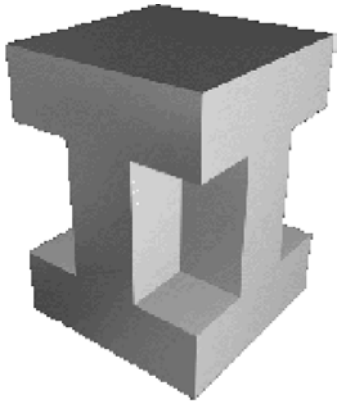
INTRODUCCIÓN AL PROLOG



Ingeniería Informática

VNIVERSITAT DE VALÈNCIA

Departamento de Informática

PRÓLOGO

Esta introducción al Prolog está pensada para las prácticas de la asignatura de Lógica y Programación de la Universitat de València, no es por tanto autocontenida ni completa. Para comprender correctamente lo aquí explicado es conveniente leer los apuntes de teoría de la asignatura.

Para cualquier comentario o error en el texto ponerse en contacto con el autor: Fernando Barber Miralles (Fernando.Barber@uv.es).

1. INTRODUCCIÓN

El Prolog es un lenguaje de programación lógica (de ahí su nombre). Esto significa que está basado en la lógica de predicados, en concreto en un subconjunto de esta lógica denominado cláusulas de Horn.

En Prolog, un programa es un conjunto de hechos y reglas que representan el problema que se pretende resolver. Ante una determinada pregunta sobre el problema, el Prolog utilizará estos hechos y reglas para intentar demostrar la veracidad o falsedad de la pregunta que se le ha planteado.

La demostración de la pregunta se basa en dos principios:

- **La unificación**: Es el algoritmo que se encarga de resolver las igualdades lógicas.
- **El principio de resolución**: Es el algoritmo que, a partir de la negación de la pregunta y los hechos y reglas del programa, intenta llegar a un absurdo para demostrar que la pregunta es cierta.

2. PROGRAMAS, PREDICADOS

Un *programa* Prolog está formado por *predicados*. Cada predicado está definido unívocamente por su *nombre* y su *aridad*. La aridad es el número de argumentos (o parámetros) de un predicado.

Ej.

```
humano(pepe).
humano(juan).
```

Para referenciar este predicado se utiliza únicamente su nombre y aridad: humano/1.

Cada predicado en Prolog puede estar definido por una o más *cláusulas*. En nuestro ejemplo, humano/1 está definido por dos cláusulas.

Las cláusulas a su vez pueden ser de dos tipos: *hechos* y *reglas*.

Los hechos son afirmaciones que consideramos ciertas en nuestro programa.

Las reglas son implicaciones lógicas, que pueden tener varios antecedentes pero un único consecuente. Este tipo de reglas se denominan cláusulas de Horn.

Ej.

humano(X) → mortal(X).

que utilizando la sintaxis de Prolog se escribe:

```
mortal(X):-
        humano(X).
```

Los hechos se pueden considerar casos particulares de reglas en donde no hay ningún antecedente.

3. TÉRMINOS DE PROLOG

Los argumentos en una cláusula pueden ser cualquier *término* de Prolog. Los términos de Prolog son los siguientes:

Átomos: Los átomos hacen las funciones de identificadores en Prolog. Un átomo puede ser cualquier combinación de caracteres alfanuméricos, pero si no empieza por una letra minúscula o si contiene caracteres que no sean letras, números o el carácter subrayado '_', debe encerrarse entre comillas simples.

Ej.

Válidos	No Válidos
unatomo	un atomo
otro_atomo	_otro_atomo
'Atomo'	Atomo
'2+3'	2+3
'un atomo'	

Enteros: Números enteros. El rango es dependiente de la implementación.

Reales: Números reales. El rango también es dependiente de la implementación.

Ej:

12.3 2.5e-4

Estructuras (funciones): Se utilizan para agrupar un número constante de términos.

Listas: Se utilizan para agrupar un número variable de términos.

