UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE INGENIERÍA

TEORÍA DE LENGUAJES (75.31)

TRABAJO PRÁCTICO

Prolog

Introducción al lenguaje y a la programación lógica

Integrantes

Axel Straminsky	XXXXX	axel_stram@hotmail.com
Demian Ferrerio	88443	epidemian@gmail.com
Martín Paulucci	88509	martin.c.paulucci@gmail.com

29 de junio de 2011

Índice

1. Programción Lógica

1.1. Concepción

El paradigma de la programación lógica surge de la necesidad de los programadores, o científicos, de expresar formalmente sus objetivos, así como sus conocimientos y suposiciones. La lógica provee las bases para deducir consecuencias a partir de premisas, para encontrar la verdad o la falsedad de una proposición a partir de otras y para verificar la valiz de de un argumento lógico.

Si bien el propósito de las computadoras es ser usadas por personas, la dificultad de su construcción fue tan grande que los lenguajes utilizados para expresar los problemas que estas debían resolver fueron diseñados desde la perspectiva del funcionamiento de la computadora en sí.

Las computadoras modernas están basadas en la arquitectura de von Neumann [2], el la cual un programa consiste en una serie de instrucciones que operan sobre registros y que se ejecutan una tras otra, pudiendo la ejecución de una instrucción influir en qué ejecución se ejecute a continuación.

A medida que los programas se hicieron más complejos, se necesitó más esfuerzo para traducir los conceptos que se querían modelar a un lenguaje que las computadoras pudieran interpretar, es decir *programar* se volvió más complejo. Para aliviar este problema se crearon lenguajes con mayor poder de abstracción, capaces de expresar las ideas del progrmador de forma más directa. Partiendo desde el lenguaje ensamblador, y pasando por C, Pascal, Java y muchos otros, todos estos lenguajes son derivados de la arquitectura de von Neumann, y por lo tanto comparten el mismo modelo subyacente de ejecución. Este paradigma de programción es el que se conoce como la *programación imperativa*.

Si bien la lógica se usó como una herramienta para diseñar computadoras y programas desde sus comienzos, su uso directo como lenguaje de progrmación, lo cual se conoce como *programción lógica*, se plantea mucho después [1] como alternativa a la programción imperativa.

[1] Finales de la década del 60 o principios de la del 70.

1.2. Declaratividad

La programción lógica, así como la programción funcional, perteneca al paradigma de la programción declarativa, y difiere enormemente de la programación imperativa. En vez estar basada en el modelo de von Neumann, la programación declarativa se basa en un modelo abstracto que no guarda ninguna relación con el modelo de la máquina.

Así, el programador, en vez de tener que adaptar sus ideas a un modelo que se diseñó para una arquitectura de computadora en un momento dado, puede expresarlas sobre un modelo diseñado para ese fín, sin tener que preocuparse por la forma en que luego la computadora ejecutará estas acciones.

Es decir, un programa en un lenguaje declarativo describe *qué* debe realizarse y no *cómo* debe realizarse.

A modo de ejemplo, se puede mencionar el popular lenguaje de consultas sobre bases de datos SQL. Éste es un lenguaje declarativo ya que el programador SQL expresa las consultas sobre un modelo abstracto (basado en el álgebra relacional) en términos de "tablas" y "registros", y es luego el entorno de ejecución (llamado motor de base de datos) quien se encarga de traducir estas consultas a instrucciones que la computadora puede ejecutar.

1.3. Programas Lógicos

Para resolver un problema dentro del paradigma lógico, en vez de darse las instrucciones necesarias para llevar a cabo las operaciones que lo resuelven, se define la información (los conocimientos) del problema y sus suposiciones en forma *axiomas lógicos*. Ese conjunto de aximas es lo que en el paradigma lógico se conoce como *programa*. Para ejecutar el programa se debe proporcionar el problema a resolver, el objetivo, expresado como una *proposición* lógica a ser probada.

Una ejecución es un intento por encontrar una solución que pruebe el objetivo dado cumpliendo con los axiomas del programa. Esta búsqueda consiste en la aplicación de las reglas de la lógica para inferir conclusiones a partir de los axiomas. El resultado de una ejecución son las soluciones que prueban la proposición objetivo. En caso de no poder probarse el objetivo, quiere decir que

el problema no tiene solución dadas las suposiciones explicitadas en el programa. De esta manera, la clave para hacer un programa lógico es poder explicitar el conjunto de axiomas que describa correctamente la solución del problema.

Una característica de la programación lógica es que las proposiciones objetivo son típicamente exitenciales. Es decir, que proponen la existencia de alguna solución que cumple con ciertas propiedades.

Un ejemplo de una proposición objetivo podría ser "existe algún X tal que X pertenece a la lista $[4, 2, 5]$". El programa donde este objetivo se ejecute deberá tener entonces los axiomas que definan la relación de pertenencia entre un elemento y una lista. Si esos axiomas están bien definidos, el resultado de la ejecución del problema serán las soluciones $X = 4$, $X = 2$ y $X = 5$.

El mismo programa lógico podría usarse para resolver problemas más complejos, por ejemplo "existe algún X tal que X pertenece a $[3, 4, 5, 6]$ y no pertenece a $[1, 2, 3, 4]$", cuyas soluciones serían $X = 5$ y $X = 6$. O "existen algunos X e Y tales que X pertenece a $[1, 2, 3]$ e Y pertenece a $[10, 20]$" en cuyo caso las soluciones serían los elementos del producto cartesiano entre los dos conjuntos dados, es decir:

$$\begin{array}{lll} X = 1, Y = 10 & X = 2, Y = 10 & X = 3, Y = 10 \\ X = 1, Y = 20 & X = 2, Y = 20 & X = 3, Y = 20 \end{array}$$

La proposición "existe algún X tal que X pertenece a $[1, 2]$ y X pertenece a $[3, 4]$" no podrá ser probada y por lo tanto su solución será vacía.

Las proposiciones objetivo no necesariamente deben ser existenciales. Por ejemplo, la proposición "3 pertenece a $[2, 3, 4]$" simplemente será verificada.

Finalmente, las soluciones no necesariamente tienen que ser finitas. Por ejemplo "existe L tal que 5 pertenece a L" tiene infinitas soluciones:

$$\begin{aligned} L &= [5|Y] \\ L &= [X_1, 5|Y] \\ L &= [X_1, X_2, 5|Y] \\ &\dots \end{aligned}$$

Donde los X_i representan elementos de cualquier valor en la lista solución, e Y representa una lista de cualquier longitud contenida al final de la lista solución.

