CONSIDERACIONES SOBRE LA CONSTRUCCIÓN DE SISTEMAS EXPERTOS UTILIZANDO EL LENGUAJE PROLOG

*MSc .Lic. Mateo Lezcano Brito**
*Dr. Giraldo Valdés Pardo***

Palabras claves: Prolog, Inteligencia Artificial, Sistemas Expertos.

RESUMEN

Una base de conocimiento puede considerarse como un conjunto de reglas que definen relaciones entre objetos; por esa razón se puede utilizar la máquina de inferencia interna que posee Prolog para el desarrollo de sistemas expertos.

Con el objetivo de ayudar a resolver esta problemática, en el presente trabajo se exponen:

- un enfoque general sobre la escritura de sistemas expertos en Prolog.
- un conjunto de predicados Prolog que pueden usarse en apoyo a este enfoque.

1. INTRODUCCIÓN

Prolog fue creado como un lenguaje de programación para manipular objetos y relaciones entre éstos y se clasifica como un lenguaje de programación lógica, debido a que se basa en la prueba de teoremas a partir de una base de datos interna formada por reglas escritas en la forma de cláusulas de Horn, donde se aplica el principio de resolución y de unificación.

El desarrollo de sistemas expertos ha cobrado un gran auge en los últimos años y están disponibles gran número de máquinas de inferencia que, utilizando formas diferentes de representación del conocimiento y diversas estrategias de razonamiento, tratan de satisfacer el objetivo común de crear productos de inteligencia artificial que se acerquen cada vez más a la forma de proceder de los expertos humanos.

Una base de conocimiento se puede considerar como un conjunto de reglas que definen relaciones entre objetos y por esta razón resulta natural pensar en Prolog como una máquina de inferencia apropiada para el desarrollo de sistemas expertos. No obstante, escribir sistemas de este tipo para una máquina de inferencia convencional resulta una tarea relativamente fácil, mientras que escribir las mismas reglas para Prolog no resulta, en general, todo lo fácil que desearían los ingenieros del conocimiento, debido principalmente a que ellos no están interesados en la programación en sí, sino en el hecho de «descubrir», dentro del universo intelectual de los expertos humanos, todas las reglas no escritas que ellos han logrado establecer a través de muchos años de trabajo, de experiencias vividas, de

* Facultad de Matemática Física y Computación,
** Facultad de Ingeniería Eléctrica, Universidad Central de Las Villas, Santa Clara, Cuba.

fracasos, etc. para finalmente poder obtener una aplicación en un tiempo razonable. Sin duda la utilización de un lenguaje de alto nivel (Prolog) brinda al programador una cantidad de facilidades que no están disponibles en ninguna máquina de inferencia convencional, ya que ellas sólo son poderosas en relación con su mecanismo interno de razonamiento y no en cuanto a las herramientas de programación.

En la carrera de Ciencias de la Computación y en general en las carreras relacionadas con la Informática, se estudia el paradigma de la Programación Lógica y el desarrollo de sistemas expertos con diversas máquinas de inferencia. Muchas veces, debido al tiempo limitado del curso, los estudiantes no tienen oportunidad de poner a punto sistemas reales y cuando abordan un problema complejo no utilizan el Prolog, debido a que el tiempo de puesta a punto es mayor por tener que programar, mientras que con una máquina de inferencia sólo tienen que definir las bases de conocimiento de acuerdo con una sintaxis específica. Las ideas que se proponen en el presente trabajo se orientan a facilitar la construcción de sistemas expertos en Prolog y están siendo aplicadas en las clases de Licenciatura en Ciencias de la Computación y en la Maestría en Computación Aplicada de la Universidad Central de Las Villas en Cuba.

2. ¿POR QUÉ PROLOG?

Prolog es un lenguaje "declarativo" basado en la Lógica, lo que significa que la tarea del programador es describir o declarar el modelo del problema a resolver y la máquina de inferencia se encarga de buscar la solución. Esta afirmación es particularmente importante para el desarrollo de sistemas expertos, ya que el ingeniero de conocimiento se debe concentrar en "qué hacer" y no en "cómo hacerlo". De esta manera, el programa resultante estará más cerca del problema a ser resuelto, será más fácil adaptarlo a los cambios naturales que sufre el conocimiento y por tanto más adecuado para su mantenimiento.

Prolog es un lenguaje expresivo, que contrasta con los "lenguajes imperativos" y contiene un conjunto de facilidades avanzadas que incluyen, además de su alto nivel declarativo, el manejo automático de la memoria, la compilación incremental y la metaprogramación.