Variables: Una variable puede ser instanciada a cualquier otro término de Prolog, incluida otra variable. No son equivalentes a las variables en los lenguajes procedurales, sino a las variables lógicas o matemáticas. El nombre de una variable ha de comenzar por una mayúscula o subrayado y puede contener cualquier combinación de letras, números y el carácter subrayado.

Ej:

Suma X2 _32 _

El carácter subrayado es un tipo de variable especial que se denomina *variable anónima*. Ésta se utiliza para escribir variables sin necesidad de darles un nombre. Cada aparición del carácter subrayado representa una variable distinta.

4. UNIFICACIÓN

La unificación, como ya se ha comentado, es el algoritmo que se encarga de resolver las igualdades lógicas.

Aunque la unificación en Prolog sustituye a la asignación de los lenguajes procedurales, no hay que confundirla con esta, son operaciones totalmente distintas. La operación de unificación se representa mediante el operador =.

Ejercicios:

```
X = pepe.
3 = X.
X = Y.
X = 3, X = 5.
f(3, 2) = f(X, 2).
f(X, p(a) ) = f(p(Y), Y).
X = 3 + 2.
5 = 3 + 2.
X + 3 = 2 + Y.
f(X, f(X) ) = f(Y, Y).
```

5. ARITMÉTICA

Como hemos visto, en la unificación no se evalúan expresiones. Para ello existe un operador especial 'is' que antes de realizar la unificación evalúa la parte derecha como si se tratase de una expresión aritmética.

X is 3 + 2 es equivalente a X = evaluar(3 + 2)

Para que esta evaluación se pueda llevar a cabo es necesario que la parte derecha no contenga ninguna variable sin instanciar, en caso contrario el sistema dará error. En el apéndice 1 hay una tabla con las operaciones y funciones más comunes en Prolog.

Ej.:	Resultado
X is 3 + 2.	X = 5 Yes
5 is 3 + 2	Yes
X is 3 + Y	Error. No se puede evaluar.
X is ln(exp(2))	X = 2.0 Yes

5.1 Comparación aritmética

Existen varios operadores para comparar expresiones aritméticas. Estos operadores evalúan sus dos operandos antes de realizar la comparación. En el apéndice 1 se pueden ver los distintos operadores existentes.

5 =:= 2 + 3. es equivalente a evaluar(5) =:= evaluar(2 + 3)

6. COMPARACIÓN DE TÉRMINOS

Para comparar términos existen también varios operadores que podemos ver en el apéndice 1. De estos operadores, los más importantes son el igual y el distinto. Para éstos hay que tener en cuenta que dos términos son iguales solamente si son exactamente el mismo término.

Ej.

5 == 5	Yes	
5 == '5'	No	Estamos comparando un entero con un átomo.
5 == 2 + 3	No	Estamos comparando un entero con una estructura.
5 == X	No	Estamos comparando un entero con una variable.

7. RESOLUCIÓN DE PROGRAMAS

En Prolog, todo lo que se introduce en el "prompt" del sistema Prolog se consideran preguntas. Para escribir un programa es necesario escribirlo en un fichero de texto y posteriormente cargarlo o compilarlo en Prolog. Para compilar un programa hay que escribir el nombre del fichero encerrado entre corchetes. Si el nombre tiene la extensión por defecto para el Prolog (en Eclipse es .pl) no hace falta incluirla. El nombre ha de ser un átomo, por tanto si contiene algún carácter especial deberá ir encerrado entre comillas simples.

Ej.

Para cargar el fichero prueba.pl que está en el directorio /prolog

[prueba]	ó	['prueba.pl']	Si estamos en el mismo directorio.
['/prolog/prueba.pl']			Si estamos en otro directorio.

Cuando se compila un fichero, todos los predicados que tiene (tanto hechos como reglas) pasan a ser axiomas del sistema, es decir, se consideran ciertos. A partir de ese momento se podrán hacer preguntas sobre esos predicados.