Es decir que la solución $L = [X_1, 5|Y]$,por ejemplo, podría leerse simplemente como "una lista de al menos dos elementos cuyo segundo elemento es 5".

1.4. Cláusulas de Horn

En lógica, una cláusula es una disjunción de literales [4], por ejemplo:

$$l_1 \vee \cdots \vee l_n$$

Un cláusula de Horn es un cláusula con a lo sumo un literal positivo [5], por ejemplo:

$$\neg b_1 \vee \cdots \vee \neg b_n \vee h$$

Lo cual se puede reescribir como una implicación lógica equivalentemente:

$$(b_1 \wedge \cdots \wedge b_n) \rightarrow h \tag{1}$$

La programación lógica tiene sus bases en las cláusulas de Horn gracias a su interpretación procedural formulada por Robert Kowalski, quien demostró que una clásula de Horn como (1) puede resolverse proceduramente [6] planteando h como el encabezado (*head*) de un procedimiento y $b_1 \wedge \cdots \wedge b_n$ como su cuerpo (*body*). Lo cual puede leerse proceduralmente como: *para resolver (ejecutar)* h*, resolver (ejecutar)* b_1 *y* b_2 *y* $\cdots$ *y* b_n.

Para remarcar esta relación entre el encabezado de la cláusula y su cuerpo, en la programción lógica suele utilizarse el operador de consecuencia lógica, cuya notación resulta más similiar a la de una *regla* de Prolog (ver sección 3.4):

$$h \leftarrow (b_1 \wedge \cdots \wedge b_n)$$

2. El Lenguaje Prolog

2.1. Historia

El nombre *Prolog* proviene de la abreviatura *PROgrammation en LOGique*. Fue creado por Alain Colmerauer juanto a Philippe Roussel en 1972 en la Universidad de Marsella, y está basado en la interpretación procedual de las Cláusulas de Horn (ver sección 1.4).

La motivación que llevó al nacimiento de Prolog fue hacer un lenguaje que permita hacer programas que llevaran a cabo demostraciones automáticas de teoremas. Así empezaron los primeros trabajos de inteligencia artificial que más de veinte años después dieron lugar al primer lenguaje de programación que contempla, como parte del intérprete, los mecanismos de inferencia necesarios para la demostración automática. Este primer lenguaje está basado en el formalismo matemático de la Lógica de Primer Orden y ha dado inicio a un nuevo y activo campo de investigación entre las matemáticas y la computación que se ha denominado la Programación Lógica.

La Programación Lógica tiene sus orígenes más cercanos en los trabajos de prueba automática de teoremas de los años sesenta. J. A. Robinson propone en 1965 una regla de inferencia a la que llama resolución, mediante la cual la demostración de un teorema puede ser llevada a cabo de manera automática. La resolución es una regla que se aplica sobre cierto tipo de fórmulas del Cálculo de Predicados de Primer Orden, llamadas cláusulas y la demostración de teoremas bajo esta regla de inferencia se lleva a cabo por reducción al absurdo. Como ya se mencionó, Prolog implementa la interpretación de estas cláusulas, lo que, gracias a su eficiencia, relega los intentos anteriores de lenguajes lógicos, y se vuelve el lenguaje lógico más popular.

2.2. Implementaciones y Extensiones

Hay, en la actualidad, distintas implementaciónes de Prolog, las cuales, según sus necesidades, implementan distintos *features*.

Algunas de estas implementaciones son:

- **SWI-Prolog** añade multithreading y concurrencia por pasaje de mensajes y aritmética de presición arbitraria entre otras cosas[2].
- **LogTalk** añade orientación a objetos basada tanto en clases como en prototipos[3].
- **λprolog** soporta programación de alto orden y tipado polimórfico[4].
- **Fprolog** añade lógica difusa.
- **Prolog+** añade clases y jerarquías de clases.

2.3. Legado

Prolog fue el precursor de un paradigma que, si bien hoy en día no goza de una alta popularidad, aún sigue vigente. Influenció la creación de muchos lenguajes lógicos y funcionales, y sigue siendo el lenguaje más utilizado académicamente para enseñar el paradigma lógico.

Algunos de los lenguajes influenciados por Prolog intentaron llevar la lógica más allá del ámbito académico. Datalog[5], por ejemplo, es un lenguaje de queries para bases de datos deductivas. Mercury[6], influenciado tanto por Prolog como por Haskell, es un lenguaje logico/funcional de propósito general.

Probablemente el lenguaje derivado de Prolog de mayor uso en la actualidad sea Erlang[7] que, si bien no es un lenguaje lógico, hereda de Prolog la mayoría de su sintaxis y bastante de su semántica (e.g. declarativid, pattern matching).

Uno de los usos que se le da hoy en día a la programación lógica son los BRMS's (*business rule management system*), que utilizan motores de inferencia de reglas y en general implementan algún lenguaje lógico para la definición de las reglas de negocio. Un ejemplo popular de BRMS es Drools [8]

[2] http://www.swi-prolog.org/
[3] http://logtalk.org/
[4] http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/
[5] http://en.wikipedia.org/wiki/Datalog
[6] http://www.mercury.csse.unimelb.edu.au/
[7] http://www.erlang.org/
[8] http://www.jboss.org/drools/

3. Programación en Prolog

La elementos básicos del lenguaje derivan de la lógica: el término, que unifica todos los tipos de datos; y la proposición (*statement*), que a su vez se divide en tres tipos básicos: hechos, reglas y queries.

La programación Prolog es interactiva: un programa es un conjunto de hechos y reglas, que representan la información de un sistema y las relaciones entre los objetos que lo componen, y para extraer información de éste programa (ejecutarlo), se realizan consultas (queries).

3.1. Hechos

El tipo de proposición más simple en Prolog es el *hecho*. Respresenta una verdad en un programa. Por ejemplo.

```
gato(felix).
```

Puede interpretarse como "Felix es un gato"[9].

Los hechos sirven para definir relaciones entre elementos. Por ejemplo:

```
padre(zeus, apolo).
```

Que puede interpretarse como: la relación `padre` se cumple para los elementos `zeus` y `apolo`. El término utilizado en Prolog para referirse a las relaciones es *predicado* y para los elementos individuales *átomo* (ver Tipos de Datos 3.5). Tanto predicados como átomos deben ser identificadores que comiencen con minúscula.

Los hechos son especialmente útiles para representar relaciones entre un conjunto de elementos finitos, lo cual puede interpretarse como una forma de *base de datos*. Por ejemplo:

```
padre(urano, cronos).
padre(cronos, hades).
padre(cronos, poseidon).
padre(cronos, zeus).
padre(zeus, apolo).
padre(zeus, artemisa).
```

[9] El único único gato.

