Conferencia 1: Programación Declarativa

M.Sc Dafne García de Armas

Tema: Programación Declarativa vs. Programación Imperativa.

Introducción a la programación lógica. Prolog

Sintaxis de Prolog, Listas.

Programar es mucho más complejo que definir una serie de instrucciones para ser ejecutadas por un ordenador. A esta diversidad en el mundo de la programación se le conoce como paradigmas o estilos de programación.

Diferentes autores (Kowalski y otros) han distinguido dos aspectos fundamentales en las tareas de programación:

- **Aspectos lógicos:** Esto es, qué debe computarse? y es la cuestión esencial y la motivación para el uso de la computadora como medio para resolver un problema determinado.
- **Aspectos de control** entre los que se distinguen:
    - Organización de la secuencia de cómputos en pequeños pasos.
    - Gestión de la memoria durante la computación.

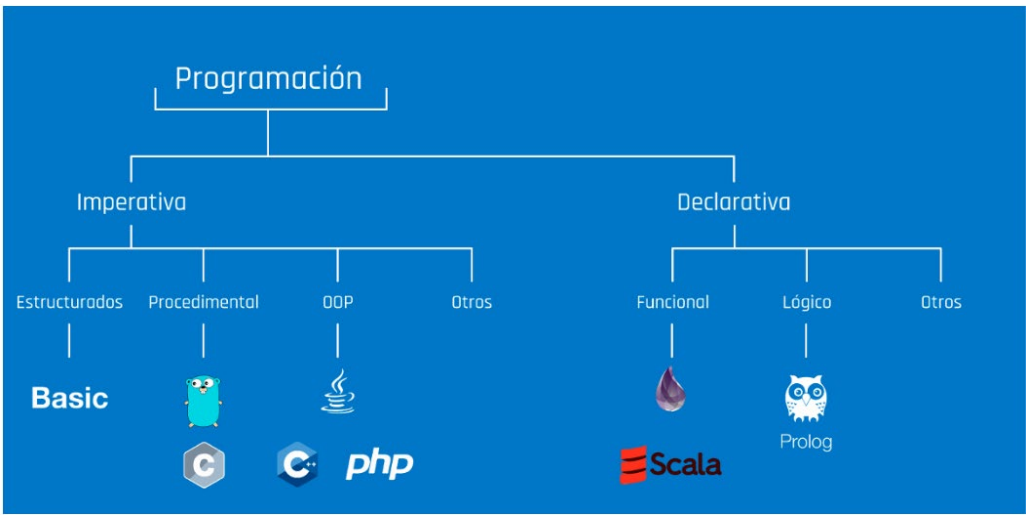
Los aspectos lógicos (especificación del problema) y los aspectos de control (implementación del programa que lo resuelve) son aspectos claramente distintos e independientes lo cual hace deseable que los lenguajes de programación permitan mantener una distancia entre ambos, sin necesidad de que las cuestiones implicadas en las tareas de especificación e implementación interfieran entre sí.

Los lenguajes imperativos no hacen esta distinción. Al mezclar los aspectos lógicos y los aspectos de control, se dificulta la verificación formal del programa, además de hacer su comprensión más difícil.

Los lenguajes declarativos, la tarea de programar consiste en centrar la atención en la lógica dejando de lado el control, que se asume automático, al sistema. La programación declarativa permite al programador concentrarse en la representación y definición de los datos y procesos relacionados con la solución de un problema haciendo abstracción de detalles de implementación y en especial del control en la interpretación y ejecución de su programa.

Estas y otras características de un lenguaje de programación dictaminan el paradigma al cual pertenece, mientras algunos lenguajes pertenecen a un solo paradigma hay otros que pertenecen a múltiples. En este curso analizaremos el paradigma de programación declarativa visto desde sus dos principales formas de expresión, la programación funcional y la programación lógica.