Ej.

```
humano(pepe).
humano(juan).

mortal(X):-
      humano(X).
```

Después de cargar este fichero podremos hacer preguntas como las siguientes:

```
[eclipse 1]: humano(pepe).          Yes
[eclipse 2]: mortal(pepe).          Yes
[eclipse 3]: mortal(javi).          No
[eclipse 4]: mortal(X).
                    X = pepe        more? (;)
                    X = juan
```

Si encuentra la pregunta como un hecho, la respuesta es directamente que **sí** (pregunta 1 del ejemplo anterior). Si encuentra la pregunta como la cabeza (o consecuencia) de una regla, toma cada uno de los predicados del cuerpo de la regla como nuevas preguntas y sólo si todas son ciertas, la respuesta será **sí** (pregunta 2). Si no encuentra ningún hecho o regla que haga cierta la pregunta, la respuesta será **no** (pregunta 3).

Es importante tener en cuenta que para comprobar si la pregunta coincide con un hecho o con la cabeza de una regla se aplica unificación. Esto significa que si en la pregunta hay alguna variable, ésta unificará con los términos del hecho o regla, es decir, será igual a los valores que hacen cierta la pregunta. Esto se utiliza para extraer resultados (pregunta 4). Si existe más de un resultado, habrá que pulsar ";" para verlos.

Ejercicio:
Sea el siguiente programa Prolog:

```
padre(juan, pepe).
padre(pepe, javi).
padre(pepe, jose).
padre(juan, david).
```

Completarlo con varias reglas que definan las relaciones hijo/2, abuelo/2, hermano/2, primo/2, etc. Por ejemplo, para hijo/2 será:

```
hijo(X, Y):-
      padre(Y, X).
```

8. VISIÓN PROCEDURAL DE LOS PROGRAMAS

Un programa en Prolog puede ser visto desde dos puntos de vista distintos:

- Visión declarativa: Consiste en ver el programa como un problema lógico, es decir, como un conjunto de hechos y reglas a partir de los cuales se deduce el objetivo buscado, sin importar cómo se deduce.
- Visión procedural: Consiste en ver el programa como un conjunto de instrucciones de un lenguaje, que se van ejecutando en cierto orden. Aquí es fundamental conocer el orden en que ejecuta las instrucciones el Prolog y por tanto, el proceso por el cual se llega al objetivo.

Hasta ahora hemos visto los programas con una visión declarativa, sin embargo esto no siempre es posible, debido a que existen ciertas instrucciones que no pertenecen a la lógica, sino que están dentro de las instrucciones típicas de un lenguaje de programación, como por ejemplo la entrada y salida de datos. Por ello, es necesario también conocer el orden en que se ejecutan las instrucciones.

A grandes rasgos, el orden de ejecución de las instrucciones es:

1. Para la elección de una cláusula a partir de las cláusulas que componen un predicado, el orden de elección es de arriba a abajo.
2. Dentro de una regla, para comprobar si todos los subobjetivos se cumplen, se sigue un orden de izquierda a derecha.

Cuando un objetivo tiene más de una solución, se devuelve la primera que se encuentra y se deja lo que se denomina un "punto de backtracking". Esto significa que si posteriormente se llega a un fallo, la ejecución volverá atrás hasta el punto de backtracking más cercano para probar otra solución.

(Ver apuntes de clase para completar este apartado)

Ejercicio:

1. Implementar el predicado fact/2, que calcula el factorial de un número.
2. Volver a implementar un predicado que calcule el factorial de un número, pero esta vez usando recursividad final (harán falta tres parámetros).

9. ESTRUCTURAS Y LISTAS

Las estructuras representan las funciones de la lógica de predicados. Como ya se ha comentado, las estructuras se utilizan en Prolog para agrupar un número fijo de términos.

