando la cláusula *agrclaus* quiera agregar una cláusula definida con el atributo *inv*, debe verificar también si existen otras definiciones de esta misma cláusula en la base de datos, si es así, entonces no agrega la cláusula que pretende agregar; pero si no existen, la agrega. Y si una cláusula está definida con el atributo *ext* o si no está definida con ningún atributo, la cláusula *agrclaus* debe agregarla directamente a la base de datos del intérprete, sin importar si existen o no otras definiciones de esta misma cláusula en la base de datos. Para cada uno de los atributos *pri*, *inv* y *ext*, existe una definición del predicado *agrclaus*.

Cuando la cláusula que se pretende agregar está definida con el atributo *pri*, la definición de la cláusula *agrclaus* es la siguiente:

```
agrclaus( LlaveMod, [ pri : ( CabClaus :- CuerClaus ) |
                        RestClaus ] ) :-
   functor( CabClaus, NomCabClaus, AridClaus ),
   AridInc is AridClaus + 1,
   functor( CabClausLl, NomCabClaus, AridInc ),
   arg( 1, CabClausLl, LlaveMod ),
   clause( CabClausLl, CuerpClausExist ),
   retract( ( CabClausLl :- CuerpClausExist ) ),
   agrclaus( LlaveMod, [ ( CabClaus :- CuerClaus ) |
            RestClaus ] ) .                    ..... ( 26 )

agrclaus( LlaveMod, [ pri : ( CabClaus :- CuerClaus ) |
          RestClaus ] ) :-
   agrclaus( LlaveMod, [ ( CabClaus :- CuerClaus ) |
            RestClaus ] ) .                    ..... ( 27 )

agrclaus( LlaveMod, [ pri : ( Hecho ) | RestClaus ] ) :-
   functor( Hecho, NomHecho, AridHecho ),
   AridIic is AridHecho + 1,
   functor( HechoLl, NomHecho, AridInc ),
   arg( 1, HechoLl, LlaveMod ),
   clause( HechoLl, true ),
   retract( ( HechoLl ) ),
   agrclaus( LlaveMod, [ ( Hecho ) | RestClaus ] ) . .. ( 28 )

agrclaus( LlaveMod, [ pri : ( Hecho ) | RestClaus ] ) :-
   agrclaus( LlaveMod, [ ( Hecho ) | RestClaus ] ) . .. ( 29 )
```

Cada una estas cláusulas recibe como primer argumento la llave de asociación de unidad de ejecución y como segundo argumento, una lista que contiene las distintas definiciones del predicado que la cláusula *agrclaus* pretende agregar a la base de datos. Las cláusulas (26) y (27) consideran el caso cuando la cláusula que se pretende agregar posee un cuerpo, mientras que las cláusulas (28) y (29) el caso cuando la cláusula es de tipo hecho. A continuación se describirá cada una de estas cláusulas.

Lo primero que hace la cláusula (26), es verificar si existen en la base de datos del intérprete, otras definiciones de la primera cláusula de la lista que recibe como segundo argumento.

Esta cláusula es la siguiente:

*CabClaus :- CuerClaus* ..... ( 30 )

Si existen otras definiciones de la cláusula (30) en la base de datos, es porque fueron agregadas antes cuando se agregó el cuerpo de los módulos importados o recibidos como parámetros, del módulo que pretende agregar la cláusula (30). Por consiguiente, cada una de las definiciones que existe en la base de datos de la cláusula (30) tiene, tanto en la cabeza como en los predicados que conforman el cuerpo de la cláusula, como primer argumento, la llave de asociación de unidad de ejecución. Por lo tanto, antes de verificar la cláusula *agrclaus* si existen otras definiciones de la cláusula (30) que pretende agregar, debe incorporar a la cabeza de esta cláusula, la llave de asociación de unidad de ejecución como primer argumento. Esto se hace para que al buscar en la base de datos otras definiciones mediante la cabeza de la cláusula que se pretende agregar, coincidan exactamente en nombre y número de argumentos. Los primeros cuatro predicados del cuerpo de la cláusula (26), se encargan de agregar a la cabeza de la cláusula (30), la llave de asociación de unidad de ejecución como primer argumento. El primer predicado *functor( CabClaus, NomCabClaus, AridClaus )* obtiene la aridad de la cabeza de la cláusula (30). La siguiente instrucción incrementa en uno este valor, ya que va a ser agregado un nuevo argumento a la cabeza de la cláusula (30), que es la llave de asociación de unidad de ejecución. Después, la siguiente instrucción *functor( CabClaus, NomCabClaus, AridClaus )*, construye un funtor llamado *CabClausLl*, cuyo nombre es el mismo que la cabeza de la cláusula (30).