Es preciso comenzar este estudio mostrando los principales atributos de los dos estilos de programación mayormente usados: ***la programación imperativa y la programación declarativa.***



Programación Imperativa:

La programación imperativa, en contraposición a la programación declarativa es un paradigma de programación que *describe la programación en términos del estado del programa y sentencias que cambian dicho estado*. Los programas imperativos son un conjunto de instrucciones que le indican al computador **cómo** realizar una tarea.

La implementación de hardware de la mayoría de las computadoras es imperativa; prácticamente todo el hardware está diseñado para ejecutar código de máquina, escrito en una forma imperativa. Esto se debe a que el hardware implementa el paradigma de las Máquinas de Turing. Desde esta perspectiva de bajo nivel, el estilo del programa está definido por los contenidos de la memoria, y las sentencias son instrucciones en el lenguaje de máquina nativo (por ejemplo el lenguaje ensamblador).

Los lenguajes imperativos de alto nivel usan variables y sentencias más complejas, pero aún siguen el mismo paradigma.

Los primeros lenguajes imperativos fueron los lenguajes de máquina. En estos lenguajes, las instrucciones fueron muy simples, lo cual hizo la implementación de hardware fácil, pero obstruyeron la creación de programas complejos. Fortran, cuyo desarrollo fue iniciado en 1954 por John Backus en IBM, fue el primer gran lenguaje de programación en superar los obstáculos presentados por el código de máquina en la creación de programas complejos.

Programación declarativa

Casi simultaneando con FORTRAN en el tiempo aparece la programación declarativa con el lenguaje LISP, un lenguaje basado en el concepto de función y de definición recursiva de funciones. El concepto de recursividad fue introducido por la programación declarativa. La recursión es un poderoso instrumento de definición y de ahí su carácter declarativo y el hecho de su esencialidad y utilización sistemática en los lenguajes de programación declarativos para definir estructuras de datos y procesos. Posteriormente fue adoptado en lenguajes imperativos como ALGOL.

La declaratividad es una modalidad lógica del intercambio de información entre los seres humanos, es decir, una modalidad de ciertas proposiciones de un lenguaje con las cuales intercambiamos y compartimos conocimiento. Por ejemplo, en la lengua española las siguientes proposiciones son declarativas:

- *Julio es abuelo de Ana.*
- *Un número es par si es divisible por 2*
- *El agua hierve a 100 grados centígrados de temperatura a nivel del mar.*
- *Un paciente tiene presión alta si su sistólica es mayor que 140 y su diastólica es mayor que 90.* (Caracterización de presión alta).

Uso procedimental (imperativo) de esta última (si se quiere comprobar si un paciente tiene presión alta, determine si su sistólica es mayor que 140 y si su diastólica es mayor que 90).

Con las proposiciones declarativas en el lenguaje natural, intercambiamos conocimiento *afirmando (negando)* hechos, *enunciando* principios, leyes, reglas, *definiendo* conceptos, *describiendo* objetos, relaciones entre objetos, situaciones, sucesos, etc. Cuando afirmamos (negamos), enunciamos, definimos o describimos con proposiciones de un lenguaje, utilizamos proposiciones declarativa.

Una característica de las proposiciones declarativas es que a las mismas puede asociarse una función lógica de verdad. La más frecuente de estas funciones son las denominadas funciones booleanas en una lógica bivalente estándar que asigna a toda proposición el valor verdadero o falso como se estudió en el curso de Lógica y como es visible en la utilización de las proposiciones booleanas en programación no-declarativa.

Característica esencial de la programación declarativa:

Con más precisión, la característica esencial de la programación declarativa es el uso de la lógica como lenguaje de programación, lo cual puede conceptualizarse de la siguiente forma:

Un programa es una teoría formal en una cierta lógica, esto es, un conjunto de fórmulas lógicas que resultan ser la especificación del problema que se pretende resolver, y la computación se entiende como una forma de inferencia o deducción en dicha lógica.

Finalmente podemos decir que la Programación Declarativa, es un paradigma de programación que está basado en el desarrollo de programas especificando o "declarando" un conjunto de condiciones, proposiciones, afirmaciones, restricciones, ecuaciones o transformaciones que ***describen el problema y detallan su solución***. La solución es obtenida mediante mecanismos internos de control, sin especificar exactamente cómo encontrarla (tan sólo **se le indica a la computadora qué es lo que se desea obtener o que es lo que se está buscando**). No existen asignaciones destructivas, y las variables son utilizadas con transparencia referencial.