Ej. El número complejo 2 + 3i se podría representar como:

complejo(2, 3)

El nombre de la estructura se denomina *functor*, y al igual que con los predicados, el número de argumentos es su *aridad*.

Ejercicio: Escribir un predicado que sume dos números complejos:

```
suma_c(C1, C2, Cres):-
      C1 = complejo(C1r, C1i),
      C2 = complejo(C2r, C2i),
      Cresr is C1r + C2r,
      Cresi is C1i + C2i,
      Cres = complejo(Cresr, Cresi).
```

Como se puede ver por el ejemplo, la forma de acceder a los componentes de una estructura es utilizando unificación. Aprovechando que en la cabeza de la regla ya se realiza unificación, podemos poner todas las unificaciones directamente en la cabeza, con lo que el programa quedará de la siguiente manera:

```
suma_c(complejo(C1r,C1i),complejo(C2r,C2i),
                                    complejo(Cresr,Cresi)):-
      Cresr is C1r + C2r,
      Cresi is C1i + C2i.
```

9.1. Listas

Mediante el uso de estructuras es posible crear cualquier tipo de dato, incluidos los tipos de datos recursivos. El tipo de dato recursivo más sencillo es la lista, que en Prolog ya esta predefinida y utiliza como functor el carácter punto '.' con dos argumentos: un elemento y otra lista. Una lista con los elementos '1' y '2' se escribiría de la siguiente manera:

'.'(1, '.'(2, []))

El átomo '[]' se utiliza para indicar la lista vacía. Puesto que esta notación es bastante incómoda, en Prolog se puede escribir también una lista de la siguiente forma:

[1, 2]

En esta notación, para separar una lista en sus dos argumentos se utiliza el operador '|':

[1| [2]] representa [1, 2]

Al primer elemento de la lista se le denomina *cabeza* y al resto de la lista *cola*. Para separar una lista determinada en su cabeza y su cola, se utiliza unificación, igual que sucedía con las estructuras.

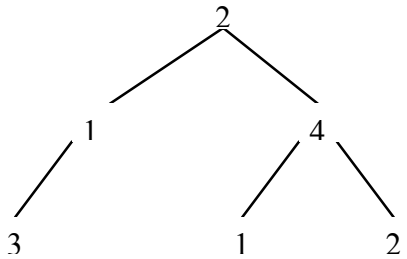
Ej. Si la lista L es [1, 2, 3]
L = [Cab| Cola]. Da como resultado:
Cab = 1
Cola = [2, 3]

Ejercicios:

1. Implementar un predicado suma_L/2 que realice la suma de una lista de números.
2. Implementar un predicado miembro/2 al que se le pasan un elemento y una lista y es cierto si el elemento está en la lista.
3. Implementar un predicado borrar/3 que a partir de una lista y un entero N, devuelva otra lista igual a la primera pero sin el elemento N-ésimo.
4. Implementar un predicado unir/3 que a partir de dos listas devuelve una tercera que es la concatenación.

9.2. Árboles

Es posible implementar otros tipos de datos con estructuras. Un ejemplo común son los árboles binarios. Para representar un árbol podemos utilizar el functor arbol con tres argumentos: valor, arbol izquierdo y arbol derecho. El árbol nulo lo representaremos como a_nulo.
Ej.



Este árbol se representará de la siguiente manera:

```
arbol(2, arbol(1, arbol(3, a_nulo, a_nulo), a_nulo),
      arbol(4, arbol(1, a_nulo, a_nulo), arbol(2, a_nulo, a_nulo) ) )
```

Ejercicios:

1. Escribir un predicado buscar/2 al que se le pasa como parámetros un árbol y un elemento y es cierto si el elemento se encuentra en el árbol.
2. Implementar un predicado write_ap/1 y un predicado write_ai/1 que recorran un árbol y lo impriman en pantalla en orden prefijo e infijo respectivamente.
3. Implementar un predicado arbol_lista_p/2 y un predicado arbol_lista_i/2 que recorran un árbol y lo conviertan en una lista siguiendo un orden prefijo o infijo respectivamente.