Prolog es una buena herramienta para realizar "prototipos incrementales", en los que el código es gradualmente construido a partir de sucesivas abstracciones en un período de tiempo dado.

Los programas Prolog tienen un nivel natural de abstracción y son independientes de los datos, como los lenguajes orientados a objeto.

Predicados como *assert* y *retract* permiten que los programas crezcan dinámicamente, lo cual está estrechamente relacionado con las técnicas modernas de compilación incremental.

Los programas Prolog pueden considerarse como un conjunto de datos y variables que pueden ser instanciadas en tiempo de corrida, no solo con átomos, enteros, cadenas, etc. sino también con cláusulas, ya que Prolog soporta la metaprogramación, o sea programas que manipulan a otros programas.

Sus facilidades incluyen la interfaz con otros lenguajes, la gestión de bases de datos elaboradas con diversos paquetes profesionales y más recientemente, el soporte de la programación orientada a objeto.

3. PROPUESTAS PARA ESCRIBIR SISTEMAS EXPERTOS EN PROLOG

Para diseñar y elaborar sistemas expertos mediante el enfoque que se plantea, se necesita un conjunto de predicados que aparecen en el libro Prolog y Sistemas Expertos [Lezcano, 1996]. Los programas allí descritos están listos para correr y emplean la versión 6.0 de Arity/Prolog. Se elaboraron en una forma lo suficientemente general como para permitir su uso con cualquier otra versión de Prolog con sólo pequeños cambios.

Las limitaciones de espacio no permiten describir aquí cada una de las soluciones propuestas, las cuales pueden consultarse en el libro mencionado; allí se desarrolla una metodología completa para escribir sistemas expertos en Prolog.

El método se orienta a la elaboración de sistemas que utilicen la búsqueda "dirigida por objetivos" (*backward chainning*), aunque es posible realizar pequeñas inferencias donde se utilice una estrategia "dirigida por datos" (*forward chainning*).

Seguidamente se exponen de manera resumida los pasos fundamentales del método, empleando la sintaxis de Edimburgo y predicados internos que son comunes a la definición estándar del Prolog.

- Dividir el conocimiento en partes lógicas.

  Muchas veces los Ingenieros del Conocimiento no ven una división clara del conocimiento que tienen que representar; sin embargo, aunque éste parezca "indivisible", realmente siempre es posible agruparlo de acuerdo con algún criterio. Esta división permite discriminar la búsqueda, lo que dará mayor velocidad al proceso de inferencia. Cada parte lógica formará una base de conocimiento.

- Utilizar reglas de producción para representar el conocimiento.
- Tratar en forma especial los atributos que tengan preguntas asociadas.

Por lo general las máquinas de inferencia (shell) poseen un *mecanismo interno* que controla la interfaz de *preguntas - respuestas*. El Ingeniero del Conocimiento no tiene que preocuparse por controlar si una pregunta dada ya fue formulada (el mismo atributo puede estar en varias reglas). Prolog no posee ese mecanismo, ya que él no es una simple máquina de inferencia y por tanto no está preparado para una acción tan específica. Por este motivo la interfaz y el control sobre las preguntas es responsabilidad del programador.

La interfaz debe construirse utilizando las facilidades de cajas de diálogo que brindan las versiones actuales de Prolog. El programador debe definir una caja de diálogo para cada atributo con pregunta asociada. El nombre del atributo y el nombre de la caja de diálogo asociada deben coincidir.

Las cajas de diálogo nunca deben invocarse directamente; esto se hará a través de un predicado como el que se presenta a continuación:

```
dialogar(Atributo, Valor) :-
        Atributo(Temporal),
        !,
        Valor = Temporal.
dialogar(Atributo, Valor) :-
        presenta_dialogo(Atributo),!,
        Atributo(Valor).
```

El primer argumento de *dialogar* es el nombre de un atributo; el segundo argumento es el valor que debe tener el atributo en una regla dada para que se cumpla una condición específica.

La línea *Atributo(Temporal)* de la primera cláusula de *dialogar* provoca que se ejecute el predicado pasado en la variable *Atributo*. La llamada tendrá éxito si existe ese predicado, lo que significa que el atributo ya ha sido preguntado y se ha agregado con un predicado tipo *assert* a la memoria de trabajo; en ese caso la variable libre *Temporal* se instancia con el valor que tiene asociado ese atributo. La línea *Valor = Temporal* tiene un doble objetivo; si la variable *Valor* llega libre, se instancia con el contenido de *Temporal*, si llega instanciada se produce una comparación entre la instancia de *Valor* y la instancia de *Temporal;* si la comparación tiene éxito, entonces *dialogar* también tendrá éxito; si no es así, falla y el corte que le precede le prohibe buscar soluciones alternativas.