Los **subparadigmas de programación** dentro de la programación declarativa son: programación lógica, programación funcional, programación basada en restricciones, lenguajes de dominio específico (DSLs), lenguajes descriptivos para un propósito específico, tales como HTML, CSS y SQL.

Programación lógica

La programación lógica consiste en la aplicación del corpus de conocimiento sobre lógica para el diseño de lenguajes de programación; no debe confundirse con la disciplina de la lógica computacional. La programación lógica gira en torno al concepto de predicado, o relación entre elementos, mientras que la

programación funcional se basa en el concepto de función (que no es más que una evolución de los predicados), de corte más matemático.

Programación funcional

En ciencias de la computación, la programación funcional es un paradigma de programación declarativa basado en la utilización de funciones aritméticas que no maneja datos mutables o de estado. Enfatiza la aplicación de funciones, en contraste con el estilo de programación imperativa, que enfatiza los cambios de estado. La programación funcional tiene sus raíces en el cálculo lambda, un sistema formal desarrollado en los 1930s para investigar la definición de función, la aplicación de las funciones y la recursión. Muchos lenguajes de programación funcionales pueden ser vistos como elaboraciones del cálculo lambda.

Los lenguajes de programación funcional, especialmente los que son puramente funcionales, han sido enfatizados en el ambiente académico principalmente y no tanto en el desarrollo de software comercial. Sin embargo, lenguajes de programación importantes tales como Scheme, Erlang, Objective Caml y Haskel, han sido utilizados en aplicaciones comerciales e industriales por muchas organizaciones. La programación funcional también es utilizada en la industria a través de lenguajes de dominio específico como R (estadística), Mathematica (matemáticas simbólicas), J y K (análisis financiero), F# en Microsoft.NET y XSLT (XML). Lenguajes de uso específico usados comúnmente como SQL y Lex/Yacc, utilizan algunos elementos de programación funcional, especialmente al procesar valores mutables.

La programación funcional también puede ser desarrollada en lenguajes que no están diseñados específicamente para la programación funcional. Es el caso de JavaScript, uno de los lenguajes más ampliamente utilizados en la actualidad, el cual incorpora capacidades de programación funcional.

Ventajas de la programación declarativa

Cuando pensamos en los beneficios de programar en forma declarativa, en general se empieza pensando en las ventajas propias del lenguaje a utilizar. Por ejemplo, si se está usando un lenguaje funcional, la principal ventaja es que al lidiar puramente con funciones, no necesitamos preocuparnos por el estado de la información, ya que los datos son inmutables. Por otro lado, en el caso de los lenguajes basados en reglas, los programas son más claros y entendibles incluso por los usuarios.

A pesar de todo esto, la ventaja más importante de la programación declarativa consiste en que el indicar a la computadora "qué" tarea es la que tiene que hacer, en lugar de "cómo" hacerla nos protege de cambios en el contexto tecnológico. En ese sentido, el qué perdura mucho más que el cómo.

Imperativa vs. Declarativa:

Instrucciones o Declaraciones

En la programación imperativa, de la cual hacen parte muchos de los principales lenguajes de programación tales como ***C***, ***Java*** y ***PHP***, un **programa se describe en términos de instrucciones**, condiciones y pasos que modifican el estado de un programa al **permitir la mutación de variables**, todo esto con el objetivo de llegar a un resultado. En contraparte, en la programación declarativa un programa **se describe en términos de proposiciones y afirmaciones** que son declaradas para describir el problema, sin especificar los pasos para resolverlo; en este tipo de programas, **el estado no puede ser modificado ya que todos los tipos de datos son inmutables**.