10. ESTRUCTURAS DE CONTROL

En Prolog, a diferencia de lenguajes procedurales como Pascal o C, no existen estructuras de control para bucles. Éstos se implementan mediante predicados recursivos. En cambio existen estructuras de control nuevas que no existen en otros lenguajes y que describiremos a continuación.

10.1. AND

El “AND” o Y lógico se representa en Prolog mediante la coma ‘,’. Este operador ya se ha visto en detalle en todos los ejemplos utilizados hasta ahora.

10.2. OR

El “OR” o O lógico se puede realizar en Prolog de dos formas distintas: mediante varias cláusulas para un mismo predicado o mediante el operador ‘;’.

Mediante varias cláusulas se consigue poniendo cada una de las opciones en una cláusula distinta del predicado. Entre las distintas cláusulas de un mismo predicado se puede considerar que existe un “OR”. Esto de hecho ya se ha utilizado por ejemplo en el factorial (se define mediante dos cláusulas) o en el predicado padre/2.

El operador ‘;’ se utiliza igual que el operador ‘,’. Únicamente hay que prestar especial atención a la prioridad de los operadores, ya que ‘;’ tiene menor prioridad que ‘,’, lo que significa que normalmente habrá que encerrar entre paréntesis la operación. A este respecto hay que recordar que el “OR” se considera una suma lógica y el “AND” una multiplicación lógica.
Ej.
Escribir un predicado numero/1 que sea cierto para los números enteros o reales y falso en caso contrario.

Varias cláusulas

```
numero(X):-
      integer(X).
numero(X):-
      real(X).
```

Operador ‘;’

```
numero(X):-
      integer(X);
      real(X).
```

Normalmente se utiliza la disyunción mediante cláusulas puesto que resulta más claro.

10.3. NOT

La negación se realiza mediante el operador not. El operador ‘not’ antes de la llamada a un predicado *P* cambia su valor de verdad, es decir, si el predicado *P* tiene éxito, not *P* fallará y si el predicado *P* falla, not *P* tendrá éxito.
Ej.

```
      no_entero(X):-
            not integer(X).
```

Sin embargo hay que tener la precaución de aplicar la negación únicamente en llamadas a predicados donde todas las variables existentes estén ya instanciadas, ya que si no el comportamiento no es el esperado.
Ej.

```
      padre(pepe, juan).

      huerfano(X):-
            not padre(Z, X).
```

Para la pregunta huerfano(pepe) funcionará bien y dirá que sí, pero si preguntamos quién es huérfano con huerfano(X) dirá que no, es decir, que no hay ningún huérfano.

10.4. Corte

El operador de corte '!' lo que hace es limitar el backtracking. Cuando se ejecuta elimina todos los puntos de backtracking anteriores dentro del predicado donde está definido (incluido el propio predicado).

Se puede considerar que lo que hace es cortar la entrada al puerto de REDO.

Existen dos razones fundamentales para utilizar el corte:

1. Para aumentar la eficiencia. Se eliminan puntos de backtracking que se sabe que no pueden dar ninguna solución. Por ejemplo, cuando hemos encontrado una solución y sabemos que no existen más.
2. Para modificar el comportamiento del programa. Se eliminan puntos de backtracking que podían dar soluciones válidas. Esto es muy peligroso pues se está modificando el significado del programa. Este tipo de corte debe utilizarse lo menos posible.

10.5. Fallo, Cierto

El predicado de fallo, *fail*, se utiliza para obligar al Prolog a dar un fallo.

El predicado cierto, *true*, se utiliza como instrucción nula, es decir, cuando se tiene que escribir una instrucción pero no se quiere que haga nada.

