

PROGRAMACIÓN DECLARATIVA

JOSÉ CARPIO CAÑADA
GONZALO ANTONIO ARANDA CORRAL
JOSÉ MARCO DE LA ROSA



Universidad
de Huelva

MATERIALES
PARA LA
DOCENCIA
[95]

2010

©

Universidad de Huelva
Servicio de Publicaciones

©

Los Autores

Maquetación
COPIADORAS BONANZA, S.L.

Impresión
COPIADORAS BONANZA, S.L.

I.S.B.N.
978-84-15147-05-3

Presentación

El material que aquí presentamos pretende guiar al alumno en el estudio de la Programación Lógica y la Programación Funcional, ambos paradigmas incluidos dentro de la Programación Declarativa.

Este material tiene un enfoque eminentemente práctico. Hemos reducido al mínimo los conceptos teóricos, incluyendo únicamente los elementos que consideramos imprescindibles para entender que es un programa declarativo. Hemos incluido los conocimientos teóricos básicos para que el alumno pueda empezar a programar declarativamente desde la primera sesión.

El material está dividido en catorce sesiones teóricas y nueve sesiones prácticas con una duración aproximada de una hora y media por sesión. Las sesiones teóricas incluyen ejercicios que sirven para reforzar los conceptos teóricos. En las sesiones prácticas se proponen una serie de ejercicios que el alumno debe programar utilizando el ordenador y el compilador o intérprete de Prolog o Haskell según el caso.

¿Cómo utilizar este material?

Entender la filosofía de la programación declarativa no es tarea fácil. Requiere de un cambio en la forma de pensar un programa. Pasamos de ordenar una serie de acciones (programación imperativa de C, C++, Java o Perl) a describir el problema utilizando una serie de reglas (programación declarativa Prolog y Haskell). Para llegar a entender el paradigma de la programación declarativa necesitamos reforzar la idea de que es posible programar cambiando el enfoque en la construcción de los programas. Esto se consigue de forma progresiva, empezando por ejercicios simples, aumentando la dificultad hasta implementar ejercicios con estructuras de datos complejas como árboles o grafos.

Al principio, puede ayudar al alumno tener cerca el ordenador y probar los ejercicios propuestos y ver que realmente funcionan. No siempre resulta evidente ver que un programa declarativo funciona tras echarle un vistazo al código. Y lo que es más importante, de qué forma se ha llegado a la solución. Este material utiliza un enfoque basado en el principio de inducción matemática y se apoya en este en el proceso de construcción de los programas. No todos los manuales que encontramos en la bibliografía resaltan esta idea. Más bien, utilizan un enfoque similar a la descripción de otros lenguajes imperativos, sintaxis del lenguaje y conjunto de funciones, métodos o predicados. Nosotros pensamos que en la programación declarativa es fundamental entender que es necesario cambiar el enfoque a la hora de programar y hacemos un especial hincapié en ello. La idea de que a partir del caso más pequeño ($n-1$) deducimos el caso genérico (n) se repite en la mayoría de las sesiones.

Tras el estudio de este material, si el alumno ha comprendido el modo de construir los programas no necesitará seguir mentalmente la ejecución del programa para saber si este funciona. Basándose en el principio de inducción, debe ser capaz de saber si el programa puede funcionar o no tras echarle un vistazo al código. A partir de este momento, los ejercicios que al principio parecían imposibles de entender o programar, de se muestran entendibles desde la perspectiva declarativa.

Esperamos que así sea y que tras el estudio de este material el alumnos entienda que se pueden hacer programas de un modo distinto.

Agradecimientos

Queremos agradecer a los alumnos que asistieron a nuestras clases con atención e interés aportando muchas ideas que están reflejadas ahora en este manual de docencia. A nuestro compañero Manuel Maestre Hachero que amablemente se ofreció a revisar el material y corregirlo desde la visión de un matemático. A Francisco Moreno Velo, Marcos del Toro Peral, Águeda López Moreno y a Diego Muñoz Escalante que impartieron con nosotros esta asignatura y aportaron sus ideas. A los profesores Jim Royer y Susan Older de la Universidad de Siracusa en Estados Unidos que amablemente nos permitieron incluir los ejercicios de la segunda sesión de Haskell de este manual.

Autores

José Carpio Cañada
Gonzalo Antonio Aranda Corral
José Marco de la Rosa

CONTENIDOS

TEORÍA

Programación funcional con Haskell

0. Introducción
1. Definición de funciones
2. Prioridad de operadores
3. Evaluación perezosa
4. Funciones de orden superior
5. Currificación
6. Composición de funciones
7. Listas
8. Patrones
9. Tuplas
10. Recursividad

Programación lógica con Prolog

0. Introducción
1. Unificación
2. Tipos de datos
 - 2.1 Listas
 - 2.2 Árboles
 - 2.3 Grafos
3. Control de la ejecución
4. Problemas de estados

PRÁCTICAS

Programación funcional con Haskell

- Introducción al entorno de programación Haskell Hugs
- Listas en Haskell
- Patrones, tuplas, recursividad y notación extendida de listas en Haskell
- Ejercicios de Haskell de exámenes anteriores

Programación lógica con Prolog

- Introducción al entorno SWI-Prolog
- Predicados sencillos en Prolog
- Predicados sobre listas en Prolog
- Árboles en Prolog
- Grafos en Prolog

TEORÍA

Introducción a la programación declarativa

0. Introducción

1. Paradigmas de programación

2. Programación imperativa

3. Programación declarativa

3.1 Programación funcional

3.2 Programación lógica

4. Bibliografía

SECCIÓN O. INTRODUCCIÓN

El objetivo de esta asignatura es introducir al alumno en una forma de programación basada en un paradigma muy diferente al más usual paradigma imperativo.

Esta tarea ayudará al alumno a mejorar su capacidad de abstracción y a practicar con conceptos de programación como la recursividad o la evaluación perezosa, poco comunes en otras materias de programación.

El paradigma declarativo provee al alumno además de un entorno conceptual frecuentemente utilizado para el estudio de determinados problemas de inteligencia artificial.

SECCIÓN 1. PARADIGMAS DE PROGRAMACIÓN

Paradigma: (según el diccionario Merrian-Webster)

“A philosophical and theoretical framework of a scientific school or discipline within which theories, laws and generalizations, and the experiments performed in support of them are formulated; broadly : a philosophical or theoretical framework of any kind.”

“Un entorno filosófico y teórico de una escuela científica o disciplina dentro del cual son formulados teorías, leyes y generalizaciones de éstas, así como los experimentos realizados para justificarlos; De forma más general: un entorno filosófico o teórico de cualquier tipo.”

La programación de ordenadores se puede definir como la creación de descripciones codificadas que un ordenador pueda interpretar para resolver un determinado problema. Entenderemos problema en su forma más amplia, incluyendo cualquier funcionalidad de interacción que podamos imaginar con los distintos componentes de la máquina.

La tarea de programación dependerá totalmente de la capacidad de interpretación de código del ordenador en cuestión ya que tendremos que adaptar las descripciones generadas al lenguaje entendido por la máquina.

En los inicios, el ‘lenguaje de programación’ de la máquina era algo tan rudimentario como la conexión eléctrica de un punto a otro dentro de un circuito. Con el tiempo hemos llegado a un muy diverso abanico de lenguajes que se han ido creando de forma incremental sobre ese ‘lenguaje’ primigenio.

Los diversos lenguajes de programación nos permiten comunicarnos con el ordenador con la intención de resolver un problema. Sin embargo, no es trivial establecer el modo en que es más interesante estructurar esta comunicación y varias aproximaciones diferentes han ido desarrollándose desde los inicios de la programación hasta hoy.

Cada lenguaje de programación aborda la solución de problemas en base a un conjunto de conceptos y una visión de cómo el ordenador debe comportarse para resolver dichos problemas. Si bien existen multitud de lenguajes de programación, los conjuntos de conceptos y visiones del dominio de problemas no son tan variados, de modo que cada conjunto tiene asociado normalmente varios lenguajes.

Cada uno de estos conjuntos de visión y conceptos condicionan el modo en que plantearse los problemas y las soluciones para poder expresarlos de modo que el ordenador sea capaz de resolverlos. A cada uno de estos conjuntos se denomina **paradigma de programación**.

Ejemplos de diferentes paradigmas de programación son:

- Programación imperativa
- Programación estructurada
- Programación declarativa
- Programación lógica
- Programación funcional
- Programación dirigida por eventos
- Programación modular
- Programación orientada a aspectos
- Programación orientada a objetos
- Programación con restricciones

SECCIÓN 2. PROGRAMACIÓN IMPERATIVA

El primer paradigma que se suele estudiar es el paradigma imperativo. Este paradigma entiende que para resolver un problema se deben realizar una serie de pasos y el programador es el encargado de describir de forma ordenada y sistemática los pasos que debe seguir el ordenador para obtener la solución.

Ejemplos de lenguajes imperativos, si bien hay muchos más, son:

- BASIC
- Lenguaje de programación C
- Fortran
- Pascal
- Perl
- PHP

Ejemplo de programa: ordenación con el algoritmo de la burbuja.