Los lenguajes de programación imperativos generalmente hacen uso de procedimientos, rutinas o funciones impuras para establecer mecanismos de control, que potencialmente generan efectos secundarios y mutan el estado del programa durante su ejecución, a diferencia de los lenguajes de programación declarativos, en donde los mecanismos de control están dados por funciones o expresiones puramente matemáticas que carecen de efectos secundarios.

Estructuras Cíclicas de Control

La mayoría de lenguajes de programación imperativos tienen estructuras cíclicas de control tales como ***while***, ***do..while***, ***for*** y ***loop*** para iterar sobre un determinado bloque de códigos dada una condición o expresión, mientras que la gran mayoría de lenguajes declarativos, especialmente los funcionales (subparadigma declarativo), carecen de este tipo de estructuras cíclicas, de hecho la única forma de representar un flujo cíclico es a través de la recursión (funciones que se llaman a sí misma) o través de funciones de alto nivel tales como ***map*** y ***reduce*** (que internamente usan recursión).

Asignación Destructiva de Variables

La instrucción más relevante dentro de los lenguajes imperativos es la asignación. Dicha operación genera polémicas matemáticas, porque en el ámbito matemático, un símbolo de variable siempre denota el mismo objeto, cualquiera que sea la ocurrencia de ese símbolo, sin embargo la asignación utiliza las variables de modo matemáticamente impuro. Por ejemplo, en C, **a = 2+a** (asignar a la variable **a** el valor de la expresión aritmética **2+a**), donde la ocurrencia de la variable en la parte derecha de la asignación no denota lo mismo que la ocurrencia de la parte izquierda (en general, la parte izquierda de una instrucción de asignación debe entenderse como una referencia a una dirección de memoria, mientras que la parte derecha como una expresión que se evaluará para almacenar el resultado en la palabra cuya dirección es la variable de la izquierda).

En este sentido, otro aspecto importante de la programación declarativa, es que **no existe la asignación destructiva de variables**, de hecho **no existe el concepto de asignación de variables** como se conoce comúnmente en la mayoría de lenguajes imperativos. El símbolo = en los lenguajes de programación funcionales no es una asignación, es más como una aserción, ya que tiene un significado puramente algebraico, es decir cuando se escribe la expresión ***y = mx + b*** en realidad se está haciendo una igualdad, la cual solo puede ser verdadera si el valor de la derecha coincide con el valor de la izquierda.

Ejemplo: Por otra parte, en javascript una variable puede ser asignada múltiples veces, y debido a que es débilmente tipado es posible asignarle diferentes tipos de datos a una variable previamente asignada con otro tipo.

```
var x = 10;
x = 11;
x = [1, 2, 3];
X = "Hola Mundo!";
```

Mientras que en la programación declarativa tenemos:

X = Y. Es una unificación: intenta hacer X e Y iguales mediante unificación.

X \= Y. Su opuesto: no unifican

Transparencia Referencial

En los lenguajes de programación declarativos, al no haber referencias variables, se puede decir con certeza que **no hay efectos secundarios**, es decir, al *ejecutar* una función determinada, esta no cambiará nada que se haya definido por fuera de su alcance. Tampoco dependerá para nada de lo que se haya definido por fuera de ese entorno. Se dice que una función tiene transparencia referencial si, para un valor de entrada, produce *siempre* la misma salida. Esta propiedad no se da en lenguajes imperativos, donde abundan los efectos colaterales por asignaciones destructivas; es decir, no hay garantía alguna de que al ejecutar una función, procedimiento o método, este no afecte algo que se haya definido en su exterior.

Conclusiones:

Hemos puesto de manifiesto algunos puntos débiles de los lenguajes convencionales, pero el debemos ser consciente de que ellos reúnen otras ventajas que los han hecho preferidos entre los programadores. Entre estas ventajas podríamos citar: eficiencia en la ejecución, modularidad, herramientas para la compilación separadas, herramientas para la depuración de errores.