Un ejemplo típico es para imitar las instrucciones *if-then*:

```
cero(X):-
      (X == 0, write('Cero');
      true
      ).
```

10.6. Repeat

El predicado repeat se utiliza para crear un punto de backtracking infinito. Se puede considerar que está definido de la siguiente manera:

```
repeat.
repeat:- repeat.
```

Se utiliza típicamente para menús o en entradas de datos.
Ej.

```
menu(X):-
      repeat,
      write('Opcion:'),
      read(X),
      (X == 1; X == 2; X == 3).
```

11. BASE DE DATOS DINÁMICA

En Prolog es posible añadir o eliminar hechos y predicados de forma dinámica, es decir, mientras se ejecuta el programa. De esta forma es posible introducir o modificar datos del programa e incluso modificar el propio programa mientras se ejecuta.

Para añadir una cláusula se utiliza el predicado assert/1, que introduce la cláusula al final de todas las cláusulas del predicado.

Ej.

```
padre(pepe, juan).

[eclipse 1]: padre(pepe, X).
          X = juan
          Yes.
[eclipse 1]: assert(padre(pepe, javi) ), padre(pepe, X).
          X = juan   more? (;)
          X = javi
          Yes.
```

Si se introduce una regla, habrá que encerrarla entre paréntesis para que los distintos operadores que posea no confundan al Prolog:
Ej.

```
assert( (hijo(X, Y):- padre(Y, X) ) ).
```

Existe otro predicado para introducir la cláusula al principio de las cláusulas del predicado: asserta/1.

También existe un predicado para eliminar una cláusula: retract/1. Este predicado elimina la primera cláusula que unifique con el parámetro que se le pasa. Además puede dejar un punto de backtracking, por lo que si se hace backtracking puede eliminar nuevas cláusulas.
Ej.

```
padre(pepe, juan).
padre(pepe, javi).

[eclipse 1]: retract(padre(pepe, X) ), padre(pepe, X).
          X = javi
          Yes.
```

Ej. Eliminar todo un grupo de predicados.

```
retractall(X):-
      retract(X),
      fail.
retractall(_).
```

Cuando se utilizan estos predicados es importante distinguir entre compiladores e intérpretes. Esto es debido a que para los intérpretes todos los predicados pueden ser modificados dinámicamente. Sin embargo, para los compiladores sólo pueden ser modificados dinámicamente los predicados que se han especificado como dinámicos.

Para especificar un predicado como dinámico se utiliza el operador *dynamic*.
Ej.

```
[eclipse 1]: dynamic padre/2.
```

Aunque enunciar los predicados dinámicos resulta más incómodo, tiene como ventaja que permite depurar más fácilmente el programa.

11. GRAMÁTICAS EN PROLOG

Una gramática se puede definir fácilmente mediante reglas lógicas. Es por ello que el Prolog resulta especialmente adecuado para representar gramáticas.

En Prolog existe una notación especial para facilitar la escritura de gramáticas que se denomina Gramática de Cláusulas Definidas o DCG. Esta gramática es una generalización de las gramáticas libres de contexto.

La notación que se utiliza es la siguiente:
- Elementos terminales: encerrados entre corchetes.
- Elementos no terminales: términos de Prolog.
- Símbolo de definición: `-->`

Ejemplo:
Dada la siguiente gramática escrita en BNF:

```
<frase> ::= <sn><sv>
<sn> ::= <det><nombre>
<sv> ::= <verbo><sn>

<det> ::= el
<nombre> ::= musico | violin
<verbo> ::= toca
```

Su escritura utilizando DCG sería:

```
frase --> sn, sv.
sn --> det, nombre.
sv --> verbo, sn.

det --> [el].
nombre --> [musico] | [violin].
verbo --> [toca].
```

Esta gramática admitiría frases como “el musico toca el violin”.