Este funtor tiene un argumento más que la cabeza de la cláusula (30), que es en donde se guardará la llave de asociación de unidad de ejecución. Luego mediante la instrucción *arg( 1, CabClausLl, LlaveMod )*, se agrega al funtor *CabClausLl* la llave de asociación de unidad de ejecución como primer argumento.

Una vez construido un funtor similar al de la cabeza de la cláusula (30), pero con un argumento más, que es la llave de asociación de unidad de ejecución, la cláusula (26) llama al siguiente predicado que es *clause( CabClausLl,CuerpClausExist )*. Este predicado está incorporado en el sistema del intérprete de *Arity Plolog* y verifica si existe una cláusula en la base de datos cuya cabeza unifique con el funtor que recibe como primer argumento dicho predicado. Si es así, el predicado *clause* regresa el valor verdadero y el cuerpo de la cláusula cuya cabeza unificó, lo regresa en el segundo argumento. Pero si no existe en la base de datos una cláusula tal, *clause* regresa el valor falso. En caso de que el predicado *clause* tenga éxito cuando se llama en la cláusula (26), la siguiente instrucción, *retract(( CabClausLl :- CuerpClausExist ) )*, se encarga de eliminar la cláusula existente

en la base de datos cuya definición de la cabeza de la cláusula (30) es otra.

Una vez eliminadas de la base de datos las definiciones existentes de la cláusula (30), el último predicado que llama la cláusula (26) es *agrclaus*. Este predicado se llama de nuevo con los mismos argumentos que la cabeza de la cláusula (26), pero sin el atributo con el que fue definida la cláusula que se pretende agregar. La definición de la cláusula *agrclaus* se explicará más adelante. Sin embargo, si la cláusula *claus* regresa el valor falso, significa que no existen en la base de datos otras definiciones de la cláusula (30) y entonces la cláusula (26) falla. Cuando esto sucede, el intérprete intenta agregar la cláusula (30), llamando a la siguiente definición del predicado *agrclaus*, que es la cláusula (27). Lo que realiza esta cláusula, es llamar a *agrclaus* de la misma forma que en la cláususula (26).

Las cláusulas (28) y (29), consideran el caso cuando la cláusula que hay que agregar del cuerpo de un módulo, es de tipo hecho. El funcionamiento de estas cláusulas es muy similar al de las cláusulas (26) y (27) y por ello no se describirán.

La definición de la cláusula *agrclaus*, considerando ahora que la cláusula que pretende agregar está definida con el atributo *inv*, es la siguiente:

```
agrclaus( LlaveMod, [ inv : ( CabClaus :- CuerClaus ) |
                        RestClaus ] ) :-
   functor( CabClaus, NomCabClaus, AridCabClaus ),
   AridInc is AridCabClaus + 1,
   functor( CabClausLl, NomCabClaus, AridInc ),
   arg( 1, CabClausLl, LlaveMod ),
   clause( CabClausLl, CuerpClausExist ) .      ..... ( 31 )

agrclaus( LlaveMod, [ inv : ( CabClaus :- CuerClaus ) |
                        RestClaus ] ) :-
   agrclaus( LlaveMod, [ ( CabClaus :- CuerClaus ) |
                         RestClaus ] ) .        ..... ( 32 )

agrclaus( LlaveMod, [ inv : ( Hecho ) | RestClaus ] ) :-
   functor( Hecho, NomHecho, AridHecho ),
   AridInc is AridHecho + 1,
   functor( HechoLl, NomHecho, AridInc ),
   arg( 1, CabClausLl, LlaveMod ),
   clause( HechoLl, true ) .                    ..... ( 33 )
```

```
agrclaus( LlaveMod, [ inv : ( Hecho ) | RestClaus ] ) :-
    agrclaus( LlaveMod, [ ( Hecho ) | RestClaus ] ) . . ( 34 )
```