La segunda cláusula de *dialogar* sólo se ejecuta cuando falla la primera línea de la primera cláusula, lo que significa que la pregunta no ha sido formulada. Entonces se invoca un predicado *presenta_dialogo* para que ejecute la caja de diálogo. El programador debe usar un predicado tipo *assert* como parte del predicado *presenta_dialogo* para agregar la respuesta del usuario a la memoria de trabajo.

- Establecer puntos de entrada a las bases de conocimientos.

Los sistemas estarán formados por varias bases de conocimientos (a no ser que sea muy limitado el conocimiento a expresar) las que residirán en diferentes archivos. Por esa razón cada base de conocimiento tendrá un punto de entrada o predicado principal, que será el encargado de iniciar el proceso de inferencia dentro de la base en particular.

Conceptualmente, las bases se pueden clasificar en *principales* y *secundarias*. Las bases principales son las que el usuario domina, debido a que se conocen determinadas características que permiten acceder a ellas directamente. Por otra parte, a las bases secundarias no se accede por ninguna acción directa del usuario y sólo se llega a ellas a través del proceso de inferencia.

Desde el punto de vista del usuario del sistema que desee consultar una base principal, el punto de entrada será alguna opción del menú. Para el programador, ese punto de entrada será invocado desde un predicado *case* o similar, en el módulo principal del sistema.

- Formato para el punto de entrada de una base dada.

Cada base es un archivo independiente; el proceso de enlace se encargará de agruparlo todo en un archivo ejecutable para el caso que se disponga del compilador. Si sólo se dispone de un intérprete, los predicados de consulta permiten cargar las bases cuando sea necesario. El punto de entrada debe programarse como una secuencia *repeat..fail* como la que se presenta a continuación:

```
comenzar :-
    write($ Base de conocimientos de ... $),
    repeat,
        [!
        % Llamada a los predicados deseados
    borrar_todo(Datos)
    !],
    dialogar(fin, 'No').
```

En el ciclo anterior se deben destacar los siguientes aspectos:

Comenzar es el predicado que se invoca desde el módulo principal del sistema (que reside en otro archivo) por lo que debe

declararse público. El predicado contiene un ciclo *repeat..fail* que comienza con el predicado *repeat* y finaliza con la invocación al predicado *dialogar;* el segmento interno al ciclo aparece controlado por una *tijera* [!...!]. Esta estructura de control no está disponible en todas las implementaciones Prolog. Se pueden usar otras técnicas para obtener el resultado deseado, que es un control del regreso atrás (*backtracking*) que permite "saltar", en ese retroceso, todo el código que está comprendido en ella.

El comentario que se presenta (encabezado por %) debe sustituirse por la invocación a los predicados que se deseen ejecutar. Estas llamadas funcionarán como una orden "probar un atributo".

El predicado *borrar_todo(Datos)* es obligatorio, ya que se supone que los hechos resultantes de un proceso de inferencia se van almacenando en la memoria de trabajo y esos hechos no tendrán influencia en sucesivas consultas. Por ese motivo se debe programar un predicado *borrar_todo* que se encargue, haciendo uso de los predicados de tipo *retract*, de "limpiar" la memoria de trabajo.

El *efecto de fallo* se obtiene con la inclusión al final del predicado *comenzar* de la línea *dialogar(fin,No).* En este caso se invoca al predicado *dialogar* con su primer argumento (*fin*) que debe ser una caja de diálogo. Según la idea presentada, *dialogar* espera en su segundo parámetro como respuesta del usuario el átomo 'No'.

Si el usuario selecciona 'No', el predicado *dialogar* tendrá exito y la consulta a la base actual finaliza. Si selecciona 'Si' *dialogar* falla y ese fallo provoca un regreso atrás. Como ese regreso está controlado por una tijera, se irá directamente hasta el predicado *repeat*, el que forzará un regreso adelante para comenzar otro proceso de inferencia.

- Transformar las reglas de producción.