Como ya se ha dicho, la DCG es una notación. Esto significa que las reglas de la gramática anterior se acaban transformando en reglas de Prolog. La forma en que se realiza esta conversión es añadiendo dos parámetros a todos los términos. El primer parámetro es una lista que contiene los tokens de entrada a la gramática, y el segundo parámetro es la lista con los tokens de salida, es decir, los tokens que aún no han sido reconocidos. Dentro de una regla, el parámetro de salida de un término es el de entrada del siguiente. La salida del último término es la salida de la regla y la entrada de la regla es la entrada del primer término.

Traducción a reglas de Prolog del ejemplo anterior:

```
frase(Le, Ls):-
      sn(Le, L1), sv(L1, Ls).
sn(Le, Ls):-
      det(Le, L1), nombre(L1, Ls).
sv(Le, Ls):-
      verbo(Le, L1), sn(L1, Ls).

det([el|Ls], Ls).
...
```

Como se puede ver, los elementos terminales se unifican con la cabeza de la lista, y la cola es la salida.

La llamada a la gramática desde Prolog se hará, por tanto, de la siguiente forma:

```
frase([el, musico, toca, el, violin], []).
```

La DCG admiten además que los elementos no terminales posean parámetros, como cualquier predicado Prolog. En este caso, los dos parámetros extra se añaden siempre al final de los otros parámetros.

Otra característica de la DCG es que se pueden añadir directamente llamadas a predicados Prolog, estas llamadas irán encerradas entre llaves (`{ }`). Todo lo que va encerrado entre llaves no sufre ninguna transformación.

Cuando se escribe una gramática utilizando DCG es necesario recordar que sigue siendo Prolog y por tanto la resolución de la gramática se realizará mediante búsqueda en profundidad y backtracking. Hay que tener por tanto especial cuidado de no introducir recursividad infinita. Por otro lado, al utilizar backtracking, la resolución puede llegar a ser muy ineficiente, por eso es bastante recomendable el uso del corte (con precaución).

APÉNDICE 1: PREDICADOS MÁS COMUNES DE PROLOG

Estructuras de control

,	Es un "and" entre objetivos. Si una cláusula tiene una secuencia de objetivos separados por "," todos los objetivos deben cumplirse para que la cláusula sea verdad.
;	Es un "or" entre objetivos. Todos los objetivos de un sólo lado de la disyunción deben ser ciertos para que la disyunción sea cierta.
!	"Cut" o "corte" es un objetivo que siempre se cumple. El corte limita la vuelta atrás ("backtracking"), de forma que si en la vuelta atrás se encuentra un corte, entonces el predicado entero que contiene el corte falla, no sólo la cláusula.
not P	El término P es interpretado como un objetivo. Si P tiene éxito, entonces not P falla y si P falla, not P tiene éxito.
true	Es un objetivo que siempre tiene éxito.
fail	Es un objetivo que siempre falla. Útil para provocar backtracking.
repeat	Este objetivo siempre es cierto y siempre deja un punto de backtracking.
(P -> Q)	If…Then. Si el objetivo P tiene éxito, se ejecuta el predicado Q. P no debe contener ningún corte (!). No se realiza backtracking sobre P.
(P -> Q ; R)	If…Then…Else. Si el objetivo P tiene éxito, se ejecuta el término Q. Si P falla, se ejecuta el término R. En ningún caso se realizará backtracking sobre P. P no debe contener ningún corte (!).

Operadores y predicados aritméticos

Operador	Explicación	Operador	Explicación
+	Suma	-	Resta
*	Producto	/	División
//	División entera	^	Exponenciación
mod	Resto de la división	abs()	Valor absoluto
sin()	Seno	cos()	Coseno
tan()	Tangente	asin()	Arco seno
acos()	Arco coseno	atan()	Arco tangente
ln()	Logaritmo neperiano	exp()	Potencia neperiana
sqrt()	Raíz cuadrada	fix()	Trunca a entero

Operadores aritméticos de evaluación

Operador	Explicación	Operador	Explicación
>	Mayor que	<	Menor que
>=	Mayor o igual que	=<	Menor igual que
=:=	Igual que	=\=	Distinto de