Cada una estas cláusulas recibe como siempre en el primer argumento, la llave de asociación de unidad de ejecución y en el segundo, una lista que contiene las distintas definiciones de un mismo predicado que la cláusula *agrclaus* pretende agregar a la base de datos. Nuevamente, las cláusulas (31) y (32) consideran el caso cuando la cláusula que se pretende agregar tiene cuerpo, mientras que las cláusulas (33) y (34) cuando la cláusula es de tipo hecho. El funcionamiento de las cláusulas (31) y (32) es similar al de las cláusulas (26) y (27), exepto que si al llamar en la cláusula (31) al predicado *clause( CabClausLl, CuerpClausExist )*, regresa el valor verdadero, entonces significa que existen otras definiciones en la base de datos de la cláusula que pretenden agregar. Por lo tanto, ya no hay que eliminarlas ni agregar la cláusula que pretende. En caso de que en la cláusula (31), al llamar al predicado *clause* falle, lo cual significa que no existen otras definiciones de la cláusula que pretende agregar esta cláusula, las cláusulas (32) se encarga de agregarlo.

El funcionamiento de las cláusulas (33) y (34) es también muy similar al de las cláusulas (31) y (32), ya que la cláusula que pretenden agregar es de tipo hecho. Por lo que tampoco se describirán.

La definición de la cláusula *agrclaus*, considerando finalmente que la cláusula que pretende agregar está definidad con el atributo *ext*, es la siguiente:

```
agrclaus( LlaveMod, [ ext : ( CabClaus :- CuerClaus ) |
                        RestClaus ] ) :-
    nuevcab( LlaveMod, CabClaus, CabClausLl ),
    nuevcuer( LlaveMod, CuerClaus, CuerClausLl ),
    assert( ( CabClausLl :- NuevCuerClausLl ) ),
    agrclaus( LlaveMod, RestClaus ) .          ..... ( 35 )

agrclaus( LlaveMod, [ ext : ( Hecho ) | RestClaus ] ) :-
    nuevcab( LlaveMod, Hecho, HechoLl ),
    assert( HechoLl ),
    agrclaus( LlaveMod, RestClaus ) .          ..... ( 36 )
```

```
agrclaus( LlaveMod, [ ( CabClaus :- CuerClaus ) |
                      RestClaus ] ) :-
   nuevcab( LlaveMod, CabClaus, CabClausLl ),
   nuevcuer( LlaveMod, CuerClaus, CuerClausLl ),
   assert( ( CabClausLl :- CuerClausLl ) ),
   agrclaus( LlaveMod, RestClaus ) .          ..... ( 37 )

agrclaus( LlaveMod, [ ( Hecho ) | RestClaus ] ) :-
   nuevcab( LlaveMod, Hecho, HechoLl ),
   assert( HechoLl ),
   agrclaus( LlaveMod, RestClaus ) .          ..... ( 38 )
```

Nuevamente cada una de estas cláusulas recibe como primer argumento, la llave de asociación de unidad de ejecución y como segundo argumento, una lista que contiene las distintas definiciones del predicado que la cláusula *agrclaus* pretende agregar a la base de datos. Las cláusulas (35) y (37) consideran el caso cuando la cláusula que se pretende agregar tiene cuerpo.
La diferencia es que la cláusula (37) es usada cuando la cláusula que se pretende agregar no está definida con ningun atributo, es decir, se usó el atributo por omisión que es *ext*. Las cláusulas (37) y (38) son similares a las cláusulas (35) y (36), respectivamante, exepto que consideran el caso cuando la cláusula que pretende agregar es de tipo hecho. A continuación se describirá el funcionamiento de cada una de estas cláusulas.