  Las reglas de producción de la forma:

```
If
   PreguntaA > 30  And
   PreguntaB = si  And
   PreguntaC = verde
then
   H.
```

  Deberán escribirse en Prolog de la siguiente forma:

```
h:-
   [! (dialogar(preguntaA, R)  !],
   R > 30,
   [! dialogar(preguntaB, 'Si') !],
   dialogar(preguntaC, 'verde').
```

El mecanismo de inferencia de Prolog es más poderoso que el de cualquier máquina de inferencia convencional. Cuando se desee ejercer control sobre ese mecanismo, se pueden usar predicados de control, tales como las tijeras, los cortes, etc. En la regla anterior se ha hecho uso de las tijeras porque se desea un comportamiento del Prolog similar al de una máquina de inferencia. Si se desea dejar "libre" el mecanismo de regreso atrás basta con quitar las tijeras.

La primera línea de *h* hace que se invoque la caja de diálogo *preguntaA*. El segundo parámetro de *dialogar* se instanciará con el valor asociado a *preguntaA,* debido a que en este caso se le pasó la variable libre *R;* o sea, cuando se desee obtener el resultado de una respuesta o de un hecho de la base de conocimiento que fue obtenido por preguntas al usuario, se debe pasar el

segundo parámetro de *dialogar* como una variable libre.

La segunda y tercera línea de *h* hacen que se invoquen las cajas de diálogo *preguntaB* y *preguntaC* respectivamente pero en este caso se desea verificar si la respuesta o hecho asociado a *preguntaB* es 'Si' y si *preguntaC* está relacionado con el valor 'verde'.

- Definición de cajas de diálogos y menús.

  Las definiciones de cajas de diálogo y menús se deben almacenar en archivos separados, uno por cada base de conocimiento que utilice y uno para los menús Esto facilita mucho la puesta a punto del programa. Las cajas de diálogo que son utilizadas por más de una base de conocimiento se deben almacenar para ser empleadas por todas aquellas que las requieran.

- Módulo principal.

  El sistema debe contar con un módulo principal, desde el que se invoquen todos los predicados generales y los predicados principales de cada base de conocimiento (sus puntos de entrada).

  El módulo principal del sistema debe tener un predicado, como punto de entrada del programa, con el siguiente formato:

```
principal :-
    recorded(viejo, _, _),
    presentar.
principal :-
    consult(menu),
    consult(dialogar),
    cargaExplicacion,
    % poner los predicados consult que necesite.
    recorda(viejo,_,_),
    save.
```

Este predicado tiene una doble finalidad:

◊ Su primera cláusula sirve para averiguar si en la base de conocimiento ya existe una llave de nombre *viejo;* esto sucederá cuando el sistema haya sido ejecutado por segunda o posterior ocasión (todas excepto la primera vez), lo que indica que todo está preparado para comenzar el trabajo con el sistema, de ahí que se invoque a un predicado de presentación *(presentar)* que se encarga de indicar las posibilidades del sistema.

◊ La segunda cláusula se utiliza sólo la primera vez que se invoque a *principal*, ya que en ese caso el predicado *recorded(viejo, _ , _)* falló y el *backtracking* provocó que se llegara hasta ella para poder consultar todos los predicados que se deseen tener guardados permanentemente en la memoria de trabajo. Debe observarse que se está consultando el menú de la aplicación y el archivo que contiene las cajas de diálogos (en general la interfaz). También se invoca el predicado *cargaExplicacion* lo que provocará que se consulten otros predicados. Finalmente se construye un término con el nombre *viejo* y se guarda todo en forma permanente (*save*), de manera que en futuras invocaciones al predicado *principal* la búsqueda de la llave *viejo* tendrá éxito y se entrará por su primera cláusula. Algunas implementaciones Prolog sólo poseen intérpretes y esta técnica tendrá que ser valorada para ver si conviene o no la estrategia de este punto para ese caso particular. De todas formas, los predicados tipo *record* o *assert* que permiten la compilación incremental en Prolog existen en todas las implementaciones y ésta es una estrategia adecuada para tener como código permanente sólo lo que es realmente necesario.

El predicado *presentar* invocado desde *principal* será el responsable de tomar el código generado por la selección del usuario del sistema y por esa razón debe tener el siguiente aspecto:

```
presentar :-
  define_window(menu,      (0, 0), (0 , 79), ( 7,  0)),
  define_window(experto, (1, 0), (23, 79), (31, 31)),
  repeat,
    [! current_window(_,menu),
     send_menu_msg(activate(general, (0, 0)), Retorno),
      chequear(Retorno)
    !],
    Retorno == salir,
    halt.
```

Debe observarse cómo se divide la pantalla en dos zonas de trabajo (pueden ser más de acuerdo a la necesidad). Una para el menú y otra para la interfaz de preguntas - respuestas o cualquier otro mensaje del sistema.