Declaraciones iniciales:

```
void intercambiar(int *x,int *y){
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
void burbuja(int lista[], int n){
    int i,j;
    for(i=0;i<(n-1);i++)
        for(j=0;j<(n-(i+1));j++)
            if(lista[j] > lista[j+1])
                intercambiar(&lista[j],&lista[j+1]);
}
```

Resolución del problema:

```
void main(){
    // ...
    burbuja(lista,ELEMENTOS_LISTA);
    // ...
}
```

SECCIÓN 3. PROGRAMACIÓN DECLARATIVA

El paradigma declarativo, en cambio, plantea que los problemas sean descritos al ordenador con una serie de unidades conceptuales básicas que se pueden combinar según unas determinadas reglas para generar nueva información. Para la resolución de un problema, se crearán las descripciones que representan al dominio en el cual existe el problema a resolver y se planteará el problema como una pregunta que debe ser respondida bien con alguna de las unidades conceptuales iniciales o bien con una combinación válida de ellas que es el propio ordenador quien debe buscar.

Las unidades conceptuales para la descripción del problema dependerán del subparadigma de programación concreto. En esta asignatura estudiaremos dos subparadigmas del paradigma de programación declarativa: programación funcional y programación lógica.

En programación lógica, la unidad conceptual básica será el predicado lógico y en programación funcional será la función.

Además de los subparadigmas mencionados, existen otros como los lenguajes algebraicos (por ejemplo, SQL) o la programación basada en restricciones.

Ventajas:

- Descripciones compactas y muy expresivas. Es posible describir universos de problemas con muy pocas líneas de código del lenguaje que permitan la solución de un gran número de problemas.
- Desarrollo del programa no tan orientado a la solución de un único problema. Con una programación adecuada, basta haber descrito un dominio de problemas de forma correcta y saber formular nuestro problema como una simple consulta en dicho dominio. La variedad de preguntas que se pueden responder con una única descripción del dominio de problemas concreto suele ser muy elevada.
- No hay necesidad de emplear esfuerzo en diseñar un algoritmo que resuelva el problema.

3.1 Programación funcional

La programación funcional es un subparadigma de la programación declarativa que utiliza la función como concepto descriptivo básico. Esto quiere decir que en nuestro programa describiremos funciones y que estas funciones se pueden combinar unas con otras para generar nuevas funciones. Entre otras acciones que podremos realizar con funciones, podremos evaluarlas. Nuestro programa consistirá pues en la definición de una serie de funciones que son, a su vez, composición de funciones básicas y el problema que queremos resolver se planteará normalmente como la evaluación de una función basadas en las previamente definidas.

De entre los diferentes lenguajes de programación funcional existentes, en esta asignatura estudiaremos el lenguaje **Haskell**.

<http://www.haskell.org/haskellwiki/Introduction>

Ejemplo de programa: comprobación de que un número es natural.

Declaraciones iniciales:

```
natural::(Num a, Ord a)=> a -> Bool
natural 1 = True
natural n
| n > 1 = natural (n-1)
| otherwise = False
```

Resolución del problema:

```
Main> natural (-1)
False :: Bool

Main> natural (10)
True :: Bool

Main> natural (12.5)
False :: Bool
```

3.2 Programación lógica

La programación lógica es otro subparadigma de la programación declarativa que utiliza el predicado lógico como concepto descriptivo básico. Nuestro programa consistirá en una serie de predicados que describirán un mundo en el que los objetos se relacionan según las reglas de la lógica de predicados. Nuestros problemas plantearán afirmaciones para las que el sistema será capaz de obtener una explicación lógica en base a los predicados programados en caso de que ésta existiera.

De entre los diferentes lenguajes de programación funcional existentes, en esta asignatura estudiaremos el lenguaje **Prolog** en concreto la versión software libre SWI-Prolog.

<http://www.swi-prolog.org/>

Ejemplo de programa: relaciones de descendencia.

Declaraciones iniciales:

```
% padre(Padre,Hijo)
padre(juan,pedro).
padre(pedro,lluis).
padre(iker, yeray).

% descendiente(Descendiente,Ascendiente)
descendiente(Descendiente,Ascendiente):-
    padre(Ascendiente,Descendiente).
descendiente(Descendiente,Ascendiente):-
    padre(Ascendiente,Intermedio),
    descendiente(Descendiente,Intermedio).
```

Resolución de problemas:

```
?- padre(juan,X).
X = pedro.

?- descendiente(lluis, juan).
true .

?- descendiente(X, juan).
X = pedro ;
X = lluis ;
false.
```

SECCIÓN 4. BIBLIOGRAFÍA

Bibliografía general

Programación en *Prolog*

Autores: W.F. Clocksin, C.S. Mellish

Editorial: Springer Verlag

Año: 1994

Prolog programming for artificial intelligence

Autores: Ivan Bratko

Editorial: Addison Wesley

Año: 1990

Razonando con *Haskell*

Autores: Blas C. Ruíz, F. Gutiérrez, y otros

Editorial: Thompson

Año: 2004

Bibliografía específica

Prolog: the standard

Autores: P. Deransart, A. EdDbali,

L. Cervoni

Editorial: Springer

Año: 1996

An introduction to computing in *Haskell*

Autores: Manuel M. T. Chakravarty, Gabriele C. Keller

Editorial: Pearson SprintPrint

Año: 2002

Lenguajes de programación. Principios y paradigmas

Autores: A. Tucker, R. Noonan

Editorial: Mc GrawHill

Año: 2003

PROGRAMACIÓN FUNCIONAL CON HASKELL

Programación funcional con Haskell

0. Introducción

1. Definición de funciones

2. Prioridad de operadores

3. Evaluación perezosa

4. Funciones de orden superior

5. Currificación

6. Composición de funciones

7. Listas

8. Patrones

9. Tuplas

10. Recursividad

SECCIÓN 0. INTRODUCCIÓN

(<http://www.haskell.org/haskellwiki/Introduction>)

¿Qué es la programación funcional?

Lenguajes como C, Pascal, Java son lenguajes imperativos. Se llaman imperativos porque lo que hacemos al programar es indicar cómo debe resolverse algún problema. Indicamos el orden en el que deben realizarse las acciones.

```
main() {  
    primero_haz_esto();  
    despues_esto_otro();  
    por_ultimo_esto();  
}
```

El objetivo en la programación funcional es definir QUÉ CALCULAR, pero no cómo calcularlo.

Un ejemplo de programación funcional es la que realizamos cuando programamos una hoja de cálculo:

- No se especifica en qué orden deben calcularse las celdas. Sabemos que se realizarán los cálculos en el orden adecuado para que no existan conflictos con las dependencias.
- No indicamos cómo organizar la memoria. Aparentemente nuestra hoja de cálculo es infinita, sin embargo, sólo se reserva memoria para las celdas que estamos utilizando.
- Indicamos el valor de una celda utilizando una expresión, pero no indicamos la secuencia de pasos a realizar para conseguir este valor.

En una hoja de cálculo no sabemos cuándo se realizan las asignaciones, por lo tanto, no podemos hacer uso de éstas. Ésta es una diferencia fundamental con los lenguajes imperativos como C o Java, en los cuales se debe hacer una cuidadosa especificación de las asignaciones y en lo que controlar el orden de las llamadas a funciones o métodos es crucial para darle sentido al programa.

Ventajas de la programación funcional

- Programas más concisos
- Más fáciles de comprender
- Sin “core dumps”

Características de Haskell

Haskell es un lenguaje de programación funcional puro, con **tipos polimórficos estáticos**, con evaluación perezosa (**lazy**), muy diferente a otros lenguajes de programación. El nombre lo toma del matemático Haskell Brooks Curry especializado en lógica matemática. Haskell está basado en el **lambda cálculo**. La letra griega lambda es el logotipo de Haskell

- Inferencia de tipos. La declaración de tipos es opcional
- Evaluación perezosa: sólo se calculan los datos si son requeridos
- Versiones compiladas e interpretadas
- Todo es una expresión
- Las funciones se pueden definir en cualquier lugar, utilizarlas como argumento y devolverlas como resultado de una evaluación.

Implementaciones de Haskell

Existen diferentes implementaciones de Haskell: GHC, Hugs, nhc98 e Yhc. Para la realización de las prácticas utilizaremos la implementación Hugs (<http://haskell.org/hugs/>).

Resumen de las implementaciones existentes de Haskell:

	Mensajes	Tamaño	Herramientas	Notas
Hugs	+/-	++	-	Muy utilizado para aprender Haskell. Compilación rápida. Desarrollo rápido de código.
GHC	+	-	++	El código generado es muy rápido. Posiblemente la implementación más utilizada.
NHC	?	+	++	Con posibilidad de profiling, debugging, tracing
Yhc	?	+	?	Todavía en desarrollo. .
Helium	++	++	-	Creado para la enseñanza.

Para ampliar información sobre las diferentes implementaciones visitar (<http://www.haskell.org/haskellwiki/Implementations>)

¿Qué aprenderemos de Haskell?

En este bloque teórico, veremos una pequeña introducción a Haskell, que nos servirá para poder **construir pequeñas funciones, tener una idea del modo de programar en Haskell** y una **pequeña visión sobre sus posibilidades**.

¿Qué nos quedará por aprender sobre Haskell?

- No veremos ejemplos de conexión de Haskell con otros lenguajes.

¿Cuándo la programación en C es mejor?

Por lo general, C es más rápido y dependiendo de la implementación, es posible que utilice mucha menos memoria. Haskell, necesita utilizar mucha más memoria y es más lento que C.

En aquellas aplicaciones en las que la velocidad es importante, C puede ser una mejor elección que Haskell, ya que la programación en C tiene mucho más control sobre la máquina real. El lenguaje C es más cercano a la máquina que Haskell.

¿Es posible conectar Haskell con otros lenguajes de programación?