La cláusula (35) tiene como objetivo agregar a la base de datos la primera cláusula de la lista que recibe como segundo argumento, sin importar si existen o no otras definiciones de esta misma cláusula en la base de datos. Lo primero que hace la cláusula (35) es llamar al predicado *nuevcab( LlaveMod, CabClaus, CabClausLl )*, el cual agrega como primer argumento, a la cabeza de la cláusula que se pretende agregar, la llave de asociación de unidad de ejecución. Se obtiene así un nuevo predicado llamado *CabClausLl*, que es el que regresa el predicado *nuevcab* como tercer argumento. El siguiente predicado que llama la cláusula (35) es *nuevcuer( LlaveMod, CuerClaus, CuerClausLl )*, el cual agrega como primer argumento a cada uno de los predicados que conforman el cuerpo de la cláusula que se pretende agregar, la llave de asociación de unidad de ejecución. Se obtiene así un nuevo conjunto de predicados, que es el nuevo cuerpo de la cláusula a agregar, que es llamado *CuerClausLl*, y que es el que regresa el predicado *nuevcuer* como tercer argumento. El siguiente predicado que llama la cláusula (35) es *assert( ( CabClausLl :- NuevCuerClausLl ))*, el cual forma una cláusula cuya cabeza es el predicado *CabClausLl* y cuyo cuerpo es el conjunto de predicados representados por la variable *NuevCuerp* y lo agrega a la base de datos del intérprete. Finalmente la cláusula (35) agrega el resto de los predicados contenidos en la lista que recibe como segundo argumento, llamándose recursivamnete. El cuerpo de la cláusula

(37) es igual al de la cláusula (35), por lo tanto su funcionamiento es el mismo.

El funcionamiento de la cláusula (36) es similar al de la cláusula (35), exepto que la cláusula que pretende agregar no posee cuerpo. También el cuerpo de la cláusula (38) es el mismo que el de la cláusula (37) y por lo tanto su funcionamiento es el mismo.

Finalmente, el último predicado que llama el cuerpo de la cláusula *call* es *calls( LlaveDefi, Claus )*. Este predicado tiene como objetivo llamar a cada una de las metas que recibe como segundo argumento la cláusula *call*.

Las cláusulas que componen la definición del predicado *calls*, son las siguientes:

```
calls( LlaveDef, ( Claus, RestClaus ) ) :-
   nuevcab( LlaveDef, Claus, NuevClaus ),
   NuevClaus, calls( LlaveDef, RestClaus ) .   ..... ( 39 )

calls( LlaveDef, ( Claus ) ) :-
   nuevcab( LlaveDef, Claus, NuevClaus ),
   NuevClaus .                                 ..... ( 40 )

calls( LlaveDef, ( ( call( ExprMod, Claus ) ),
        RestClaus ) ) :-
   call( ExprMod, Claus ),
   calls( LlaveDef, RestClaus ) .              ..... ( 41 )

calls( LlaveDef, ( ( call( ExprMod, Claus ) ) ) ) :-
   call( ExprMod, Claus ) .                    ..... ( 42 )
```

Cada una de estas cláusulas recibe como primer argumento la llave de asociación de unidad de definición y como segundo argumento, una lista de metas a probar. A continuación se describirá cada una de estas cláusulas.

El primer predicado que llama la cláusula (39) es *nuevcab( LlaveDef, Claus, NuevClaus )*. Este predicado tiene como fin agregar como primer argumento a la primera meta de la lista que recibe la cláusula (39), la llave de asociación de unidad de definición. Esto se hace porque como la unidad de definición de un mó-

dulo, contiene las cláusulas que pueden ser llamadas exteriormente, en este caso por la lista de metas que tiene como segundo argumento la cláusula *calls*, al ser agregada cada una de las cláusulas que conforman dicha unidad a la base de datos, también a cada uno de ellas se les agregó, como primer argumento, la llave de asociación de unidad de definición. La meta modificada por el predicado *nuevcab* es devuelta como tercer argumento y es llamada *NuevClaus*. Una vez agregada la clave de asociación de unidad de definición a la primera meta de la lista que recibe como segundo argumento la cláusula (39), esta cláusula llama a esta misma meta, colocando su nombre, que es *NuevClaus* como siguiente predicado.

Finalmente la cláusula (39) llama a las metas que estan definidas en el resto de la lista que recibe como segundo argumento, llamándose recursivamente.