3.2. Consultas (Queries)

Otro tipo de proposición en Prolog son las consultas, o *queries*, que sirven para extraer información de un programa lógico. Las queries se *ejecutan* sobre un programa, es decir, se resuelven a partir de los axiomas definidos en él. En la programción lógica en general se las conoce como *proposición objetivo* (1.3)

Un ejemplo de query sobre el programa antes visto de los dioses griegos (3.1) podría ser `padre(zeus, apolo)`, que, al ejecutarlo[10], el sistema informa que se cumple devolviendo *true*:

```
?- padre(zeus, apolo).
true.
?- padre(zeus, cronos).
false.
?- padre(juan, tobias).
false.
```

Se puede observar que cuando la proposición no se cumple, el resultado es *false*. En este caso, la sintaxis de las queries resulta idéntica a la de los hechos; la diferencia radica en que los hechos se escriben en el programa, mientras que las queries se ejecutan en el el ambiente interactivo de Prolog, el cual antepone `?-` en cada linea para diferenciarlas de los resultados.

Este es el tipo más sencillo de queries, es decir, verificar si se cumple un predicado a partir de hechos. Mediante se presenten nuevas construcciones del lenguaje se mostrarán queries más complejas.

3.3. Variables

La variable lógica es un término de valor arbitrario (i.e. no especificado). Dar una definición más precisa que esa sería engorroso y probablemente poco claro. Su uso, sin embargo, resulta claro observando algunos ejemplos de queries que utilicen variables.

Volviendo al programa de los dioses griegos (3.1), con el uso de variables, se puede preguntar quién es el padre de Zeus:

[10] Los sistemas Prolog en general proveen un ambiente interactivo para ejecutar queries sobre programas. En este trabajo práctico se utilizó la implementación de SWI Prolog.

```
?- padre(X, zeus).
X = cronos .
```

Puede observarse que las variables, a diferencia de los átomos, comienzan con mayúscula.

Las queries con variables son en realidad predicados existenciales. En este caso, la query puede leerse como "existe X tal que X es padre de Zeus".

El resultado de una query, en general, son las substituciones de las variables que ésta utiliza tales que verifiquen la query. Una substitución es un par $\{X = a\}$ donde X es una variable y a es un término.

Puede no existir ninguna substitución de las variables de una query que la verifiquen:

```
?- padre(X, urano).
false.
```

No existe valor de `X` que verifique `padre(X, urano)`, es decir, Urano es huérfano.

Si una query tiene varias soluciones (i.e. varias susbtituciones que la verfican), estas se listan a medida que se resuelven[11]. Por ejemplo, para listar los hijos de Cronos:

```
?- padre(cronos, X).
X = hades ;
X = poseidon ;
X = zeus.
```

Con el uso de variables, pueden escribirse queries más complejas, por ejemplo, si se desea saber si Poseidon y Hades son hermanos:

```
?- padre(X, poseidon), padre(X, hades).
X = cronos .
```

Es decir "existe X tal que X es padre de Poseidon y X es padre de Hades".

Incluso puede utilizarse más de una variable en una query. Por ejemplo, para conocer todos los hermanos en la base de datos:

[11] En SWI Prolog hay que ingresar *n* (*next*) para que se siga ejecutando la query y se busque la siguiente solución. Si se presiona *enter*, se termina de ejecutar la query.

```
% El \== el operador 'distinto'
?- padre(X, A), padre(X, B), A \== B.
X = cronos,
A = hades,
B = poseidon ;
X = cronos,
A = hades,
B = zeus ;
X = cronos,
A = poseidon,
B = hades ;
% ... Hay un total de 8 soluciones.
```

Es decir, todas las ternas de individuos $\{X, A, B\}$ tales que X sea padre de A y de B, y A y B no sean el mismo individuo. Notar que la entre los resultados aparecen dos veces cada relación de "hermanos", por ejemplo $\{X = cronos, A = hades, B = poseidon\}$ y $\{X = cronos, A = poseidon, B = hades\}$ ya que esta relación es "conmutativa".

Hechos Universales

Las variables también peuden ser usadas en hechos para definir *hechos universales*. Por ejemplo, si se tiene un predicado `gusta/2`[12] que indica qué le gusta a cada persona:

```
gusta(lucas, programar).
gusta(juan, dormir).
gusta(juan, tv).
gusta(djfray, cocinar).
% ... etc
```

Es un hecho que a todo el mundo le gusta el chocolate (si no lo és, debería). Se podría escribir a mano cada hecho `gusta` entre cada individuo y `chocolate`, pero eso sería engorroso y además poco robusto, ya que si en un futuro se llegase a consultar por algún individuo no contemplado en esa cantidad finita de hechos daría que a éste no le gusta el chocolate, lo cual es inadmisible. La solución es sencilla, hacer que `gusta` sea un hecho para cualquier individuo y `chocolate`:

```
gusta(X, chocolate).
```

[12] El `/2` define la aridad de un predicado. Ver sección (3.5).

3.4. Reglas

Los ejemplos interesantes de queries existenciales ya vistos definían en sí relaciones en un modelo. Por ejemplo, la query `?- padre(X, A), padre(X, B), A \== B` define la relación "hermano" entre `A` y `B`. Las *reglas*, en principio, pueden utilizarse para plasmar esa relación en el programa lógico mismo y hacer así un modelo más interesante. Esa misma relación de hermandad traducida a una regla sería:

```
hermano(A, B) :- padre(X, A), padre(X, B), A \== B.
```

Puede interpretarse como "A y B son hermanos si existe un X tal que X es padre de A y de B y A y B son distintos". Así, queda definido un nuevo predicado `hermano/2`, que puede ser usado para hacer queries más directas:

```
?- hermano(apolo, artemisa).
true .
```

Las reglas son el equivalente en Prolog a las cláusulas de Horn. La forma general de una cláusula de Horn es:

$$h \leftarrow (b_1 \wedge \cdots \wedge b_n)$$

Que, traducido a una regla de Prolog sería:

```
h :- b1, ..., bn
```

Donde h es la cabeza (*head*) de la regla y $(b_1 \wedge \cdots \wedge b_n)$ es el cuerpo (*body*), y tanto h como $b_1 \cdots b_n$ son predicados lógicos.