Note que lo que se denomina "declarativo" en un lenguaje imperativo, tales como la declaración de variables y sus tipos, los modos de los parámetros (de entrada, de salida) en los procedimientos es sólo información estática para el uso del compilador, es decir, no forma parte del algoritmo para resolver una clase de problemas que representa el programa. Sin embargo, las proposiciones booleanas utilizadas en las proposiciones condicionales y en algunas estructuras de control en los lenguajes imperativos son esencialmente declarativas al igual que la adopción de la recursividad. Esto demuestra que la distinción entre declarativo y procedimental no es absoluta: de la misma manera que los lenguajes procedimentales no pueden prescindir hasta cierto punto de la declaratividad, los lenguajes declarativos no pueden prescindir de la imperatividad como se podrá ver en el desarrollo de este curso, al menos en las acciones con las cuales debe receptar valores de entradas, imprimir valores de salida y sobre todo buscando eficiencia en la ejecución de un algoritmo para resolver un problema

Esta comparación revela que un lenguaje de programación no es bueno para todas las tareas. Por consiguiente cada lenguaje tiene su dominio y aplicación. Particularmente podemos enumerar algunos campos en los que la programación declarativa es utilizada en la actualidad: Procesamiento del lenguaje natural, representación del conocimiento, química y biología molecular, desarrollo de Sistemas de Producción y Sistemas Expertos, resolución de problemas, metaprogramación, prototipado de aplicaciones, bases de datos deductivas, servidores y buscadores de información inteligentes, diseño de herramientas de soporte al desarrollo del software.

APLICACIONES DESARROLLADAS EN PROLOG DE INTELIGENCIA ARTIFICIAL

Principalmente es útil en la gestión de Juegos, en Inteligencia Artificial y Sistemas Expertos, como lenguaje especialmente pensado para construir bases de conocimientos basados en la lógica que forman parte importante de cualquier agente inteligente, en la construcción de Compiladores e Intérpretes, en el Reconocimiento del Lenguaje Natural.

Casos de Aplicación:

- **Sistema Experto**
- **Procesamiento de Lenguaje Natural**
- **Asignación de Recursos Limitados: IBM** desarrollo **EI – IA** para asignación de aviones en líneas aéreas
- **Diseño experto de Productos:** Sistema Experto para diseñar anteojos a la medida de un cliente.
- **Verificación de Circuitos digitales: CVE** desarrollado por Siemens para verificar de forma automática, la salida de un circuito digital

- **RFuzzy:** Es una herramienta implementada sobre el lenguaje de programación Prolog, que es capaz de representar y trabajar con la lógica difusa. Permite representar, manejar y razonar con conceptos subjetivos como "alto", "bajo","rápido", "lento", etc. RFuzzy se ha utilizado para potenciar la inteligencia de los robots. Por ejemplo: RFuzzy se ha aplicado para la programación de robots participantes en la liga mundial de fútbol de robots (RoboCupSoccer), que se viene desarrollando desde1996 con la finalidad de desarrollar la robótica y la Inteligencia Artificial.

- **Cloze: Técnica de la Inteligencia Artificial y aprendizajes de Lengua hechas en PROLOG:**
- Consiste, en la presentación de un texto escrito al cual se le han sustituido palabras, según un criterio previamente establecido, por una marca de longitud fija o un espacio de longitud proporcional a la de la palabra. La tarea del sujeto consiste en adivinar y proponer la palabra omitida a partir de las claves sintácticas o semánticas dadas por el contexto.

- **Desarrollo de Sistemas Multi – Agentes desarrollados mediante JavaLog:**
- Desarrollado utilizando Java y el intérprete de PROLOG aumentado con un algoritmo de programación lógica inductiva denominada PROLOG

En estos momentos, los grandes de la industria comienzan a darse cuenta de los beneficios de programar en forma declarativa. Antes de su retiro de Microsoft, el propio Bill Gates aconsejó que "deberíamos estar haciendo las cosas en forma declarativa (…) no deberíamos estar escribiendo tanto código procedural". Si queremos lograr un verdadero incremento de productividad, es importante que todos dentro de la comunidad del desarrollo de software nos demos cuenta que claramente debe existir un cambio de paradigma a la hora de programar.