HaskellDirect es una herramienta basada en **IDL (Interface Description Language)** que permite a los programas Haskell trabajar con componentes software. Es posible utilizar funciones C/C++ desde Haskell utilizando **Green Card** o **C->Haskell**,

SECCIÓN 1. DEFINICIÓN DE FUNCIONES/ DEFINICIONES LOCALES

```

{- ----- -}
-- DECLARACIÓN
noNegativo::(Num a, Ord a)=>a->Bool
{- PROPÓSITO
    Devuelve True si x es >= 0, False en otro caso
-}
-- DEFINICIÓN
noNegativo x = x >= 0
{-PRUEBAS
pru1 = positivo (-2.5) -- devuelve False
pru2 = positivo 0-- devuelve True
pru3 = positivo 5-- devuelve True
-}
{- ----- -}

```

Veamos los elementos necesarios para definir una función.

Lo primero que encontramos es un **comentario**.

- Para **comentar un bloque** {- -}
- Para **comentar una línea** --

Después del bloque comentado, encontramos la cabecera de la función.

```
<nombre_funcion>::<declaración_de_tipos>
```

El **nombre de la función** empieza por una letra minúscula y después puede continuar con mayúscula o minúsculas.

Para **definir los tipos de la función** podemos utilizar variables de tipos pertenecientes a alguna clase (Eq, Ord, Enum, Num, Fractional, etc.) o con tipos básicos (Int,Integer,Float, Doble, Char, etc.).

Haskell puede inferir qué tipos de datos necesita la función. Se inferirá la cabecera más genérica posible. Para inferir el tipo de dato, Haskell se sirve de los operadores y las funciones utilizadas en la definición de la función.

Para los ejercicios que se propongan en este curso, será obligatoria la definición de las cabeceras de la funciones.

Ejemplos de tipos de datos utilizados por los operadores:

a) Si utilizamos el operador “==” el tipo que utilizemos debe ser comparable (de la clase Eq).

```
Hugs> :info ==  
infix 4 ==  
(==) :: Eq a => a -> a -> Bool -- class member
```

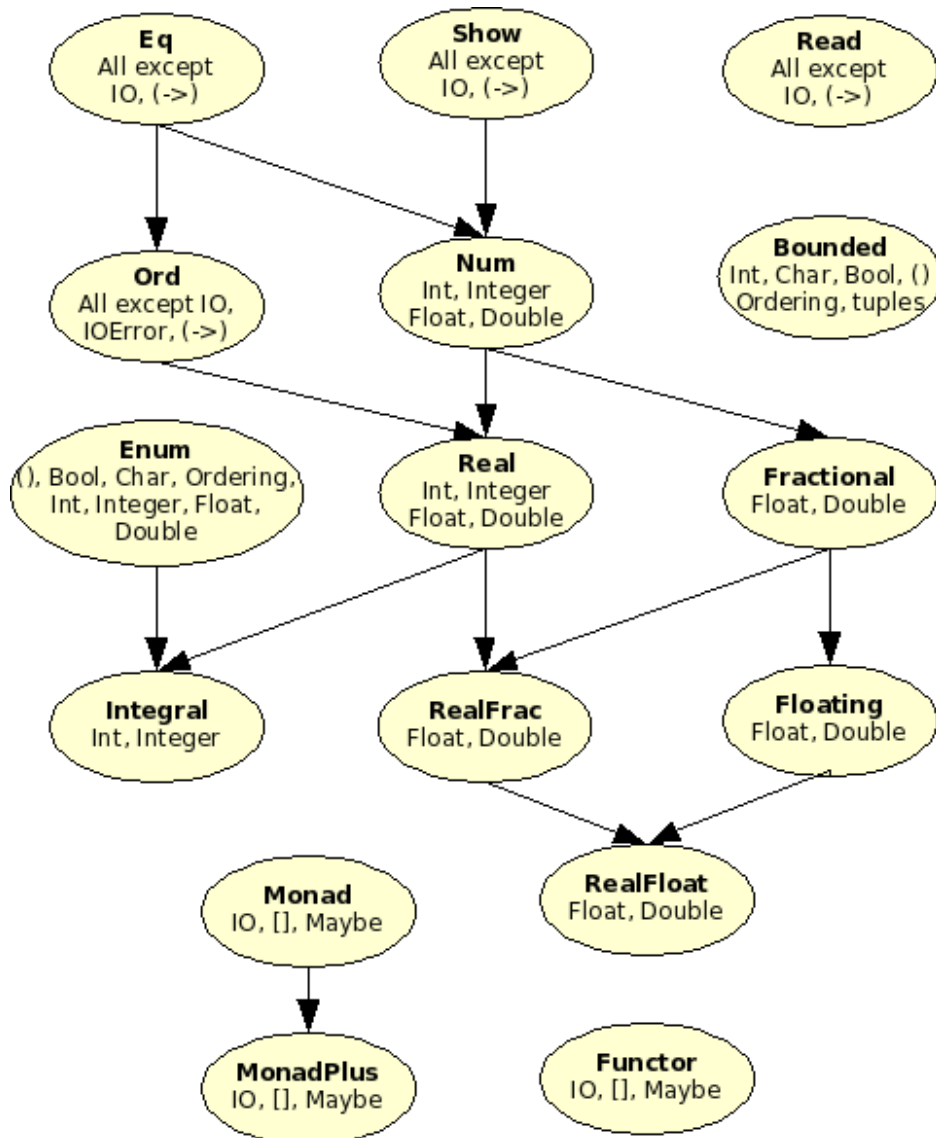
b) Si nuestra función contiene el operador “>”, el tipo debe ser ordenable (de la clase Ord)

```
Hugs> :info <  
infix 4 <  
(<) :: Ord a => a -> a -> Bool -- class member
```

c) Si nuestra función contine el operador “+”, el tipo debe ser numérico

```
Hugs> :info +  
infixl 6 +  
(+) :: Num a => a -> a -> a -- class member
```

Clasificación de tipos en Haskell:



A continuación veremos algunos **ejemplos de definición de tipos de funciones**:

a) Función identidad:

```

identidad :: a -> a
identidad x = x

```

Esta definición indica que la función recibe un tipo `a` y devuelve un tipo `a`.

b) Función iguales:

```
iguales::a->a->a->Bool
iguales x y z = x==y && y==z
```

```
ERROR file:.\iguales.hs:2 - Cannot justify constraints
in explicitly typed binding
*** Expression: iguales
*** Type: a -> a -> a -> Bool
*** Given context : ()
*** Constraints: Eq a
```

Necesitamos añadir una restricción al tipo “a”

```
iguales::Eq a=>a->a->a->Bool
iguales x y z = x==y && y==z
```

```
Main> iguales 1 1 1
True :: Bool
```

c) Función divide

```
divide::Fractional a => a -> a -> a
divide x y = x / y
```

Tenemos diferentes posibilidades a la hora de definir funciones:

- Utilizando varias ecuaciones (escribiendo cada ecuación en una línea)
- Guardas (en inglés “guards”, barreras, defensas, “[”)
- If then else
- Case
- En definiciones locales

1.1) Utilizando varias ecuaciones

```
factorial::Int->Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

1.2) Guardas

```
factorial n
  | n==0 = 1
  | n > 0 = n * factorial (n-1)
  | otherwise = error "valor negativo"
```

1.3) If then else

```
factorial n = if (n==0) then 1 else n*factorial (n-1)
```

1.4) Case

```
traduce x = case x of
  1 -> "A"
  2 -> "B"
  3 -> "C"
```

1.5) Definiciones locales

Es posible definir una función en cualquier punto de otra función:

```
divisible::Int->Int->Bool
divisible x y = resto == 0
  where resto = mod x y
```

Es muy importante que la definición esté algunos espacios a la derecha de la posición en la que empieza a definirse la función. En otro caso, Haskell mostrará un error.

SECCIÓN 2. PRIORIDAD DE OPERADORES

En la siguiente tabla se definen las prioridades de los operadores definidos en el módulo Prelude:

Notación	Prioridad	Operador
infixr	9	.
infixl	9	!!
infixr	8	^, ^^, **
infixl	7	*, /, `quot`, `rem`, `div`, `mod`
infixl	6	+, -
infixr	5	:
infixr	5	++
infix	4	==, /=, <, <=, >=, >, `elem`, `notElem`
infixr	3	&&
infixr	2	
infixl	1	>>, >>=
infixr	1	=<<
infixr	0	`, \$, \$!, `seq`

La notación **infix** indica que el operador es infijo, es decir, que se escribe entre los operandos. En el caso de encadenar operadores no se define ninguna prioridad.

```
Hugs> 1 == 1 == 1
ERROR - Ambiguous use of operator "(==)" with "(==)"
```

Necesitamos explicitar la prioridad con paréntesis:

```
Hugs> (1 == 1) == 1
ERROR - Cannot infer instance
*** Instance: Num Bool
*** Expression : (1 == 1) == 1
```

El resultado de evaluar la primera expresión es True. El error viene dado por la definición del operador “==”:

```
Hugs> :info ==
infix 4 ==
(==) :: Eq a => a -> a -> Bool -- class member
```

El operador se define para dos elementos del mismo tipo, que pertenecen a la clase **Eq** (equi-parable).

```
Hugs> (1 == 1) == True
True :: Bool
```

Infixl indica que, en caso de igualdad de precedencia se evaluará primero la izquierda:

```
Hugs> 1 - 2 - 1
-2 :: Integer
```

Infixr indica que, en caso de igualdad de precedencia se evaluará primero el operador que está más a la derecha:

```
Hugs> 2 ^ 1 ^ 2
2 :: Integer
```

Cuando utilizamos funciones y operadores en la misma expresión, tendrá mayor prioridad la función:

```
Hugs> succ 5 * 2
12 :: Integer
```

El operador que mayor precedencia tiene es la composición de funciones “.”.