Se puede observar que los hechos son en realidad un caso especial de regla cuando $n = 0$. El hecho:

```
gusta(X, chocolate).
```

Es equivalente a una regla que se cumple siempre:

```
gusta(X, chocolate) :- true.
```

Es importante notar que el cuerpo de una regla es una conjunción lógica (operador *and*). Eso quiere decir, entonces, que la cabeza de una regla es consecia de que se cumplan *todos* los predicados del cuerpo. A veces es deseable plantear que un predicado se cumpla por más de una condición, es decir una disjunción

lógica (*or*). Por ejemplo, para modelar la relación "familiar cercano" entre dos individuos, A y B, que se cumple si ambos son hermanos, o si son padre e hijo, es necesario plantear una disjunción. En Prolog, si bien existen otras formas de hacerlo[13], se estila modelar esto usando simplemente múltimples reglas:

```
familiar_cercano(A, B) :- padre(A, B).
familiar_cercano(A, B) :- padre(B, A).
familiar_cercano(A, B) :- hermano(A, B).
```

Así, se puede averiguar quiénes son los familiares cercanos de Artemisa:

```
?- familiar_cercano(artemisa, X).
X = zeus ;
X = apolo .
```

3.5. Tipos de Datos

Prolog posee un tipo de dato unificado: el *término*. Existen términos simples: los átmos, variables y numerales; y términos compuestos.

- Un **átomo** es simplemente un identificador. No tiene un sentido matemático como los números o las listas. Se escribe como una cadena de caracteres alfanuméricos comenzando con minúscula o entre comillas simples (estos últimos pueden contener espacios y comenzar con mayúscula). Algunos ejemplos de átomos: `a`, `lunes`, `'Teoria de Lenguajes'`.
- Los **numerales** pueden ser tanto enteros como de coma flotante. Los enteros son de especial interés para Prolog, ya que, al ser numerables poseen propiedades que pueden utilizarse para inferencias lógicas, como que existe *siguiente* o el *anterior* de todo número entero.
- Una **variable** representa un valor arbitrario en una expresión, y se escribe como una cadena de caracteres alfanuméricos comenzando por mayúscula, por ejemplo `X` o `Dia`.

[13] Así como el operador `,/2` (la coma) denota una conjunción, el operador `;/2` deonta una disjunción, pero no se recomienda su uso, al menos al programador Prolog principiante, ya que puede llevarlo a escribir reglas demasido complejas y no entender bien la idea de que un predicado puede definirse a partir de varias relgas.

- Un **término compuesto** es una expresión conformada por un átomo, llamado *functor*, y una serie de términos, los *argumentos* separados por coma y rodeados por paréntesis. Ejemplos de términos compuestos pueden ser `padre(X, poseidon)` o `pertenece(X, [1, 2, 3])`. Las listas y los strings son términos compuestos con una sintaxis especial.

3.6. Recursividad

Hasta ahora, usando hechos y reglas simples se pudieron definir algunos programas con capacidades de inferencia medianmente interesantes. Pero este tipo de programas no es suficiente para representar un tipo de relaciones muy interesante: las que se definen en base a sí mismas.

A modo de ejemplo, se volverá a utilizar como base el programa de los dioses griegos (3.1). Esta vez se desea representar la relación "ancestro" . Una forma ingenua de encarar este problema sería:

```
% ancestro(A, B) verifica si A es ancestro de B.
ancestro(A, B) :- padre(A, B).
ancestro(A, B) :- padre(A, X), padre(X, B).
ancestro(A, B) :- padre(A, X), padre(X, Y), padre(Y, B).
% ... ad-infinitum?
```

Es decir, plantear la relación como "A es ancestro de B si A es padre de B, o si A es abuelo de B, o si A es bisabuelo de B, o si ...". Pero, ¿hasta cuándo seguir? ¿Cuántos niveles de ascendencia son suficientes?

Es evidente que planteando la relación "ancestro" de esta manera nunca se podría llegar a una forma general. En el intento de generalizar esta relación, surge naturalmente una definición recursiva: "A es ancestro de B si A es padre de un ancestro de B, ó si A es padre de B directamente".

```
ancestro(A, B) :- padre(A, B).
ancestro(A, B) :- padre(A, X), ancestro(X, B).
```

Es importante notar que para definir la regla recursiva es necesario también otra regla no-recursiva que hace las veces de "condición de corte" para no caer en la recusividad infinita.

Ahora `ancestro/2` funciona bien para cualquier nivel de ascendencia:

```
?- ancestro(urano, apolo).
true .

?- ancestro(X, apolo).
X = zeus ;
X = urano ;
X = cronos .

?- ancestro(X, urano).
false.
```

Otra forma de definir `ancestro/2`, que resulta funcionalmente equivalente, es:

```
ancestro(A, B) :- padre(A, B).
ancestro(A, B) :- ancestro(A, X), ancestro(X, B).
```

Esta forma de definir la relación quizá resulta chocante a primera vista al programador acostrumbrado al paradigma imperativo que ve a las reglas como procedimientos y a los cuerpos de las mismas como llamadas a otros procedimientos. Es importante recordar que `ancestro/2` es un predicado definido a partir de dos reglas. Que la segunda regla de `ancestro/2` "llame" (para ponerlo en términos imperativos, aunque poco correctos) sólo a `ancestro/2`, esto no quiere decir que esa llamada sólo vaya a evaluar esa segunda regla para ejecutarse.

Más adelante se verán ejemplos de seguimientos de programas y las definiciones recursivas, en especial el modo en que se resuelven, quedará más claro.

3.7. Listas

Las listas en Prolog pueden definirse, como en muchos lenguajes funcionales, de una forma recursiva.

$$lista = \begin{cases} \textit{lista vacía} \\ \textit{un elemento seguido de una lista} \end{cases}$$

En Prolog, la lista vacía se denota `[]`, y una lista conformada por un elemento `H` seguido por una lista `T` se denota `[H|T]`. La elección de nombres `H` y `T` es en referencia a la convención utilizada generalmente en la programción funcional para referirse al primer elemento de una lista como la *cabeza* (*head*) y al resto como la *cola* (*tail*), y al operador `|/2` como *operador concatenación*.

Así, la lista con los elementos `1`, `2` y `3` podría escribirse como `[1|[2|[3|[]]]]`, pero por suerte existe una notación más conveniente: `[1,2,3]`.

Pattern Matching

La notación con el operador concatenación, sin embargo, es útil para hacer *pattern matching* en reglas (o hechos). Por ejemplo:

```
cabeza(H, [H|T]).
```

Es una hecho que dice algo trivial: la cabeza de una lista con forma `[H|T]` es `H`. Sin embargo, sirve para notar algo importante: la única lista que no *matchea*[14] con `[H|T]` es la lista vacía, hecho que se utiliza a menudo para definir condiciones de corte en predicados recursivos.