Clasificación de términos

Predicado	Explicación	Predicado	Explicación
atom()	Es un átomo	integer()	Es un entero
real()	Es un número en coma flotante	number()	Es un entero o un número en punto flotante
string()	Es una cadena	atomic()	Es un dato de tipo atómico. Es decir, átomo o número.
var()	Es una variable libre	nonvar()	No es una variable libre

Unificación de términos

Operador	Explicación	Operador	Explicación
=	Unifica ambos términos	\=	Cierto si ambos términos no pueden unificar

Comparación de términos

Operador	Explicación	Operador	Explicación
==	Términos equivalentes	\==	Términos no equivalentes
@<	Menor que	@>	Mayor que
@=<	Menor o igual que	@>=	Mayor o igual que

Manejo de la base de datos

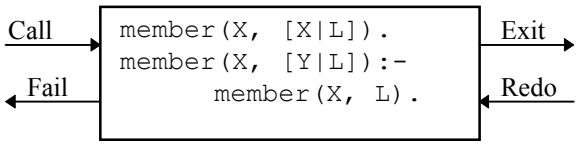
Predicado	Explicación	Predicado	Explicación
assert()	Añade una clásula al final de la lista de cláusulas del predicado	retract()	Borra la primera cláusula que unifique en la base de datos
asserta()	Añade una cláusula al comienzo de la lista de clásulas del predicado	assertz()	Igual que assert()

Entrada/Salida estándar

Predicado	Explicación
read()	Lee un término de la entrada estándar. El término debe acabar en un punto.
write()	Escribe un término en la salida estándar.
writeq()	Escribe un término en la salida estándar de forma que pueda ser leído por read().
get()	Lee un carácter (código ASCII) de la entrada estándar.
put()	Escribe un carácter (código ASCII) en la salida estándar.
nl	Escribe un salto de línea en la salida estándar.

APÉNDICE 2: EL DEPURADOR DE PROLOG

En Prolog, los predicados se consideran "cajas negras" que tienen cuatro puertos, dos de entrada y dos de salida.



Puertos:

- Call (llamada): Comienza la ejecución del objetivo.
- Exit (salida): Salida con éxito del objetivo.
- Redo (reintentar): Reintentar el predicado utilizando otra alternativa.
- Fail (fallo): Salida con fallo del objetivo. No se encontraron soluciones.

Órdenes del Depurador

Para activar el depurador se utiliza el predicado *trace/0* y para desactivarlo, el predicado *notrace/0*.

Para poner un "spy-point" (equivalente a un punto de ruptura o "breakpoint") en un predicado se utiliza el predicado *spy/1*, poniendo como argumento el predicado a "espiar".
Ej:

```
spy(member/2).
```

Las órdenes que se pueden introducir en el modo de depuración son las siguientes:

h	help - Muestra los distintos comandos disponibles.
c	creep - Avanza hasta el siguiente puerto.
<intro>	equivalente a c.
s	skip - Saltar la ejecución hasta la salida del predicado.
l	leap - Ir al siguiente "spy-point"
f	fail - Provocar un fallo.
r	redo - Reintentar el actual objetivo.
a	abort - Abortar la actual ejecución.
n	no debug - Continuar la ejecución sin depurar.

El depurador en Eclipse

```
+(4) 2 *EXIT    member(a,[b,a,c]) (dbg)?-
1 2  3 4 5          6                 7
```

1. Un '+' indica que el predicado tiene un "spy point".
2. El número de invocación del predicado. Cada llamada tiene un número único.
3. El nivel de profundidad del objetivo. El objetivo inicial tiene profundidad 0.
4. Un asterisco en el puerto EXIT indica que el objetivo ha dejado un punto de backtracking.
5. El nombre del puerto.
6. El objetivo con las variables instanciadas a los valores actuales.
7. El "prompt" del depurador. Indica que está esperando una orden del usuario. Ésta puede ser cualquiera de las vistas en la tabla anterior.