```
Hugs> succ . pred 4
ERROR - Cannot infer instance
```

```
*** Instance: Enum (b -> a)
*** Expression : succ . pred 4
```

Se produce un error debido a que primero se realiza la operación `pred 4`. El resultado es el número 3. Después se intenta realizar la composición del número 3 con la función `succ`. Para poder realizar una composición de funciones son necesarias dos funciones. No es posible componer una función con un número. De esta forma, ponemos de manifiesto que entre un operador (sea cual sea) y una función, primero se evaluará la función:

```
Hugs> (succ . pred) 4
4 :: Integer
```

Como ejemplo, implementaremos la siguiente función, utilizando la función `fromIntegral`, que convierte valores de tipo enteros a un tipo numérico general, y evitando utilizar paréntesis innecesarios

$$\max(x, y) = \frac{(x+y) + |x-y|}{2}$$

```
max x y = fromIntegral (x+y + abs (x-y)) / 2.0
```

Nos quedaría definir cuál es la orden precedencia entre dos funciones.

```
Hugs> succ pred 4
ERROR - Cannot infer instance
*** Instance: Enum (a -> a)
*** Expression : succ pred 4
```

En caso de encadenamiento de funciones con la misma prioridad, la evaluación se realiza de izquierda a derecha. Primero se intenta evaluar `succ pred`, y como la función `succ` está definida sólo para tipos enumerables, al intentar evaluar el sucesor de `pred` se produce un error.

```
Hugs> succ (pred 4)
4 :: Integer
```


SECCIÓN 3. EVALUACIÓN PEREZOSA

Los lenguajes que utilizan esta técnica sólo evalúan una expresión cuando se necesita:

```
soloPrimero::a->b->a
soloPrimero x _ = x

Main> soloPrimero 4 (7/0)
4 :: Integer
```

El subrayado “_” denota que se espera un parámetro pero que no se necesita nombrarlo ya que no se va a utilizar en el cuerpo de la función.

La expresión 7/0 en Haskell tiene el valor Infinito. La evaluación de la expresión anterior provocaría un error en la mayoría de los lenguajes imperativos, sin embargo en Haskell no se produce un error debido a que el segundo argumento no llega a evaluarse por no ser necesario.

SECCIÓN 4. FUNCIONES DE ORDEN SUPERIOR

Las funciones de orden superior son aquellas que reciben una o más funciones como argumentos de entrada y/o devuelven una función como salida.

Un ejemplo de función de orden superior es la función `map` implementada en el módulo `Prelude`. Esta función recibe una función y una lista y aplica la función a cada elemento de la lista:

```
map:: (a->b)->[a]->[b]
map _ [] = []
map f (cab:resto) = f cab : map f resto

Main> map succ [1,2,3]
[2,3,4] :: [Integer]
```

La función `f` que pasamos como argumento a `map` debe cumplir una restricción: el tipo de dato que recibe debe ser el mismo de los elementos de la lista.

SECCIÓN 5. CURRIFICACIÓN

El proceso de currificación toma nombre de Haskell Brooks Curry cuyos trabajos en lógica matemática sirvieron de base a los lenguajes funcionales.

Consiste en realizar la llamada a una función utilizando sólo algunos de los parámetros que están más a la izquierda.

```
suma :: Int -> Int -> Int -> Int
suma x y z = x + y + z
```

Podemos hacer las siguientes llamadas:

- a) suma 1 -> devuelve una función que recibe dos enteros y devuelve otro.
- b) suma 1 2 -> devuelve una función que recibe un entero y devuelve otro.
- c) suma 1 2 3 -> devuelve el entero 6

Podemos utilizar la función currificada como argumento de otra función de orden superior:

```
Main> map (suma 1 1) [1,2,3]
[3,4,5] :: [Int]
```

SECCIÓN 6. COMPOSICIÓN DE FUNCIONES

$$f.g \ (x) = f(g(x))$$

Haskell dispone de un operador para componer funciones. Se trata del operador “.”:

```
Main> :info .
infixr 9 .
(.) :: (a -> b) -> (c -> a) -> c -> b
```

Para poder componer dos funciones, éstas deben cumplir una serie de restricciones. Si observamos la cabecera del operador “.”, podemos comprobar que el segundo argumento es una función $(c \rightarrow a)$, que el primer argumento es una función $(a \rightarrow b)$, que el valor al que se quiere aplicar la composición es de tipo c y que el valor de salida es de tipo c .

La restricción más importante es que si queremos hacer la composición $f.g$, el tipo de salida de la función g , debe ser el de entrada de la función f .

Veamos un ejemplo:

```
Main> ((==True).(<0)) 5
False :: Bool
```

La función (<0) devuelve un tipo Bool, que es el que recibe la función $(==True)$. El resultado de la composición de las dos funciones es de tipo Bool.

PROGRAMACIÓN FUNCIONAL CON HASKELL

0. Introducción

1. Definición de funciones

2. Prioridad de operadores

3. Evaluación perezosa

4. Funciones de orden superior

5. Currificación

6. Composición de funciones

7. Listas

8. Patrones

9. Tuplas

10. Recursividad

SECCIÓN 7. LISTAS

Haskell proporciona un mecanismo para definir fácilmente listas:

- a) [1..10]
- b) [15..20]
- c) [15..(20-5)]
- d) ['q'..'z']
- e) [14..2]

Notación extendida de listas

La definición de una lista utilizando esta notación consta de tres partes: 1) generador, 2) restricciones (puede que no haya ninguna) y 3) transformación. El generador produce elementos de una o varias listas, las restricciones filtran algunos elementos de los generados y la transformación utiliza los elementos seleccionados para generar una lista resultado.

[(x,True)	x <- [1 .. 20],	even x, x < 15]
_____	_____	_____
Transformación	Generador	Restricciones

En este ejemplo, `x <- [1 .. 20]` es el generador, las restricciones son `even x` y `x < 15`, y `(x, True)` es la transformación. Esta expresión genera una lista de pares `(x, True)` donde `x` está entre 1 y 20, es par y menor que 15. (even es una función definida en el módulo Prelude standard de Haskell).

Funciones más usuales sobre listas

Cabecera de la función	Explicación
<code>(:) :: a -> [a] -> [a]</code>	Añade un elemento al principio de la lista
<code>(++) :: [a] -> [a] -> [a]</code>	Concatenar dos listas
<code>(!!) :: [a] -> Int -> a</code>	Devuelve el elemento n-ésimo
<code>null :: [a] -> Bool</code>	Devuelve True si lista == []
<code>length :: [a] -> Int</code>	Devuelve la longitud de una lista
<code>head :: [a] -> a</code>	Devuelve el primer elemento
<code>tail :: [a] -> [a]</code>	Todos menos el primero
<code>take :: Int -> [a] -> [a]</code>	Toma los primeros n elementos
<code>drop :: Int -> [a] -> [a]</code>	Elimina los n primeros elementos
<code>reverse :: [a] -> [a]</code>	Invierte el orden de los elementos
<code>zip :: [a] -> [b] -> [(a,b)]</code>	Crea una lista de pares zip "abc" "123" >>
<code>unzip :: [(a,b)] -> ([a],[b])</code>	A partir de una lista de tuplas genera dos listas
<code>sum :: Num a => [a] -> a</code>	Suma los elementos de una lista
<code>product :: Num a => [a] -> a</code>	Multiplifica los elementos de una lista

Consulte información sobre las funciones definidas para listas en el módulo Prelude. <http://www.cs.ut.ee/~varmo/MFP2004/PreludeTour.pdf>

SECCIÓN 8. PATRONES

Veamos una implementación posible de la función `map`:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (cab:resto) = f cab : map f resto
```

En el ejemplo anterior:

`[]` es el patrón que empareja con una lista vacía.
`(cab:resto)` empareja con una lista (el operador “`:`” devuelve una lista) cuyo primer elemento es “`cab`” y el resto es “`resto`”.

```
Hugs> :info :
infixr 5 :
(:) :: a -> [a] -> [a] -- data constructor
```

`f` empareja con cualquier valor del tipo esperado (por la cabecera de la función `map`, `f` debe ser una función) y `f` se instancia a esa función.

`_` empareja con cualquier valor del tipo esperado, sin embargo, no se hace una asignación a la variable “`_`”: sencillamente se ignora el valor.

Luego el emparejamiento de patrones es una forma de asignar valores a variables y, al mismo tiempo, una forma de dividir expresiones en sub-expresiones.

¿Qué podemos utilizar como patrón?

Existe un conjunto dado de patrones de emparejamiento, de modo que no es posible hacer emparejamientos con todas las construcciones del lenguaje que nos parezcan posibles sino sólo con aquellas que nos permita Haskell. Por ejemplo:

```
eliminarTres ([x,y,z] ++ resto) = resto
```

Esta definición de la función `eliminarTres` provoca un error:

```
ERROR file:.\eliminarTres.hs:2 - Syntax error in input (unexpected symbol
"++")
```

El problema surge porque el operador “`++`” no está permitido en los patrones. En los patrones sólo se permiten constructores y constantes (1,2, True, False, etc). Un constructor tiene la forma:

```
data Bool = True | False
data [a] = [] | a:[a]
Main> :info :
```

```
infixr 5 :
(:) :: a -> [a] -> [a] -- data constructor
```

Cuando solicitamos información sobre un operador Haskell indica si se trata de un constructor.