En general, cuando una variable sólo se utiliza una vez y luego no se vuelve a hacer referencia a ella (i.e. el valor de la misma no es importante para definir la regla), se suele utilizar un guión bajo como *placeholder* para indicar eso:

```
cabeza(H, [H|_]).
```

Algunas queries para empezar a creer en el pattern matching:

```
?- cabeza(X, [4,5,6]).
X = 4.
?- cabeza(X, []).
false.
?- cabeza(3, [1,2,3]).
false.
```

Se puede combinar la notación de listas con comas y con operador concatenación para hacer patrones más complejos, por ejemplo:

```
tercer_elemento(X, [_,_,X|_]).
```

El patrón `[_,_,X|_]` matchea con cualquier lista de tres elementos o más que tenga `X` como tercer elemento.

[14] "Igualar" o "hacer juego con" son pésimas traducciones, así que se opta por usar la versión castellanizada.

Reglas Recursivas

Al ser las listas una estructura recursiva en sí mismas, es natural que suelan utilizarse reglas recursivas para definir propiedades sobre ellas. Un ejemplo títpico es el predicado `pertenece/2`:

```
% pertenece(X,L) verifica si X es un elemento de la lista L
pertenece(X, [X|_]).
pertenece(X, [_|T]) :- pertenece(X,T).
```

La forma de definir un predicado recursivo resulta ya familiar (sección 3.6): una hecho básico, "X pertenece a una lista que comienza con X; y una regla recursiva: X pertenece a una lista que comienza con cualquier cosa si pertenece a su cola.

Con ésta definición de `pertenece/2` ya se pueden resolver en Prolog todas las consultas que se habían mencionado en la sección Programas Lógicos (1.3).

"Existe algún X tal que X pertenece a la lista $[4, 2, 5]$":

```
?- pertenece(X, [4,2,5]).
X = 4 ;
X = 2 ;
X = 5.
```

"Existe algún X tal que X pertenece a $[3, 4, 5, 6]$ y no pertenece a $[1, 2, 3, 4]$":

```
?- pertenece(X, [3,4,5,6]), pertenece(X, [1,2,3,4]).
X = 3 ;
X = 4.
```

"Existen algunos X e Y tales que X pertenece a $[1, 2, 3]$ e Y pertenece a $[10, 20]$":

```
?- pertenece(X, [1,2,3]), pertenece(Y, [10,20]).
X = 1,
Y = 10 ;
X = 1,
Y = 20 ;
X = 2,
Y = 10 ;
X = 2,
Y = 20 ;
X = 3,
Y = 10 ;
```

```
X = 3,
Y = 20.
```

"Existe algún X tal que X pertenece a $[1, 2]$ y X pertenece a $[3, 4]$":

```
?- pertenece(X, [1,2]), pertenece(X, [3,4]).
false.
```

"3 pertenece a $[2, 3, 4]$":

```
?- pertenece(3, [2,3,4]).
true.
```

"Existe L tal que 5 pertenece a L":

```
?- pertenece(5, L).
L = [5|_G288] ;
L = [_G287, 5|_G291] ;
L = [_G287, _G290, 5|_G294] ;
L = [_G287, _G290, _G293, 5|_G297] ;
L = [_G287, _G290, _G293, _G296, 5|_G300] ;
L = [_G287, _G290, _G293, _G296, _G299, 5|_G303] .
```

Este último caso resulta interesante. Las soluciones son infinitas[15], y la forma de cada solución es una lista con un 5. La primer solución es una lista de al menos un elemento, un 5; su cola puede ser una lista arbitraria y eso es lo que representa ese nombre poco-feliz. La segunda solución es una lista de al menos dos elementos, el primero de un valor arbitrario y el segundo un 5. Y así sucesivamente...

[15]Pero por suerte SWI Prolog requiere que el usuario vaya apretando *n* para ir calculando cada solución en vez de intentar calcularlas todas.

4. Modelo de Ejecución de Prolog

Hasta ahora se crearon programas y realizaron queries de forma "intuitiva". Para crear programas más complejos es necesario conocer el modelo de ejecución de Prolog con mayor detalle.

4.1. Unificación

El concepto de unificación en Prolog es lo que hasta ahora, informalmente, se había llamado pattern matching. Se dice, por ejemplo, que el predicado `a(2, [2,3,4])` *unifica* con `a(X,[X|Xs])` usando la sustitución (*binding*) `X=2`, `Xs=[3,4]`.

4.2. Objetivos

Un objetivo es un predicado que se quiere verificar, es decir, para el que se quiere hayar una solución. Cuando se ejecuta una query, ésta es procesada como un objetivo, pero para su resolución es probable que el motor de inferencia genere otros objetivos a probar, es por eso que se usa el término general "objetivo" para denotar "lo que se quiere resolver".

4.3. Ejemplos

La forma más sencilla de entender cómo Prolog resulve las consultas es mediante ejemplos de ejecuciones. Se estudiarán los *árboles de derivación* de distintas ejecuciones y se analizará el arloritmo de *backtracking*.

Para ejemplificar cómo Prolog resuelve las queries, se volverá a recurrir al programa de los dioses griegos. El programa completo que se utilizará en esta sección es:

```
padre(urano, cronos).
padre(cronos, hades).
padre(cronos, poseidon).
padre(cronos, zeus).
padre(zeus, apolo).
```

```
padre(zeus, artemisa).

hijo(A, B) :- padre(B, A).
hermano(A, B) :- padre(X, A), padre(X, B), A \== B.

familiar_cercano(A, B) :- padre(A, B).
familiar_cercano(A, B) :- hijo(A, B).
familiar_cercano(A, B) :- hermano(A, B).

ancestro(A, B) :- padre(A, B).
ancestro(A, B) :- padre(A, X), ancestro(X, B).
```

4.3.1. Sólo hechos

Una query muy sencilla sobre el programa anterior es:

```
?- padre(cronos, X).
X = hades ;
X = poseidon ;
X = zeus.
```

Para resolverla, Prolog busca en el programa alguna regla (o hecho) que unifique con el objetivo `padre(cronos,X)`. El orden en que Prolog busca las reglas no es arbitrario, es el orden en que fueron declaradas. En este caso, la primer regla que unifica con el objetivo es `padre(cronos,hades)`, con la sustitución `X=hades`; y al haber encontrado ya una solución que verifica la query, Prolog devuelve este resultado.