La cláusula (40) es similar a la cláusula (39), exepto que considera el caso cuando la lista de metas a probar posee solo una meta. La cláusula (41) considera el caso cuando una de las metas de la lista que recibe como segundo argumento, es el predicado *call*, donde el primer argumento es una expresión modular, es decir, una llamada recursiva de la cláusula (1), la cual fue llamada en el segundo argumento del predicado *calls*. El primer predicado que llama la cláusula (41) es el primer elemento de la lista de metas a probar que recibe como segundo argumento, es decir, *call( ExprMod, Claus )*. Luego el cuerpo de la cláusula (41) llama a las metas que están definidas en el resto de la lista que recibe como segundo argumento, llamándose recursivamente. La cláusula (42) es similar a la cláusula (41), solo que considera el caso cuando la lista de metas a probar posee solo una meta que es *call*, donde el primer argumento es una expresión modular.

# C O N C L U S I O N E S .

La incorporación de un sistema de módulos al lenguaje *PROLOG* tiene una serie de ventajas como son el diseño de programas estructurados; por otro lado hace posible la reutilización de programas, regula la visibilidad y la existencia de cláusulas y hechos, y facilita la comunicación entre segmentos de código.

El prototipo de sistemas de módulos desarrollado en este trabajo, además de las características antes mencionadas, tiene otras importantes entre las que destacan las facilidades de metaprogramación y uso del sistema modular en sí, que no se manejan regularmente en la mayoría de los sistemas modulares en *PROLOG*.

En cuanto a la metaprogramación, el sistema de módulos proporciona algunos predicados que permiten alterar el contenido de un módulo, agregando o eliminando cláusulas en tiempo de ejecución. Además, por la manera en que están definidos los módulos, es fácil definir otras cláusulas y predicados que permitan alterar los parámetros y los módulos que se importan, así como agregar o quitar unidades o módulos completos, en tiempo de ejecución.

En lo que se refiere a la facilidad de uso del sistema de módulos, éste se diseñó sin necesidad de incorporar símbolos o estructuras ajenas al lenguaje, que pudieran hacerlo difícil de usar o entender.

En lo que toca a la implantación del prototipo, ésta fue conceptualmente fácil pero laboriosa en su programación, ya que en esta primera versión se buscó utilizar el prototipo para analizar la viabilidad del esquema planteado, más que desarrollar una versión eficiente del modelo planteado.

En cuanto al desempeño y funcionamiento del prototipo, éste se probó con múltiples ejemplos, aparte de los aquí descritos, los cuales muestran que la implantación trabaja bien y de manera

rápida en la generación de la base de datos necesaria, para que el intérprete de *PROLOG* procese cada una de las cláusulas y términos que componen una expresión modular.

El modelo aquí presentado deja trazadas al menos dos líneas potenciales de continuidad. Una muy importante es la justificación lógica del modelo planteado, en la que hasta ahora no se ha trabajado, pero que será necesario abordar si el esquema quiere insertarse dentro del diseño de un lenguaje lógico. La segunda es el trabajo de adecuación de la parte modular dentro del diseño general de un lenguaje con tipos y objetos. Ambas líneas de trabajo serán contempladas dentro del proyecto de investigación del que forma parte este trabajo.

# A P E N D I C E  A

Tabla comparativa de algunos conceptos de módulos en los lenguajes *ADA*, *ML* y *OBJ3*. Una equis indica que el lenguaje no tiene definido el concepto correspondiente, un punto indica que sí lo tiene.

| Lenguaje / Concepto | ADA | ML | OBJ3 |
|---|---|---|---|
| Componentes del Sistema de Módulos | *Especificación de paquete* (Parte Declarativa)<br>*Cuerpo de Paquete* (Parte Ejecutable) | *Signatura* (Parte Declarativa)<br>*Estructura* (Parte Ejecutable)<br>*Functor* (Estructura Parametrizada) | *Objeto* (Parte Declarativa y Ejecutable)<br>*Teoría* (Requisitos de los Parámetros)<br>*Vistas* (Instanciación de los Módulos Parametrizados) |
| Vistas | X | X | |

| Lenguaje / Concepto | ADA | ML | OBJ3 |
|---|---|---|---|
| Implantaciones Múltiples | · | · | X |
| Módulos Parametrizados | · | · | · |
| Composición de Módulos | X | X | · |
| Declaraciones Parametrizadas | X | X | X |
| Módulos como Tipos y Unidad que la representa | X | · *Signaturas* | · · *Teorías* |
| Importación de Módulos | · | · | · |
| Subdeclaraciones | X | X | X |
| Abrir un Módulo | · | · | X |

| Lenguaje / Concepto | ADA | ML | OBJ3 |
|---|---|---|---|
| Construcción de Bibliotecas | · | · | · |
| Modificación Dinámica de Módulos | X | X | X |