[] y “:” son constructores para listas. La función eliminarTres puede implementarse así:

```
Eeliminar :: [a] -> [a]
EliminarTres (_:_:_:resto) = resto
```

Es importante el tipo del constructor, pero no el número. En el ejemplo anterior se utiliza el operador “:” como patrón tres veces.

Excepción

Hay una excepción a la regla del uso de los constructores en patrones. Se trata del patrón $(n+k)$.

```
predecesor :: Int -> Int
predecesor (n+1) = n
```

No podemos utilizar cualquier valor para que empareje con $(n+k)$. Sólo podemos utilizar enteros (Int ó Integer). Además, el valor tiene que ser mayor o igual que k.

```
Main> predecesor 0
{throw (PatternMatchFail (_nprint 0 (predecesor 0) []))} :: Integer
```

Alias de patrones

Los patrones se utilizan en ocasiones para dividir expresiones. En estos casos, puede ser interesante tener acceso a la expresión completa. Para ello, utilizaremos los alias de patrones. Escribiremos `nombre_variable@<patrón>`. Por ejemplo, una función que elimina los caracteres en blanco que se encuentran al principio de una cadena de caracteres:

```
quitaBlancosPrinc :: String -> String
-- equivalente a quitaBlancosPrinc :: [Char] -> [Char]
quitaBlancosPrinc cadena@(cab:resto)
  | cab == ' ' = quitaBlancosPrinc resto
  | otherwise = cadena
```

o este otro, una versión del factorial utilizando un alias de patrón:

```
factorial :: Integral a => a -> a
factorial 0 = 1
factorial m@(n+1) = m * factorial n
```

Utilizaremos los patrones en ecuaciones, cláusulas `let` y `where`, expresiones `case` y listas.

SECCIÓN 9. TUPLAS

Podemos agrupar expresiones de distinto tipo en una tupla. Por ejemplo:

- a) `(1,2)`
- b) `('a',1,"Hola")`
- c) `((1,2), [3,4], (0,'a'))`
- d) `((+), 7, (*))`

Veamos de qué tipo son las tuplas anteriores:

```
Main> :t (1,2)
(1,2) :: (Num a, Num b) => (b,a)
Main> :t ('a',1,"Hola")
('a',1,"Hola") :: Num a => (Char,a,[Char])
Main> :t ((+),7,(*))
((+),7,(*)) :: (Num a, Num b, Num c) => (c -> c -> c,b,a -> a -> a)
Main> :t ((1,2), [3,4], (0,'a'))
((1,2), [3,4], (0,'a')) :: (Num a, Num b, Num c, Num d) =>
((c,d), [b], (a,Char))
```

Algunos ejemplos de funciones con tuplas:

- a) `primero (x,y) = x`
- b) `primero2 (x,y,z) = x`

```
Main> primero (1,2,3)
ERROR - Type error in application
*** Expression      : Main.primero (1,2,3)
*** Term            : (1,2,3)
*** Type            : (c,d,e)
*** Does not match : (a,b)

Main> primero2 (1,2,3)
1 :: Integer
```

Definición de tipos con tuplas

Veamos el siguiente ejemplo:

```
type Entrada = (Persona, Edad, Telefono)

type Persona = String
type Edad = Int
type Telefono = String
```



```
type Listin = [Entrada]

encontrar::Listin -> Persona -> [Telefono]

encontrar lista persona = [telef | (per, edad, telef) <- lista,
persona == per]

Main> encontrar [("Pedro", 20, "636000000"), ("Juan",
21,"607222222"), ("Alberto", 24, "635111111")] "Pedro"

["636000000"] :: [Main.Telefono]

Main> encontrar [("Pedro", 20, "636000000"), ("Juan",
21,"607222222"), ("Alberto", 24, "635111111"), ("Pedro", 20,
"635444444")] "Pedro"

["636000000","635444444"] :: [Main.Telefono]
```


PROGRAMACIÓN FUNCIONAL CON HASKELL

0. Introducción

1. Definición de funciones

2. Prioridad de operadores

3. Evaluación perezosa

4. Funciones de orden superior

5. Currificación

6. Composición de funciones

7. Listas

8. Patrones

9. Tuplas

10. Recursividad

SECCIÓN 10. RECURSIVIDAD

En Haskell no tenemos posibilidad de definir bucles. La forma de “iterar” es utilizando la recursividad. Una función recursiva es aquella que en su definición contiene una llamada a sí misma.

La recursividad se apoya en el principio de inducción. Este principio es ampliamente utilizado en matemáticas para demostrar que se cumple una propiedad para cualquier valor del ámbito que se esté tratando.

Principio de inducción:

- (1) La afirmación P es cierta para un valor inicial n_0 (“caso base”)
- (2) P será cierta para un valor $n > n_0$, si es cierta para el valor anterior a n , es decir, si P es cierta para $(n-1)$ entonces lo será para n .

Podemos utilizar el principio de inducción para definir los números naturales:

El número 1 es natural.
 n es natural si $n-1$ es natural.

En Haskell:

```
natural 1 = True
natural n = natural (n-1)
Main> natural 5
True :: Bool
```

;;El principio de inducción funciona!! -> Vamos a utilizarlo.

pero ...

```
Main> natural (-3)
```

```
;;;No termina!!!
```

Igual sucede con el número “3.5”. No hemos tenido en cuenta que, el valor tiene que ser mayor que el primero. Una nueva versión:

```
natural :: (Num a, Ord a) => a -> Bool
natural 1 = True
natural n
  | n > 1      = natural (n-1)
  | otherwise = False

Main> natural (-1)
False :: Bool
```

```
Main> natural 3.5
False :: Bool
```

Recomendación importante: No seguiremos mentalmente la secuencia de llamadas recursivas para resolver los problemas. **Nos centraremos en definir bien el caso base (el primer elemento que cumple la propiedad) y la relación de un valor n con el anterior.**

Ejemplo de recursividad con listas

Trasladaremos la misma idea a la resolución de un problema recursivo con listas. Definiremos recursivamente una función que devuelva el número de elementos de una lista.

Para este problema, el primer caso no es el 1. **¿Cuál es el primer caso cierto conocido para listas?** En este caso (y en la mayoría de los problemas de listas) el primer caso conocido corresponde con la lista vacía. Es la lista más pequeña, al igual que el 0 es el natural más pequeño.

```
numElementos [] = 0
```

Esta ecuación cumple la propiedad: **es cierto que una lista vacía tiene cero elementos.**

Veamos qué pasa con n . Ahora no tenemos un número n y un número anterior $n-1$ como sucedía con los naturales. El ejemplo de los números naturales dice que, si se cumple la propiedad para $n-1$, se cumplirá para n .

¿Cómo construimos el valor n y el $n-1$ con listas?

Utilizaremos el patrón “cab:resto” que separa el primer elemento del resto.

Nuestro elemento n será la lista completa, y el elemento $n-1$ será la lista con un elemento menos (el contenido de la variable resto).

Si numElementos resto entonces numElementos (cab:resto)

Si `numElementos (resto)` se cumple, devolverá el número de elementos que contiene el resto de la lista (todos menos el primero).

Por el principio de inducción sabemos que:

Si `numElementos resto` es cierto (devuelve el valor correcto) `numElementos (cab:resto)` también lo será.

¿Cuánto vale numElementos (cab:resto)?

Para saber cuánto vale, utilizaremos el valor devuelto por numElementos resto

numElementos (cab:resto) =	numElementos resto
_____	_____
Queremos conocer este	Supondremos que sabemos
valor	cuánto vale

¿Qué cambio tengo que hacer en la salida parcial, para obtener la salida total?

Ésta será la pregunta que haremos siempre para resolver un problema recursivo.

En este caso la transformación es sencilla. Si sabemos cuántos elementos tiene una lista a la que le falta un elemento, para saber cuántos tiene la lista completa bastará con sumar uno.

numElementos (cab:resto) = 1 +	numElementos resto
_____	_____
Queremos conocer este	Supondremos que sabemos
valor	cuánto vale

Veamos otro ejemplo algo más complicado:

Deseamos implementar la función `foldl` definida en Haskell Prelude. Esta función se define para listas no vacías. Podemos encontrar un ejemplo en el documento “A tour of the Haskell Prelude” <http://www.cs.ut.ee/~varmo/MFP2004/PreludeTour.pdf>

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Descripción: Une los elementos de una lista utilizando un operador binario y un elemento inicial, utilizando asociación por la izquierda. Cuando la lista de entrada está vacía, devuelve el elemento inicial.

Dediquemos algún tiempo a entender qué hace la función. Para esto podemos realizar algún ejemplo más:

```
Hugs> foldl (^) 1 []
1 :: Integer

Hugs> foldl (^) 1 [2]
1 :: Integer
(1^2)

Hugs> foldl (^) 1 [2,3]
1 :: Integer
(1^2)^3
```

Ahora que tenemos algo más claro que hace la función. Empecemos con la implementación:

- 1) **Empezaremos por el caso base.** Suele ser el caso más sencillo. Debe ser cierto por sí sólo, sin tener en cuenta lo que escribamos después. Suele estar descrito en la definición de la función o es muy evidente, como el caso de la función `numElem []` cuyo resultado es 0.