Si se le pide un siguiente resultado, Prolog deberá *volver hacia atrás* en la cadena de derivación hasta el punto que haya tenido alguna regla alternativa. Ésto es lo que se conoce como *backtracking*. En este caso volver atrás significa descartar la sustitución `X=hades` volver a intentar probar el objetivo `padre(cronos,X)` con otra regla, y así el proceso se repite hasta encontrar todas las soluciones.

El árbol de derivación para este ejemplo sencillo es:

```
       padre(cronos,X)
    _________|_________
   |         |         |
X=hades  X=poseidon  X=zeus
 true      true       true
```

4.3.2. Regla simple

Cuando la regla que unifica con objetivo a probar es una regla propiamente dicha (en vez de un hecho), Prolog sustituye las variables del objetivo en la regla y toma como nuevo objetivo el cuerpo de la regla. Es decir, intenta probar que el cuerpo de la regla se cumpla.

Por ejemplo, para la query:

```
?- hijo(X, zeus).
X = apolo ;
X = artemisa.
```

El árbol de derivación sería:

```
   hijo(X,zeus)
        |
   padre(zeus,X)
    ____|____
   |         |
X=apolo  X=artemisa
 true       true
```

Otro modo de ver cómo es que se resuelve esta query es observando un *trace*[16] de la misma:

```
[trace]  ?- hijo(X, zeus).
   Call: (6) hijo(_G732, zeus) ? creep
   Call: (7)   padre(zeus, _G732) ? creep
   Exit: (7)   padre(zeus, apolo) ? creep
   Exit: (6) hijo(apolo, zeus) ? creep
```

[16] En SWI Prolog, ejecutando `trace.` en la consola interactiva.

```
X = apolo ;
   Redo: (7)   padre(zeus, _G732) ? creep
   Exit: (7)   padre(zeus, artemisa) ? creep
   Exit: (6) hijo(artemisa, zeus) ? creep
X = artemisa.
```

Nota: la indentación se agregó para mayor legibilidad. El nivel de anidamiento de las llamadas está indicado por los números entre paréntesis.

Las variables intermedias como `_G732` aparecen a causa de sustituciones. En este caso `_G732` está bindeada a `X`, por lo cual, cuando se termina bindeando `apolo` a la misma, `X` termina también bindeada a `apolo`.

Las lineas del trace que comienzan con `Call` son evaluaciones de un objetivo y las que comienzan con `Exit` son resoluciones exitosas de un objtivo.

4.3.3. Regla compuesta y búsqueda

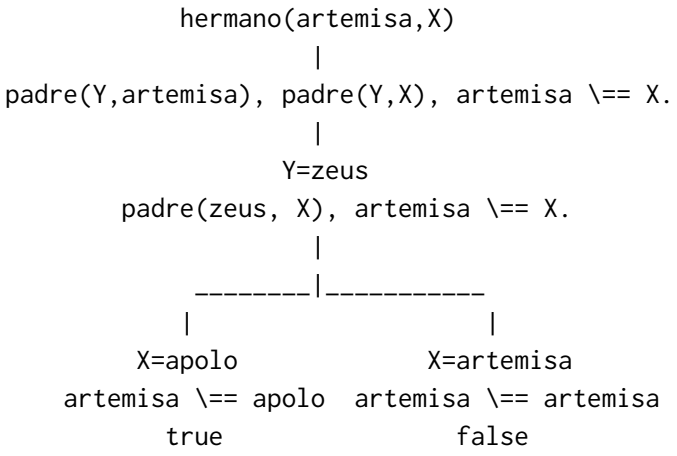
Cuando un objetivo no se puede probar, es decir que falla, Prolog vuelve hacia atrás en la cadena de derivación hasta el punto en que tenía alguna opción para probar con otra regla. En caso de no poder volver más hacia atrás (i.e. llegar hasta la raiz del árbol de derivación), la query no tendrá más resultados y se imprimirá *false*.

Por ejemplo, considérese la siguiente query:

```
?- hermano(artemisa, X).
X = apolo ;
false.
```

El `false` aparece porque luego de llegar a la solución `X=apolo` todavía existían caminos sin tomar en el árbol de derivación. Cuando se le pidió seguir calculando soluciones, el intérprete evaluó estos caminos y se encontró que ninguno arrojaba una solución.

El árbol de derivación tiene la forma:



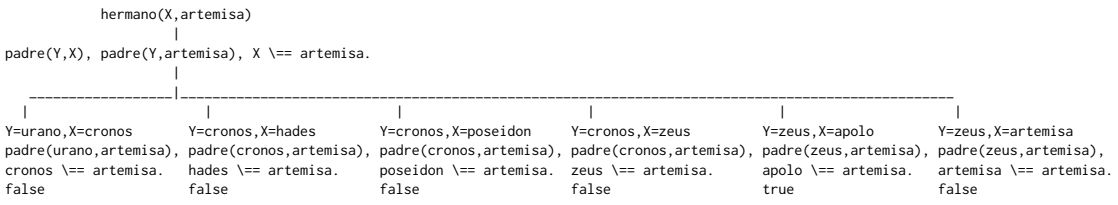
Puede observarse que para probar un objetivo compuesto, se van probando sus predicados en orden. Si alguno de estos no se puede verificar, el objetivo entero falla.

4.3.4. "Conveniencia" de una query sobre otra

Considérese la query:

```
?- hermano(X, artemisa).
X = apolo ;
false.
```

Si bien los resultados son idénticos a los del ejemplo anterior (pues el predicado `hermano/2` es "conmutativo"), el árbol de derivación resulta bastante más complejo:



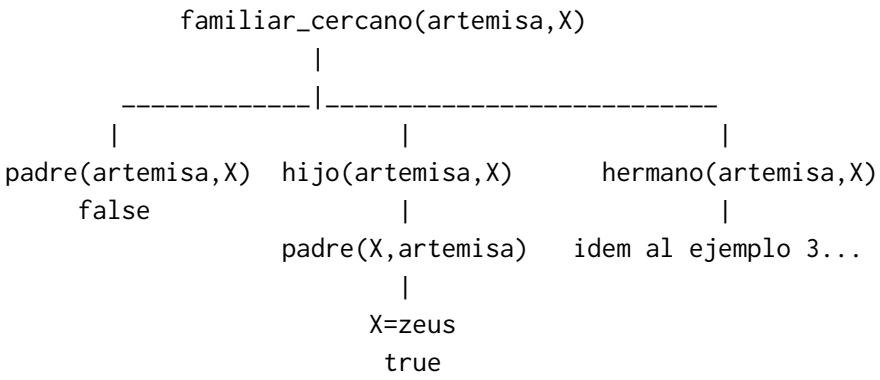
Esto ocurre porque se intenta probar el objetivo `padre(Y,X)`, con lo cual se terminan evaluando todos los 6 hechos para `padre/2`.