Tabla comparativa de algunos conceptos de módulos en algunos lenguajes lógicos. Una equis indica que el lenguaje no tiene definido el concepto correspondiente, un punto indica que si lo tiene.

| Lenguaje / Concepto | Programación Lógica Contextual | ML-PROLOG | BIM-PROLOG | SEPIA |
|---|---|---|---|---|
| Componentes del Sistema de Módulos | *Unidades* Agrupa Cláusulas y Predicados | *Signatura* ( Parte Declarativa )<br><br>*Estructura* (Parte Ejecutable)<br><br>*Functor* (Estructura Parametrizada) | Directiva *:- module* | Directiva *:- module* |
| Vistas | X | X | X | X |
| Implantaciones Múltiples | X | . | X | X |

| Lenguaje / Concepto | Programación Lógica Contextual | ML-PROLOG | BIM-PROLOG | SEPIA |
|---|---|---|---|---|
| Módulos Parametrizados | X | · | · | · |
| Composición de Módulos | X | X | X | X |
| Declaraciones Parametrizadas | X | X | X | X |
| Módulos como Tipos y Unidad que la representa | X | *Signatura* | X | X |
| Importación de Módulos | · | · | · | · |
| Subdeclaraciones | X | X | X | X |
| Abrir un Módulo | X | · | · | · |
| Construcción de Bibliotecas | · | · | · | · |

| Lenguaje / Concepto | **Programación Lógica Contextual** | ML-PROLOG | BIM-PROLOG | SEPIA |
|---|---|---|---|---|
| Modificación Dinámica de Módulos | X | X | X | · |
| Razonamiento Contextual | · | X | X | X |

# A P E N D I C E C

Tabla comparativa de algunos conceptos de módulos en algunos lenguajes lógicos y el prototipo diseñado, marcado con un asterisco. Una equis indica que el lenguaje no tiene definido el concepto correspondiente, un punto indica que si lo tiene.

| Lenguaje / Concepto | **Programación Lógica Contextual** | ML-PROLOG | BIM-PROLOG | SEPIA | * |
|---|---|---|---|---|---|
| Componentes del Sistema de Módulos | *Unidades* Agrupa Cláusulas y Predicados | *Signatura* (Parte Declarativa)<br><br>*Estructura* (Parte Ejecutable)<br><br>*Functor* (Estructura Parametrizada) | Directiva *:- module* | Directiva *:- module* | *Unidad de Definicion* (Parte Declarativa)<br><br>*Unidad de Ejecución* (Parte Ejecutable)<br><br>*Unidad Mapea (vista)* |

| Lenguaje / Concepto | **Programación Lógica Contextual** | ML-PROLOG | BIM-PROLOG | SEPIA | * |
|---|---|---|---|---|---|
| Vistas | X | X | X | X | · |
| Implantaciones Múltiples | X | · | X | X | · |
| Módulos Parametrizados | X | · | · | · | · |
| Composición de Módulos | X | X | X | X | · |
| Declaraciones Parametrizadas | X | X | X | X | X* |
| Módulos como Tipos y Unidad que la representa | X | *Signatura* | X | X | · *Unidad de Definición* |
| Importación de Módulos | · | · | · | · | · |
| Subdeclaraciones | X | X | X | X | X |

* La versión actual del prototipo tiene unidades de declaración parametrizadas, aunque no fueron explicadas en el capítulo tercero.

| Lenguaje / Concepto | **Programación Lógica Contextual** | **ML-PROLOG** | **BIM-PROLOG** | **SEPIA** | * |
|---|---|---|---|---|---|
| Abrir un Módulo | X | · | · | · | · |
| Construcción de Bibliotecas | · | · | · | · | · |
| Modificación Dinámica de Módulos | X | X | X | · | · |
| Razonamiento Contextual | · | X | X | X | · |

# B I B L I O G R A F I A .


[Ada 83] Reference Manual for the Ada Programming Language . ANSI/MIL-STD-1815 A . 1983 .