En este caso:

```
foldl f el [] = el
```

- 2) **Caso recursivo.** Empezaremos planteando el caso recursivo de la siguiente forma:

```
foldl f el (cab:resto) =  
foldl f el resto
```

Para problemas de listas, utilizaremos en la mayoría de los casos la misma técnica, separar la cabeza del resto de la lista utilizando el operador “:”. Ahora, nos centraremos en averiguar qué devuelve la función llamada con una lista de un elemento menos como parámetro. Para ello lo mejor es escribir un ejemplo:

```
foldl (^) 1 [2,3,4] -> (((1^2)^3)^4)
```

con un elemento menos:

```
foldl (^) 1 [3,4] -> ((1^3)^4)
```

Ahora nos preguntaremos:

¿Qué transformación tengo que hacer en el resultado de la función que tiene un elemento menos para que se convierta en el resultado de la lista completa?

Buscamos las diferencias entre la salida parcial y la total.

```
((1^2)^3)^4 -> Salida total
```

```
(( 1 ^3)^4) -> Salida parcial
```

Observando las dos salidas, vemos que la diferencia está en que la salida parcial contiene un “1” y la total contiene “1^2”, siendo 2 el primer elemento de la lista. El resto es igual. Parece que la diferencia sólo está en el segundo elemento de la función. El cambio que necesita la llamada a la función parcial es cambiar el elemento “el” por “el^cab”. De una forma más genérica: “f el cab”

Completaremos la llamada recursiva con el cambio. Donde escribíamos “el”, escribiremos “f el cab”.


```
foldl f el (cab:resto) =
```

```
foldl f el resto
  | ____ |
cambiaremos el por
  f el cab
```

```
foldl f el (cab:resto) =
```

```
foldl f (f el cab) resto
```

Finalmente la función quedará como sigue:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f el [] = el
foldl f el (cab:resto) =
foldl f (f el cab) resto
```

```
Main> foldl2 (^) 1 [2,3,4]
1 :: Integer
```

```
Main> foldl (++) "com" ["po", "ner"]
"componer" :: [Char]
```

¿Cómo comprobar si una función recursiva es correcta?

Debo responder a las siguientes preguntas:

- 1) ¿El caso base es cierto?
- 2) ¿El caso recursivo tiende al caso base?
- 3) ¿Están contemplados todos los casos una única vez?
- 4) ¿A partir de la salida recursiva $(n-1)$ puedo construir la salida general (n) ? Si/No.
¿Se han realizado los cambios de forma correcta?

En general, podemos decir que:

“Si las respuestas a estas preguntas son afirmativas la función recursiva funcionará”.

Sí. Pero, ¿cómo se llega a la solución?

Respuesta: Preguntando siempre al problema con un elemento menos.

Lo que sucede es que se encadenan una serie de preguntas hasta que se alcanza el caso base. A partir de la solución que aporta el caso base se construyen el resto de soluciones parciales hasta llegar a la primera de las preguntas.

```

num_elem [1,2,3] --> num_elem [2,3] --> num_elem [3] --> num_elem []
                                     |
1+1+1+0 <----- 1+1+0 <----- 1+0 <----- 0

```

¿Qué sucede si alguna de las respuestas es negativa?

- 1) Si el caso base no fuese cierto, se construirá la solución a partir de una premisa errónea.
- 2) Si el caso recursivo no tendiese al caso base, no podría asegurar que el problema termine.
- 3) Si alguno de los casos no estuviese contemplado, la secuencia de preguntas se romperá y no se alcanzará al caso base.

4) Si a partir de la llamada recursiva no puedo construir la solución general, no será posible encontrar la solución.

Si no construimos bien la solución general a partir de la solución parcial $(n-1)$, el resultado no será correcto.

PROGRAMACIÓN LÓGICA CON PROLOG

0. Introducción

1. Unificación

2. Tipos de datos

2.1 Listas

2.2 Árboles

2.3 Grafos

3. Control de la ejecución

4. Problemas de estados

SECCIÓN 0. INTRODUCCIÓN

Lógica Proposicional:

Utiliza afirmaciones simples (proposiciones). Escribiremos reglas para indicar que si se produce un evento (antecedente) entonces se producirá una consecuencia (consecuente).

```
Si llueve -> se regarán las plantas
|_____| |_____|
Antecedente      Consecuente
|_____|
                     REGLA
```

Con la regla anterior, no se indica que *esté lloviendo*. Para afirmar que llueve necesitamos un hecho.

```
llueve
|_____|
      HECHO
```

Una cláusula es una regla o un hecho.

Con cláusulas crearemos una base de conocimiento (una representación de lo que conocemos). Una vez definida la *base de conocimiento*, realizaremos consultas.

Un “*programa lógico*” se compone de dos elementos:

“Programa lógico” = Base de conocimiento + Consultas

Sintaxis Prolog: Escribiremos las reglas empezando por el consecuente. Todas las cláusulas terminan con un punto “.” Escribimos la regla *Si llueve -> se regaran las plantas* de esta forma:

```
se_riegan_las_plantas :- llueve.
```

Primer programa Prolog utilizando sólo proposiciones:

```
% Mi primer programa Prolog
se_riegan_las_plantas :- llueve.
llueve.
```

Para afirmar que llueve, escribiremos el siguiente hecho: llueve.

Debemos realizar consultas, para obtener algún resultado:

```

1 ?- consult('d:/lluvia.pl').
% d:/compiled 0.00 sec, 672 bytes

Yes
2 ?- llueve.

Yes
3 ?- se_riegan_las_plantas.

Yes

```

En la línea 1 cargamos el programa “lluvia.pl” utilizando el procedimiento `consult`. Prolog indica el tiempo de compilación y los bytes compilados. Ante cualquier consulta, salvo error, Prolog responde “Yes” o “No”. Dependiendo de la versión de Prolog, el mensaje “Yes” puede sustituirse por la solución de la consulta.

En la línea 3 preguntamos si llueve. La respuesta es “Yes”. La siguiente pregunta es `¿se_riegan_las_plantas?`. La respuesta es “Yes”.

Imaginemos que deseamos expresar lo siguiente: *Si llueve entonces, se riegan los tomates y las lechugas y las fresas.*

Utilizando lógica proposicional escribiríamos lo siguiente:

```

Si llueve -> se_riegan_los_tomates ^ se_riegan_las_lechugas ^ se_riegan las_
fresas.

```

Para facilitar la representación del conocimiento y la asociación de conceptos se utilizan los predicados.

Lógica de Primer Orden o Cálculo de Predicados de Primer Orden:

Extiende la lógica proposicional empleando variables, predicados y cuantificadores de variables.

Reescribiremos la regla anterior utilizando el predicado `se_riegan(Plantas)`. Este predicado representa que una determinada planta se riega.

```

Si llueve -> se_riegan(tomates) ^ se_riegan(lechugas) ^ se_riegan(fresas)

```

Esta nueva definición utilizando predicados permitiría preguntar *¿Qué se riega?* y Prolog respondería, tomates, lechugas y fresas.

Sin embargo, esta regla escrita de esta forma no se puede implementar directamente en Prolog ya que no es una *Cláusula de Horn*.

Cláusulas de Horn. Una *cláusula de Horn* es aquella que está formada por una conjunción de cero o más términos en el antecedente y un único término en el consecuente.

```
Si llueve -> se_riegan(tomates)
|_____|   |_____|
Término   Término
```

Para resolver si una determinada consulta (**fórmula**) es cierta o falsa, se utiliza un algoritmo de resolución. Este algoritmo de resolución es **decidible** (siempre termina con una solución válida) si las cláusulas son todas *Cláusulas de Horn*. En otro caso, no se podría garantizar que el algoritmo de resolución terminara.

En algunos casos podemos encontrar una equivalencia para que una cláusula se escriba como cláusula de Horn.

1. $a \wedge b \rightarrow c$ C. de Horn

2. $a \rightarrow b \wedge c$ No es C. de Horn \Rightarrow Equivalencia $\begin{cases} a \rightarrow b \\ a \rightarrow c \end{cases}$

3. $a \vee b \rightarrow c$ No es C. de Horn \Rightarrow Equivalencia $\begin{cases} a \rightarrow c \\ b \rightarrow c \end{cases}$

4. $a \rightarrow b \vee c$ No es C. De Horn \Rightarrow No existe equivalencia

Nuestro programa sobre la lluvia y el riego utilizando cláusulas de Horn quedaría así:

```
% Lluvia, riego y vegetales usando predicados

se_riegan(tomates):- llueve.
se_riegan(lechugas):- llueve.
se_riegan(fresas):- llueve.
llueve.
```

Podemos preguntar entonces: ¿Qué se riega?

Sintaxis Prolog: Escribiremos las variables empezando por mayúscula.

```
2 ?- se_riegan(Vegetales).  
Vegetales = tomates ;  
Vegetales = lechugas ;  
Vegetales = fresas  
3 ?
```

Para obtener todas las respuestas, escribimos “;” después de cada respuesta.

Hipótesis del mundo cerrado:

Lo no definido es falso. Con el fin de simplificar la definición de la base de conocimiento, se llega al acuerdo de responder como falso a todo lo no definido. Si hacemos alguna pregunta sobre algún valor que no existe Prolog responderá “No”.

```
3 ?- se_riegan(manzanas).  
  