En el módulo principal se debe definir un predicado *chequear,* que se encargue de tomar una acción adecuada de acuerdo con la elección del usuario en el menú del sistema. Este predicado debe programarse con el auxilio de un predicado *case* o similar, para que desde él se elija el predicado a ejecutar de acuerdo a un código generado por el menú. Obsérvese que se hace uso del predicado *send_menu_msg* que está enviando un mensaje a un menú llamado *general* para que se active y devuelva un código en la variable libre *Retorno,* que será instanciada cuando el usuario del sistema haga su elección. Si la versión de Prolog que se está utilizando no dispone de esta facilidad deberá programarse, lo que no constituye una tarea difícil.

- Si se desean explicaciones a la pregunta *¨por qué?* se debe tener un archivo texto con cada una de las explicaciones escritas en la forma que se muestra seguidamente. Este archivo se puede borrar una vez que se ha compilado la aplicación.

```
[reaccionA, reaccionB, reaccionC].

$ Porque estoy tratando de investigar si se
trata de una Poliamida, un Poliuretano o una
Resina de Urea, para lo cual estas reacciones son determinantes $.

'etérea'.

$ Las temperaturas de fusión determinadas
son las temperaturas de los monómeros que
forman el plástico. $.
```

En el ejemplo anteriormente mostrado, se supone que existen tres cajas de diálogo nombradas *reaccionA*, *reaccionB* y *reaccionC,* que van a tener una explicación común a la pregunta *Por Qué*, de ahí que los nombres de esas preguntas estén dentro de una lista.

Por otra parte, la pregunta *etérea* tiene una explicación que es particular para ella, de ahí que esté solamente precedida por el átomo *'etérea'*.

El predicado *cargaExplicacion*, que tendrá que incluirse en el módulo principal de la aplicación será el encargado de cargar toda la información en forma adecuada. La idea es leer ese archivo y preparar con predicados tipo *record* las explicaciones que estén asociadas con las preguntas dadas para dejarlas bajo una llave común que tendrá un término por cada explicación Si esa explicación viene precedida por una lista en el archivo fuente, se debe asociar el término dado con todas las preguntas que responden a la misma justificación.

- Inferencias *forward*.

Aunque el Prolog es un lenguaje que se basa en un mecanismo de inferencia *backward chainning*, es posible incluir dentro de él pequeñas inferencias *backward*. Puede lograrse un efecto *forward* usando los predicados de tipo *findall, setup* y *bagof*.

Si la aplicación requiere una inferencia que, totalmente o en su mayoría, se comporte *forward*, no debe utilizarse esta metodología y se debe escoger otro tipo de herramienta, como Flex [Spencer].

- Es posible trabajar con factores de certidumbre, pero por motivos de espacio no se abordan estos aspectos que serán tratados en otro artículo.

3. CONCLUSIONES

Las experiencias acumuladas sobre construcción de sistemas expertos con fines docentes y por requerimientos de la industria avalan el empleo de este método, desarrollado por el grupo de Inteligencia Artificial de la Universidad Central de Las Villas.

El método que se presenta aumenta considerablemente la productividad de los ingenieros del conocimiento, que se ven liberados de tareas propias de la programación y pueden dedicar sus esfuerzos a la Ingeniería del Conocimiento, a la vez que se les dota de una excelente herramienta de desarrollo que de otra forma no resulta fácil de emplear.

Este enfoque ha contribuido a la formación de hábitos correctos de programación en los estudiantes que lo han utilizado en sus estudios de Licenciatura o Maestría.

BIBLIOGRAFÍA

Arity/Prolog Corp. *Arity/Prolog Language Reference manual*. 1989.

BRATKO, I. *Prolog Programming for Artificial Intelligence*. 1986.

EDWARDS, J. *Building Knowledge-Based Systems*. Halsted Press. 1991.

LEZCANO, M. *Prolog y Sistemas Expertos*. Ediciones Universidad Central de Las Villas. Santa Clara, Cuba. 1996.

ROTH, A. *The practical application of Prolog*. AI Expert. April 1993.

ROTH, A. Spencer Clive. *The benefits of Prolog*. Software Development. November 1993.

RICH, E.; Knight, K. *Inteligencia Artificial*, segunda edición., Mcgraw- Hill. 1994.

SPENCER, C. LPA-flex 1.2 *Technical Product Information*. July 1996.

SPENCER, C. *The beginning of Prolog*. Prime marketing publications LTD.