4.3.5. Reglas múltiples

Considérese la query:

```
?- familiar_cercano(artemisa, X).
X = zeus ;
X = apolo ;
false.
```

El árbol de derivación resulta:



Si los argumentos en la consulta se darían vuelta

```
?- familiar_cercano(X, artemisa).
X = zeus ;
X = apolo ;
false.
```

Se obtendría un árbol de derivación distinto y se observaría el mismo comportamiento que en el ejemplo 4.

4.3.6. Recursividad: listas

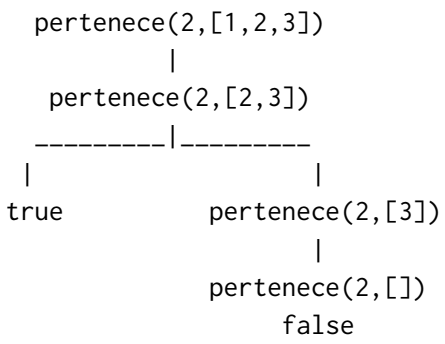
Para el ejemplo de reglas recursivas, se recurrirá al ya conocido predicado `pertenece/2` (ver sección 3.7), definido como:

```
pertenece(X,[X|_]).
pertenece(X,[_|Xs]) :- pertenece(X,Xs).
```

La consulta:

```
?- pertenece(2,[1,2,3]).
true ;
false.
```

Se resuelve con el siguiente árbol de derivación:



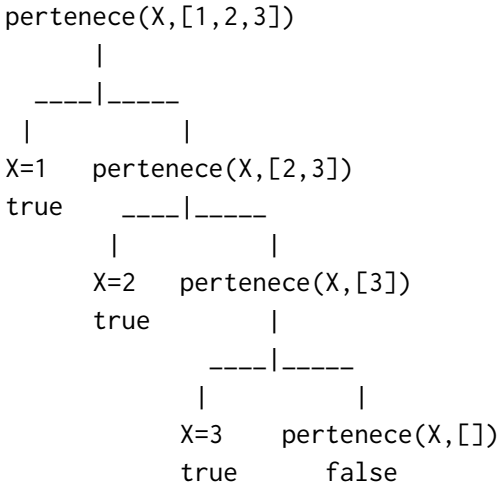
Puede observarse que, por más que la query se verifique con `pertenece(2,[2,3])` por el primer hecho, el algoritmo de backtracking sigue ejecutándose, pues `pertenece(2,[2,3])` también unifica con la cabecera de la segunda regla `pertenece(X,[_|Xs])`. Y es por ello el resultado de la query termina con un `false`. Eso quiere decir que si hay varias apariciones del elemento en la lista, la query devolverá varios `true`s;

```
?- pertenece(2,[2,1,2,3,2]).
true ;
true ;
true ;
false.
```

Esto puede parecer absurdo, pero no lo és. Si para el usuario es suficiente con que la query se verifique una vez, él puede optar por detener la búsqueda de más soluciones. Éste comportamiento del algoritmo de backtracking es el que permite obtener, por ejemplo, todos los elementos que pertenecen a una lista, con la query:

```
?- pertenece(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
false.
```

El árbol de derivación, en este caso, sería:



4.3.7. Operador !

A veces es necesario "cortar" el backtracking cuando ya se llegó a una solución. Quizá porque sólo initeresa obtener una solución, o quizá porque sólo existe una y se sigue buscando se caería en un loop infinito.

Por ejemplo, si se quiere que el backtracking corte al encontrar una solución que cumpla con `pertenece/2`, se puede hacer:

```
pertenece(X,[X|_]) :- !.
pertenece(X,[_|Xs]) :- pertenece(X,Xs).
```

El operador `!` siempre se verifica, y al evaluarse le indica al algoritmo de backtracking que debe dejar de buscar soluciones. Una query booleana, entonces, terminará ni bien se verifique:

```
?- pertenece(2,[1,2,3]).
true.
```

Pero una query existencial también terminará ni bien se encuentre una solución, con lo cual `pertenece/2` deja de ser útil para conocer todos los elementos de una lista:

```
?- pertenece(X,[1,2,3]).
X = 1.
```

4.3.8. Recursividad: factorial

A parte de las reglas recursivas para conjuntos finitos (como `ancestro/2`) y la recursividad de listas, en Prolog también se pueden hacer reglas con recursividad aritmética (o cualquier combinación de esas cosas). Y qué mejor ejemplo de la recursividad artimética que el factorial?

La definición matemática de la función factorial es:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n * (n-1)! & \text{si } n > 0 \end{cases}$$

Su taducción a Prolog resulta bastante sencilla:

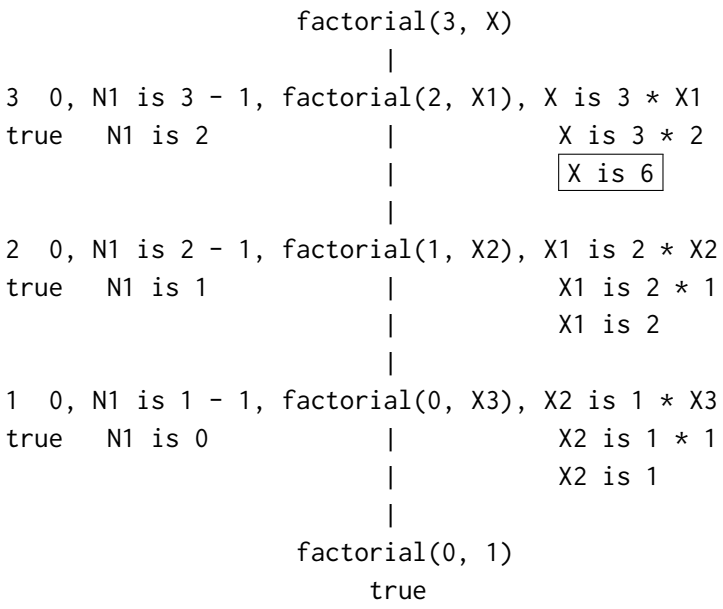
```
factorial(0, 1).
factorial(N, F) :- N   0, N1 is N - 1,
                   factorial(N1, F1), F is N * F1.
```

El operador `is` sólo bindea una variable a un valor. La primer proposición hace de condición de corte: el factorial de 0 es 1. La segunda proposición es la que se encarga de la generalización del factorial: calcula el factorial de `N - 1` en una variable `F1` y luego asocia `F` (el resultado del factorial) a `N * F1`.