Introducción a la Programación Lógica

Como mencionamos anteriormente, la programación declarativa incluye como paradigmas más representativos la programación lógica y la funcional. La programación lógica se basa en fragmentos de la lógica de predicados, siendo el más popular la lógica de cláusulas definidas, mientras que la programación funcional se basa en el concepto de función (matemática) y su definición mediante ecuaciones (generalmente recursivas), que constituyen el programa, centrándose, desde el punto de vista computacional, en la evaluación de expresiones (funciones) para obtener un valor. Ambas tienen diferentes realizaciones en lenguajes como Prolog y Haskell respectivamente. En este curso veremos más aspectos de las mismas comenzando con la programación lógica.

Primeros pasos para la programación lógica

Durante el curso el lenguaje de programación lógica que se empleará será Prolog por lo cual es necesario ver los principales aspectos del mismo.

Prolog:

PROLOG, nombre que proviene del francés "PROgrammation en LOGique", es un lenguaje de programación declarativa muy utilizado en Inteligencia Artificial, principalmente en Europa. El lenguaje fue creado a principios de los años 70 en la Universidad de Aix-Marseille (Marsella, Francia) por los profesores Alain Colmerauer y Philippe Roussel, como resultado de un proyecto de procesamiento de lenguajes naturales. Alain Colmerauer y Robert Pasero trabajaban en la parte del procesamiento del lenguaje natural y Jean Trudel y Philippe Roussel en la parte de deducción e inferencia del sistema. Interesado por el método de resolución SL, Trudel persuadió a Robert Kowalski para que se uniera al

proyecto, dando lugar a una versión preliminar del lenguaje PROLOG a finales de 1971, cuya versión definitiva apareció en 1972.

La primera versión de PROLOG fue programada en ALGOL W e, inicialmente, se trataba de un lenguaje totalmente interpretado. En 1983, David H.D. Warren desarrolló un compilador capaz de traducir PROLOG en un conjunto de instrucciones de una máquina abstracta, denominada Warren Abstract Machine. Se popularizó en la década de los 80 por la aparición de herramientas para microordenadores (p.ej. Turbo PROLOG, de Borland) y su adopción para el desarrollo del proyecto de la quinta generación de computadoras, para el que se desarrolló la implementación paralelizada del lenguaje llamada KL1. Las primeras versiones del lenguaje diferían, en sus diferentes implementaciones, en muchos aspectos de sus sintaxis, empleándose mayormente como forma normalizada el dialecto propuesto por la Universidad de Edimburgo , hasta que en 1995 se estableció un estándar ISO (ISO/IEC 13211-1), llamado ISO-Prolog.

PROLOG usa Lógica de Predicados de Primer Orden (restringida a cláusulas de Horn) para representar datos y conocimiento, **utiliza encadenamiento hacia atrás** y una **estrategia de control retroactiva sin información heurística (backtracking**). Posee una semántica operacional susceptible de una implementación eficiente, como es el caso de la resolución SLD.

Como semántica declarativa se utiliza una semántica por teoría de modelos que toma como dominio de interpretación un universo de discurso puramente sintáctico: el universo de Herbrand. **La resolución SLD es un método de prueba por refutación (reducción al absurdo) correcto** y completo para la lógica HCL, que emplea **el algoritmo de unificación como mecanismo** de base y permite la extracción de respuestas (e.g. el enlace de un valor a una variable lógica).

Notación

- Las letras mayúsculas representan variables,
- El símbolo **:-** representa la operación lógica ⇒ (implicación) utilizada en su notación
  <consecuente> ⇐ <antecedente> (lectura: "<consecuente> si <antecedente>)")
- La coma representa la operación lógica de la conjunción (∧)
- El punto y coma representa la operación lógica de disyunción (∨)
- El punto representa el final de la fórmula.

Elementos del lenguaje

- Hechos (átomos).
- Reglas (cláusulas de Horn).
- Preguntas u objetivos (conjunciones ó disyunciones de átomos).