[AG 88] AG Siemens . Siemens-Prolog Manual V1.0A. Junio 1988 .

[Arity 91] Arity Corporation 1989-1991 . 30 Bomino Drive . Concord, Massachusetts 01742 . *Using The Arity/PROLOG Compiler and Interpreter* . *The Arity/PROLOG Language Reference Manual* .

[BIM 89] BIM SA/NV . BIM-Prolog Manual 2.4 . Marzo 1989 .

[Bowen 81] Bowen, D. L. . Dec-10 Prolog User's Manual . *Occasional Paper 27, University of Edinburgh, Department of Artifical Intelligence* . Diciembre 1981 .

[Dahl 66] Dahl, Ole-Johan y Nygaard, Kristen . SIMULA-An ALGOL-Based Simulation Language . *Communications of the ACM* . Septiembre 1966 . Vol 9 . Num 9 . pags. 671-678 .

[Dorochevsky 89] Dorochevsky, Michel Meier, Micha y Pérez Bruno . SEPIA Module System Specification Report . *Technical Report IR-LP-13-20, ECRC* . Diciembre 1989 .

[Dorochevsky 91] Dorochewsky, Michel. Key Features Of A PROLOG Module System . *Technical Report DPS-103 ECRC* . Marzo 1991 .

[Drossopoulou 87a] Drossopoulou, S. .
About Parametrization Of Interfaces Another Solutions To The Witness Problem .
*Imperial College Research Report* .
IC/FPR/PROG/2.2/10 .

[Drossopoulou 87b] Drossopoulou, S. .
Proposal Four For A Module System .
*Imperial College Research Report* .
IC/FPR/PROG/2.2/9 .

[Drossopoulou 88] Drossopoulou, S. Eisenbach, S. y McLoughlin L. .
Module and Type System : a Tour .
Internal Report . Febrero 1988 .
*Imperial College Research Report* .
DOC 88/13 .

[Goguen 88] Goguen, Joseph A. y Winkler Timothy .
Introducing OBJ3 .
*SRI INTERNATIONAL, Computer Science Laboratory* . Agosto 1988 .
SRI-CSL-88-9 .

[Harper 86] Harper, R., MacQueen, D. B. y Milner, R. .
Standard ML .
*Laboratory for Foundations of Computer Science, University of Edinburgh* .
ECS-LFCS-86-2 .

[Mello 89] Mello, Paola y Natalli, Antonio.
Logic Pogramming in a Software Engineering Perspective .
*North American Conference on Logic Programming* .
1989 . MIT Press . Ed. E. Luck y R. Overbeek.

[Miller 86] Miller, Dale .
A Theory Of Modules For Logic Programming .
*Procedings of the 1986 Symposium on Logic Programming* . Septiembre 1986 .
pags. 106-114 .

[Miller 89] Miller, Dale .
A Logic Analisys of Modules in Logic Programming .
*Journal of Logic Programming* . 1989 .
pags. 79-108 .

[Monteiro 89] Monteiro, Luis y Porto, Antonio .
Contextual Logic Programming .
*Procedings of the Sixth Conference on Logic Programming* . Lisboa 1989 .
Ed G. Levi y M. Martell .

[O' Kee 85] O'Kee, Richard A. .
Towards an algebra for constructing Logic Programs .
*Procedings of the IEEE Symposium on Logic Pogramming* . 1985 . Boston .

[Quintus 87] Quintus Computer Systems, Inc. .
Quintus-Prolog Reference Manual 2.0 .
Marzo 1987 .

[Sanella 92] D.T. Sanella y Wallen, L. A. .
A Calculus For The Construction Of Modular PROLOG Programs .
*Journal of Logic Programming* . 1992 .
num. 12 pags. 147-177 .

[Steele 84] Steele, Guy L. .
Common Lisp: The Language .
*Digital Press* . 1984 .

[Wirth 83] Wirth, Nicklaus .
Programming in Modula-2 .
*Springer Verlag* . 1983 .