La correctitud de la definición puede probarse haciendo algunas queries:

```
?- factorial(5,120).
true ;
false.

?- factorial(3, 4).
false.

?- factorial(-1, X).
false.

?- factorial(3, X).
X = 6 ;
false.

?- factorial(30, X).
X = 265252859812191058636308480000000 ;
false.
```

El árbol de derivación para `factorial(3,X)` es:



5. Ejemplos Interesantes

Vemos a continuación dos ejemplos más completos.

5.1. Cambio

"Cambio" es un programa que simula las posibilidades de cambiar un billete por monedas.

Se tiene un stock de 10 monedas de cada tipo, a saber: 5 centavos, 10 centavos, 25 centavos, 50 centavos, y 1 peso. Es decir, en total se cuenta con 50 monedas. Además, se puede pedir cambio de cualquiera de estos billetes: 2,5,10,20,50, o 100 pesos.

El código fuente del programa es el siguiente:

```
pertenece(X,[X|_]).
pertenece(X,[_|R]) :- pertenece(X,R).

stock(X) :- pertenece(X,[0,1,2,3,4,5,6,7,8,9,10]).
billetes(X) :- pertenece(X,[2,5,10,20,50,100]).

cambio(Billete,Peso,Cincuenta,Veinticinco,Diez,Cinco) :-
        stock(Peso),
        stock(Cincuenta),
        stock(Veinticinco),
        stock(Diez),
        stock(Cinco),
        billetes(Billete),
        100*Billete =:= 100*Peso + 50*Cincuenta + 25*Veinticinco + 10*Diez +
```

Luego de compilar el programa, vamos a hacer unas pruebas, y a partir de ahí vamos a razonar el funcionamiento del programa:

5.1.1. Pruebas

- **Prueba de verificación**: Se pregunta al interprete si se puede cambiar un billete de $2 por 1 moneda de un peso, 1 de cincuenta, y 2 de veinticinco.

```
?- cambio(2,1,1,2,0,0).
true .
```

Otro ejemplo, podría ser si se puede cambiar un billete de $2 por 2 monedas de un peso y una de 5 centavos.

```
?- cambio(2,2,0,0,0,1).
false.
```

Vemos que una utilidad del programa es verificar una propuesta de cambio es correcta, o no.

- **Prueba con variables**: Ahora, en vez de pasarle valores a la funciones, y que la misma nos conteste si esa condición verifica o no, es decir, si corresponde a un modelo válido o no, vamos a pasarle variables, para que el modelo nos indique *para que valores* se verifica. Hay que tener en cuenta, que al pasar las variables, la solución no es necesariamente única, sino que puede haber una, varias o ninguna solución.

  En primer lugar, vamos a pedir cambio de cincuenta, sin restricciones:

```
?- cambio(50,Peso,Cincuenta,Veinticinco,Diez,Cinco).
false.
```

  Devuelve *false*, eso quiere decir que no encontró ninguna solución posible. Esto es bastante lógico, ya que nos limita el stock de monedas: aún devolviendole todas, no vamos a poder llegar a juntar 50 pesos.

  Veamos, ahora un ejemplo que tenga soluciones. Por ejemplo, pidiendo cambio de 5 pesos:

```
?- cambio(5,Peso,Cincuenta,Veinticinco,Diez,Cinco).
Peso = 0,
Cincuenta = 2,
Veinticinco = 10,
Diez = 10,
Cinco = 10
```

  Acá vemos una solución posible. Sin embargo, el intérprete nos indica que no es la única. Apretando la tecla “n” vemos más soluciones:

```
Peso = 0,
Cincuenta = 3,
Veinticinco = 8,
Diez = 10,
Cinco = 10 ;
```

```
Peso = 0,
Cincuenta = 3,
Veinticinco = 9,
Diez = 8,
Cinco = 9 ;

Peso = 0,
Cincuenta = 3,
Veinticinco = 9,
Diez = 9,
Cinco = 7 ;

Peso = 0,
Cincuenta = 3,
Veinticinco = 9,
Diez = 10,
Cinco = 5
```

Y así podríamos seguir un buen rato...

5.1.2. Análisis del Código Fuente

Veamos el código fuente por partes. En primer lugar, uno se podría preguntar a que se refiere esta parte del código:

```
pertenece(X,[X|_]).
pertenece(X,[_|R]) :- pertenece(X,R).

stock(X) :- pertenece(X,[0,1,2,3,4,5,6,7,8,9,10]).
billetes(X) :- pertenece(X,[2,5,10,20,50,100]).
```

Vemos la regla *stock(X)*. Esta regla se puede leer como "stock(X) es verdadero si X es miembro de la lista formada por números del 1 al 10". Es decir, si X es un número de 1 a 10, *hay stock de X*. La misma regla se aplica para billetes, solo que en este caso X puede tomar los valores mencionados anteriormente.

La función "cambio":

```
cambio(Billete,Peso,Cincuenta,Veinticinco,Diez,Cinco) :-
        stock(Peso),
        stock(Cincuenta),
        stock(Veinticinco),
```

```
        stock(Diez),
        stock(Cinco),
        billetes(Billete),
        100*Billete =:= 100*Peso + 50*Cincuenta + 25*Veinticinco + 10*Diez +
```

Esta función devuelve *true* si se cumplen todas las condiciones enumeradas. Primero, que haya *stock* de monedas, de un peso, cincuenta centavos, veinticinco, 10 , y cinco centavos. Luego, que el billete ingresado tome uno de los valores válidos, a saber, 2,5,10,20,50, o 100. Por último, la condición más importante, que la suma de las monedas sea igual, en valor monetario, al valor del billete.

5.2. N Reinas

6. Conclusiones

Referencias

[1] Leon Sterling & Ehud Shapiro, *The Art of Prolog: Advanced Programming Techniques*. The MIT Press, 2nd Edition, 1999.

[2] *Von Neumann Architecture*, artículo en Wikipedia: http://en.wikipedia.org/wiki/Von_Neumann_architecture

[3] *Prolog*, artículo en Wikipedia: http://en.wikipedia.org/wiki/Prolog

[4] *Clause (logic)*, artículo en Wikipedia: http://en.wikipedia.org/wiki/Clause_(logic)

[5] *Horn Clauses*, artículo en Wikipedia: http://en.wikipedia.org/wiki/Horn_clauses

[6] Robert Kowalski, *Predicate Logic as Programming Language*. IEEE Computer Society Press, 1986.

[7] *Historia de Prolog*, http://prolog6.tripod.com/historia.htm