Listas

Muy utilizadas en PROLOG para almacenar series de términos (e incluso otras listas):

```
[a, f(b) , c]
[a, [b ,f(b)] , h , [i , j]]
[] /* Lista vacía */
```

Las listas constituyen una estructura de datos de amplio uso en la programación y en particular en la programación declarativa. Desde un punto de vista lógico una lista es un tipo particular de término construido mediante el símbolo constructor binario "." y el símbolo de constante *nil* que denota la lista vacía, de acuerdo con las siguientes especificaciones:

i) *nil* es una *lista*.
ii) Si *s* es un término cualquiera y *t* es una lista, entonces *.(s, t)* es una *lista*.

En la lista *.(s, t)*, *s* se denomina el *primer elemento* de la lista y *t* el *resto* de la lista. Obsérvese que el resto de una lista es también una lista. Los siguientes son ejemplos de listas:

nil
*.(*2, *nil)*
*.(*2, *.(*5, *nil))*
*.(.(*1, *nil), .(*1, *.(*3, *nil)))*

Introduzcamos desde ahora la notación externa de Prolog que facilita la manipulación de listas en un programa mediante ejemplos:

- [] denota la lista vacía.
- [1, 2, 3] denota la lista *.(*1, *.(*2, *.(*3, *nil)))* que tiene como únicos elementos los números 1, 2 y 3.
- [X |Y] denota una lista con un primer elemento X y un resto Y. Se trata de un esquema o patrón de lista que representa cualquier lista que tiene al menos un elemento.
- [X, Y |Z] es el esquema de lista que representa cualquier lista con al menos dos elementos.

Una lista se compone de:

- Cabeza [head]: El primer elemento de la lista (lo que hay antes de la primera coma).
- Cola [tail]: La lista formada por todos los elementos menos el primero (esto es, toda la lista menos su cabeza).
  [a,b,c] Cabeza: a, Cola: [b,c]
- Para extraer la cabeza y la cola de lista se usa la notación:
  [<Cabeza> | <Cola>]

Un programa en Prolog es un conjunto de fórmulas lógicas de cierta estructura que se definirá más adelante. A continuación analizaremos un primer programa Prolog en el cual se da una definición recursiva de la concatenación de dos listas. El programa concatena hace uso de la *recursividad* tanto en la manipulación de las listas que son definidas recursivamente, como en la definición de la operación que aplica (concatenar): la concatenación de una lista con otra se define recursivamente colocando primero los elementos de una de las listas y luego los de la otra.

```
concatena([], X, X). %La concatenación de la lista vacía [] y otra lista X es la propia lista X
concatena([X|Rx], Y, [X|Z]) :- concatena(Rx, Y, Z).
% La concatenación de dos listas [X|Rx] e Y es la lista que resulta de añadir el primer
% elemento X de la lista [X|Rx] a la lista Z que se obtiene al concatenar el resto Rx de la
% primera lista a la segunda lista Y.
```

Haciendo uso del mecanismo de resolución SLD, el programa puede responder a diferentes cuestiones (objetivos) sin necesidad de efectuar ningún cambio en el programa. Esto se debe a que el mecanismo de cómputo permite una búsqueda indeterminista de soluciones. Esta misma característica permite computar con datos parcialmente definidos y hace posible que la relación de entrada/salida no esté fijada de antemano. Por ejemplo, el programa sirve para responder a los siguientes objetivos:

```
:- concatenar([2,4,6],[1,5],[2,4,6,1,5]).
:- concatenar([2,4,6],[1,5],L).
:- concatenar(X,Y,[2,4,6,1,5]).
```

Al primer objetivo se responde con Verdadero (true), al segundo con L = [2; 4; 6; 1; 5] (aquí la variable L se utiliza con un sentido puramente matemático, una vez que se establece el valor de esta variable su valor no cambia, contrario a lo que sucede en los lenguajes imperativos que permiten el empleo de la asignación destructiva). El tercer objetivo tiene un conjunto de respuestas (fX = []; Y = [2; 4; 6; 1; 5]g; fX = [2]; Y = [4; 6; 1; 5]g; fX = [2; 4]; Y = [6; 1; 5]g; : : :), pues existe un conjunto de listas X y Y que su concatenación es la lista [2; 4; 6; 1; 5]. Este ejemplo pone de manifiesto las **ventajas de un mecanismo de cómputo que permita la búsqueda indeterminista de soluciones.**

El programa anterior responde a la pregunta *qué* es concatenar dos listas, es decir, da esencialmente una *definición* de concatenar como se puede apreciar a través de los enunciados que constituyen una lectura declarativa de las fórmulas del programa.

Implementación de predicados sobre listas

- isList(L):- triunfa si L es una lista.
  myIsList([]).
  myIsList([_|Y]).

isList([[]|[]]). True;	isList(A). A = [] ; A = [_G8|_G9]
isList(2). false	isList([X|X]). True;

- member(X, L):- triunfa si X es un elemento de la lista L. *I/O Adireccional*.
- **myMember(X,[X|_]).**
- **myMember(X,[Y|R]):-myMember(X,R).**

```
myMember(1, [1,2,3]).
true ;
false.

myMember(1, X).
        X = [1|_G360] ;
        X = [_G8, 1|_G12] ;
        X = [_G8, _G11, 1|_G15] ;
X = [_G8, _G11, _G14, 1|_G18]
```

```
myMember(X, [1,2,3]).
        X = 1 ;
        X = 2 ;
        X = 3 ;
false.
myMember(X, X).
        X = [X|_G9] ;
        X = [_G8, X|_G12] ;
        X = [_G8, _G11, X|_G15] ;
        X = [_G8, _G11, _G14, X|_G18] ;
X = [_G8, _G11, _G14, _G17, X|_G21]
```

- Divide(L, I, P):- triunfa si *L* es la concatenación alternada de las listas *I* y *P*.

myDivide([], [], []).
myDivide([X], [X], []).
myDivide([X,Y|Z], [X|V], [Y|W]):-myDivide(Z, V, W).

```
myDivide([1,2,3,4], I, P).
```

```
myDivide(X, [1,2], [3,4]).
```

- Push(X, L, R):- triunfa si R es el resultado de adicionar al inicio de L a X.
  myPush(X, Y, [X|Y]).

  myPush(1,[2,3], L).
  L = [1,2,3]

  myPush(X, Y, [1,2,3]).
  X = 1,
  Y = [2, 3].

- Pop(L, R):- triunfa si R es el resultado de eliminar el primer elemento de L.
  myPop([X|Y], Y).

- Add(X, L, R):- triunfa si R es el resultado de adicionar X al final de L.

  myAdd(X, [], [X]).

  myAdd(X, [Y|Z], [Y|R]):-myAdd(X, Z, R).

- concat(L1, L2, R):- triunfa si R es el resultado de concatenar las listas L1 y L2.
  myConcat([1,4],[2,3], L).

  myConcat(X, Y, [1,2,3]).

- **Implementar member, push, pop y add a partir del Concat.**

- Insert(X, L, R):-triunfa si R es el resultado de insertar a X en alguna posición de R.

- Eliminar(X, L, R):- triunfa si R es el resultado de eliminar X de la lista L.

- Eliminar(X, L, R):- eliminar el último elemento de una lista.

- Permutation(L, R):- triunfa si R es una permutación de L.

- Consecutive(X, Y, R):- triunfa si X, Y son elementos consecutivos de L.

- Reverse(L, R):- triunfa si R es el reverso de L.

- Palindrome(L):-triunfa si L es palíndromo.

- Subconjunto(S, L):- triunfa si S en un subconjunto de la lista L.

Para finalizar dejamos abierta la siguiente interrogante, si en la programación declarativa las tareas lógicas de un programa están bien planteadas, ¿Quién se encarga entonces del control o ejecución de un programa declarativo? Como se estudiará en este curso con respecto al lenguaje Prolog, un *intérprete* de los programas convenientemente construido *hace una lectura procedimental de un programa declarativo y lo ejecuta.* En especial el intérprete introduce el control necesario para la ejecución del programa.