No
```


PROGRAMACIÓN LÓGICA CON PROLOG

0. Introducción

1. Unificación

2. Tipos de datos

2.1 Listas

2.2 Árboles

2.3 Grafos

3. Control de la ejecución

4. Problemas de estados

SECCIÓN 0. INTRODUCCIÓN (CONTINUACIÓN)

Variables

Escribiremos las variables **empezando por mayúscula**. No es necesario declarar una variable para poder utilizarla. Una **variable en Prolog puede instanciarse con cualquier tipo de dato** (árbol, lista, grafo, etc.). **Una variable, una vez que se instancia, no cambia su valor.**

Ejemplos de variables:

- 1) Persona
- 2) Lista
- 3) X
- 4) Arbol
- 5) _

Ejemplos de uso:

```
?- X = 1, X = 2.  
No  
  
?- Persona = manuel.  
Persona = manuel  
Yes  
  
?- Lista = [1,2,3]  
Lista = [1,2,3]  
  
? Lista = manuel  
Lista = manuel  
Yes
```

En el último de los ejemplos anteriores, observamos que Prolog no hace ningún tipo de comprobación a la hora de asignar valores a las variables. Podemos asignar a una variable llamada “Lista” (que debería contener una lista) cualquier valor. Si la variable Lista está libre (no se le asignó un valor previamente), será posible realizar la siguiente unificación Lista=manuel y Prolog responderá que es cierto si Lista se instancia al valor “manuel”.

Variables Libres e Instanciadas

Diremos que **una variable está libre si aún no se asignó un valor** a través de una unificación.

Diremos que **una variable está instanciada si ya se le asignó un valor** a través de una unificación. Esa variable no podrá modificar su valor.

```
?- X=1, X=2.
No
```

Por legibilidad, **debemos asignar a las variables nombres representativos de lo que contendrán**: si las variables contienen números podemos llamarlas X, Y, Z, etc.; si las variables contienen listas, las llamaremos Lista, ListaR, etc. Esto nos ayudará a entender más fácilmente los programas.

Ámbito de una variable

Es común en otros lenguajes de programación que si una variable se define dentro de una función, la variable puede ser referenciada desde cualquier punto de la función. Decimos que el ámbito de la variable es la función en la que está definida.

En Prolog el ámbito de una variable está restringido a una cláusula.

Más allá de esa cláusula, la variable no puede ser referenciada. Esta es una diferencia importante de Prolog respecto a lenguajes de programación.

Véase el siguiente ejemplo:

```
suma(X,Y,Z):- X \= Y, Z is X + Y.
suma(X,X,Z):- Z is X + X.
```

Código 1: Primera versión del predicado suma

En el ejemplo anterior la variable X y la variable Z aparecen en las dos cláusulas, sin embargo no existe conexión entre ellas. Podemos cambiar el nombre de las variable y el programa funcionará de la misma forma.

Esto permite que, observando únicamente una cláusula, podamos saber si es correcta o no, independientemente de lo que se escriba a continuación.

```
suma(X,Y,Z):- X \= Y, Z is Y + Z.
suma(N,N,R):- R is N + N.
```

Código 2: Segunda versión del predicado suma

Esta segunda versión de la implementación del predicado suma, funciona exactamente igual que la anterior.

Variable anónima (“_”)

Cuando queramos indicar que un argumento tenga un valor, pero no nos importa cuál sea ese valor, utilizaremos la *variable anónima*, que se representa con el carácter “_”.

La variable anónima tiene un sentido de “*todo*” o “*cualquier valor*”.

Por ejemplo: Si quiero indicar, que cuando llueve se riegan *todas* las plantas puedo escribir:

```
se_riegan(Planta):- llueve.
llueve.
```

Al ejecutar el programa anterior, Prolog mostrará el siguiente mensaje:

```
Warning (d:/llueve4.pl:1):
Singleton variables: [Planta]
```

Este mensaje indica que una variable aparece una única vez en una cláusula, lo que significa que no existe ninguna restricción sobre el valor que puede tomar la variable y, por tanto, no es necesario darle un nombre. Este mensaje es sólo un aviso con lo que el programa se podrá ejecutar, sin embargo, debemos corregirlos. Si nos acostumbramos a verlos, cuando nos equivoquemos realmente al escribir el nombre de una variable, es muy posible que no hagamos ningún caso al aviso.

Esto puede suceder por dos razones:

1) Que sea correcto. Realmente quiero indicar que haya una variable pero no utilizaré su valor, en cuyo caso escribiremos

```
se_riegan(_):-llueve.
```

2) Que haya escrito mal el nombre de una variable, en cuyo caso debo corregir el error:

```
crece(Plamta):- se_riega(Planta).
se_riega(_):- llueve.
llueve.
```

Provocará el siguiente aviso.

```
Warning (d:/llueve4.pl:1):
Singleton variables: [Plamta, Planta]
```

SECCIÓN 1. UNIFICACIÓN

Diremos que dos términos unifican:

- 1) Si no tienen variables, unifican si son idénticos.
- 2) Si tienen variables, unifican si es posible encontrar una sustitución de las variables de forma que lleguen a ser idénticos.

Ejemplos:

?- X=1.

X=1

Yes

?- X = mujer(maria).

X = mujer(maria)

Yes

?- X = 2 + 3.

X=2+3

Yes

?- X=1, X=2, X=Y.

No

?- X=2+X.

X=+**

Yes

?- (1+2) = 1+2.

Yes

? 1+1 = +(1,1).

Yes

? (1+1)+1 = 1+(1+1).

No

Operadores

Los operadores más utilizados en Prolog son "=", "==", "is", "=:=", "<", "<=", ">=".

"=" unificador:

El operador de unificación no evalúa operaciones aritméticas. Antes de unificar una variable a un valor diremos que la variable está "libre" o "sin instanciar". Tras unificar un valor a una variable, la variable deja de estar libre y diremos que la variable está "instanciada".

```
?- X = 1+1.
```

```
X = 1+1.
```

```
Yes
```

"==" comparador de identidad:

Es verdadero si los valores comparados son exactamente *el mismo valor*.

```
?- 2 == X.
```

```
No.
```

```
?- 2 == 2.
```

```
Yes.
```

```
?- 2 == 1+1.
```

```
No.
```

"is" evaluador de expresiones aritméticas:

Evalúa a la derecha y unifica con la izquierda. Para poder realizar una operación aritmética, todas las variables que estén a la derecha del operador tienen que estar instanciadas. En otro caso se producirá un error.

```
?- X is 1+1.
```

```
X = 2
```

"=:" evaluador y comparador de expresiones aritméticas:

Evalúa a derecha e izquierda y es cierto si el resultado de las evaluaciones es idéntico.

```
?- 2 =:= 1+1.
```

```
Yes
```

Ejemplos:

```
?- X is 2 + 3, X = 5.
```

```
X=5
```

```
Yes
```

```
?- X is 2 + 3, X = 2 + 3.
```

```
No
```

```
?- 2 + 3 is X.
```

```
ERROR: is/2: Arguments are not sufficiently instantiate
```

```
?- X is 5, 6 is X + 1.
```

```
X=5
```

```
Yes
```

```
?- X = 5, 6 = X + 1.
```

```
No
```

```
?- 6 is X + 1, X = 5.
```

```
ERROR: is/2: Arguments are not sufficiently instantiate
```

Tabla de operadores

Precedencia	Notación	Operador
1200	xfx	<code>:-</code>
1000	xfx	<code>,</code>
900	fy	<code>\+</code>
700	xfx	<code><, =, =.., =@=,</code> <code>:=, =<, ==, =\=,</code> <code>>, >=, @<, @=<, @>,</code> <code>@<, @>=, \=, \==,</code> <code>is</code>
500	yfx	<code>+, -, xor</code>
500	fx	<code>+, -, ?, \</code>
400	yfx	<code>*, /, mod, rem</code>
200	xfx	<code>*</code>
200	xfy	<code>^</code>

- Las operaciones cuyo operador tiene un número de precedencia menor se realizarán antes.

- yfx: asociativo por la izquierda. (Ejemplo: `X is 2/2/2`, `X=0.5`).

- xfy: asociativo por la derecha. (Ejemplo: `X is 2^1^2`, `X=2`).

Para obtener un listado completo de operadores ejecutar `apropos(operators)`.

PROGRAMACIÓN LÓGICA CON PROLOG

0. Introducción

1. Unificación

2. Tipos de datos

2.1 Listas

2.2 Árboles

2.3 Grafos

3. Control de la ejecución

4. Problemas de estados

SECCIÓN 1. TIPOS DE DATOS

Aunque Prolog no es un lenguaje tipado, sí que existen determinados valores con peculiaridades de notación y predicados asociados que los diferencian del resto de valores convirtiéndolos en una especie de tipo de datos. En todo caso, necesitaremos delimitar grupos de valores que actúen como tipos de datos diferenciados en nuestros programas. En esta sección veremos cómo representar y utilizar algunos importantes tipos de datos con Prolog.

2.1 Listas

Notaremos las listas entre corchetes, separando los elementos por comas. Éstas son algunas características de las listas en Prolog:

Las listas **contienen términos (variables, constantes o estructuras) en general**, con lo que no se restringe a que contengan valores de uno de esos tipos en concreto o con una notación uniforme (sólo números, sólo variables, etc.). **Es posible anidar listas.**

Notaremos **la lista vacía** (“[]”) con un corchete de apertura y otro de cierre.

Ejemplos de listas:

```
[1,2,3,4]
[]
[a,b,c,d]
[1,'a',b,x,[1,2,3]]
[[1,2],[3,4],[5,6]]
[se_riegan(Plantas),[3,4],pedro]
```

Nota: Si queremos que una variable se instancie a una lista, lo haremos de la siguiente forma:

```
?- Lista = [1,2,3,4]
Lista=[1,2,3,4]
```

Recuerde que, una variable libre se puede instanciar con cualquier tipo de dato. Un error frecuente es el siguiente:

```
?- [Lista] = [1,2,3,4]
No
```

Con [Lista] hacemos referencia a una lista que contiene un único término.

Operador “|”

Utilizaremos el operador “|” (barra vertical), para separar el primer elemento de una lista del resto. Este operador es muy útil para construir predicados recursivos y lo utilizaremos frecuentemente.

Ejemplos:

```
1 ?- [Cabeza|Resto] = [1,2,3]
Cabeza=1
Resto=[2,3]
```

```
2 ?- [Cabeza|Resto] = [1]
Cabeza=1
Resto=[]
```

```
3 ?- [C1,C2|Resto] = [1,2,3]
C1=1
C2=2
Resto=[3]
```

```
4 ?- [_|_]= []
No
```

```
5 ? [Cabeza|Resto] = [[1,2,3]]
Cabeza=[1,2,3]
Resto = []
```

Nota: Hemos de tener en cuenta que **el resto siempre es una lista**. En el segundo de los ejemplos anteriores $[C|R]=[1]$, es posible la unificación, siendo el resto una lista vacía “[]”, $R=[]$.

Veamos ahora algunos ejemplos de predicados que trabajan con listas:

Implementaremos el predicado `num_elem/2` que cuenta el número de elementos de una lista.

Implementaremos el predicado en varios pasos:

Paso 1) Caso base.

Suele ser el más sencillo. Cuando tratamos listas, el caso base suele hacer referencia a la lista vacía. **El caso base no debe depender de otras reglas que hagan referencia al predicado definido.**

```
% num_elem(+List, -Int)
num_elem([],0).
```

Paso 2) Planteamos el caso recursivo.

Construimos el caso recursivo **con un tamaño del problema algo menor**. Usualmente con el resto. En lugar de una variable Lista, escribimos [Cabeza|Resto], que separará el primero de la lista del resto.

```
num_elem( [Cabeza|Resto],      ) :-
num_elem(Resto, Num_resto) ,
```

Si la llamada recursiva funciona, devolverá el número de elementos del resto.

Recuerde el principio de inducción:

Si es cierto para n_0 y es cierto para $n-1$ con $n > n_0$, entonces es cierto para cualquier $n > n_0$

Ejemplo en Prolog, `cierto(N) :- Nmenos1 is N-1, cierto(Nmenos1).`

Paso 3) A partir del caso recursivo (resultado parcial), obtenemos el resultado total. En este caso, basta con sumar 1 a NumResto, para obtener el total.

```
num_elem([Cabeza|Resto], NumLista ):-
num_elem(Resto, NumResto),
numLista is NumResto + 1.
```

Paso 4) Por último, comprobamos que todos los casos están contemplados una única vez. En este ejemplo, tenemos un caso para listas vacías y un caso genérico para una lista de 1 o más elementos. Hemos contemplado todos los tamaños de listas posibles.

Tendremos en cuenta que al escribir [Cabeza|Resto] obligamos a que la lista tenga, al menos, un elemento ya que el resto puede estar vacío pero la cabeza debe unificar forzosamente con un término.

El programa quedaría así:

```
% num_elem(+List, -Int)
num_elem([],0).
num_elem([Cabeza|Resto], NumLista ):-
num_elem(Resto, NumResto),
numLista is NumResto + 1.
```

Ejercicios:

Implementar los siguientes predicados utilizando recursividad:

`reverse(+List, -ListR)` que es cierto cuando ListR unifica con una lista que contiene los mismos elementos que List pero en orden inverso.

`aniadir_final(+Elem, +Lista, -ListaR)` que es cierto cuando ListaR unifica con una lista que contiene los mismos elementos que la lista Lista más el elemento Elem añadido al final.

`elemento_enesimo(+Pos, +Lista, -Elem)` que es cierto cuando Elem unifica con el elemento que ocupa la posición Pos dentro de Lista.

Predicados predefinidos

Existen predicados predefinidos en Prolog para listas que podemos utilizar en nuestros programas. Si hacemos la siguiente consulta `help(length)`. Obtendremos el siguiente resultado:

```
length(?List, ?Int)
```

```
True if Int represents the number of elements of
list List. Can be used to create a list holding
only variables.
```

El comportamiento es similar, a nuestro predicado `num_elem/2` implementado anteriormente.

```
Nota: En la sección 4.1 del manual de SWI-Prolog ( ?- help(4-1).
),
encontramos información sobre la notación utilizada en la des-
cripción de los predicados:
? indica que el argumento puede ser de entrada o de salida.
+ indica que el argumento es usualmente de entrada.
- indica que el argumento es usualmente de salida.
```

Otros predicados predefinidos para listas son:

```
append/3
member/2
reverse/2
nth0/3
nth1/3
```

Predicados reversibles

Una de las características que diferencia a Prolog de otros lenguajes de programación es la capacidad que tienen los predicados (a diferencia de una función definida en C, por ejemplo), de ser utilizados de forma reversible.

Podemos utilizarlos de forma similar a una función, es decir introduciendo unos valores en las entradas, esperando que las variables de salida, unifiquen con el resultado. Sin embargo, también podemos utilizarlos al revés, es decir, **introduciendo la salida esperando que Prolog encuentre la/s entrada/s** necesarias para hacer cierto el predicado.

Veamos un ejemplo:

```
6 ?- length([1,2,3], Longitud).  
Longitud=3  
  
7 ? length(Lista, 3).  
Lista = [_G331, _G334, _G337]
```

En el ejemplo 7 hemos utilizado el predicado `length/2` de forma contraria a la habitual. Indicamos cual es la salida, en este caso la longitud de la lista, y esperamos que Prolog encuentre una posible lista de longitud 3. La respuesta de Prolog en `Lista=[_G331,_G334,_G337]`. Es la forma que tiene Prolog de hacer referencia a una lista de tres elementos cualesquiera (`_G331,_G334,_G337` identifican variables).

Esta forma de uso de los predicados no se encuentra en los lenguajes imperativos. **Debemos hacer uso de la reversibilidad de los predicados**, cuando sea posible.

Ejercicios:

- 1) Comprobar el funcionamiento de los predicados predefinidos en Prolog para listas.
- 2) Comprobar si la implementación del predicado `num_elem/2` vista en este tema es reversible y razone por qué.
- 3) Revisar los exámenes de años anteriores y realizar los problemas de listas que se encuentren.

Bibliografía

Ayuda de SWI-Prolog

PROGRAMACIÓN LÓGICA CON PROLOG

0. Introducción

1. Unificación

2. Tipos de datos

2.1 Listas

2.2 Árboles

2.3 Grafos

3. Control de la ejecución

4. Problemas de estados

SECCIÓN 2. TIPOS DE DATOS (CONTINUACIÓN)

2.1 Listas (continuación)

Las listas son el tipo de dato más utilizado en Prolog ya que se sirven de ellas otros tipos de datos, como por ejemplo, árboles genéricos y grafos.

Es muy importante que comprendamos bien el funcionamiento de las listas y, sobre todo, cómo deben construirse los predicados recursivos. Si entendemos bien estos conceptos, es posible que lleguemos a ser unos buenos programadores declarativos. En otro caso, es muy posible que necesitemos demasiado tiempo para resolver los problemas y terminemos por pensar que la Programación Declarativa es demasiado compleja y que no merece la pena utilizarla.

El profesor David Enrique Losada Carril de la Universidad Santiago de Compostela en la página web de la asignatura Programación Declarativa da algunas claves para resolver problemas declarativamente: “piensa en especificar el problema y sus restricciones y no en secuenciar instrucciones;”

Recomendaciones importantes para resolver los ejercicios:

- No intentar seguir la ejecución paso a paso de los predicados para hallar la solución.
- Pensar en los predicados como algo estático.

Seguir los siguientes pasos:

- 1) **Plantear el caso recursivo con el tamaño de la llamada recursiva un poco más pequeño** (n-1, el resto de una lista, etc.);
- 2) Entonces plantear: **¿Si la llamada recursiva funciona, qué tengo en la salida “parcial”?** (Utilizar un ejemplo sencillo si es necesario para entender qué contienen las variables de salida de la llamada recursiva).
- 3) Una vez aclarado esto, plantear **¿Qué cambios se han de realizar a la salida de la llamada recursiva (“parcial”) para obtener la salida “total”?** (la del caso general n, de la lista completa, etc.).
- 4) **Por último comprobar que el caso base es correcto y que todos los posibles casos están contemplados.**

Si después de resolver el problema de esta forma, no existiera la seguridad de que funciona... ¡Probarlo en SWI-Prolog!

Veamos los siguientes ejemplos:

1. Invertir una lista
2. Concatenar dos listas
3. Ordenación por burbuja
4. Ordenación por inserción
5. Ordenación por Quicksort
6. Encontrar los primos entre X e Y
7. Permutar los elementos de una lista

8. Encontrar el elemento que aparece más veces en una lista

1) Invertir una lista

```
%-----
% invertir(+Lista, -ListaR) es cierto cuando ListaR
% unifica con una lista que contiene los mismos
% elementos que Lista en orden inverso.
%-----
invertir([], []).
invertir([Cab|Resto], RT) :-
    invertir(Resto, R),
    append(R, [Cab], RT).
```

2) Concatenar dos listas

```
%-----
% concatena(+List1, +List2, -ListR).
% es cierto cuando ListaR unifica con una lista
% que contiene los elementos de la lista List1
% en el mismo orden y seguidos de los elementos
% de la lista List2 en el mismo orden.
%-----
concatena([], L, L).
concatena([Cab|Resto], List2, [Cab|R]) :-
    concatena(Resto, List2, R).
```

3) Ordenación por burbuja

```
%-----
% ordena_burbuja(+Lista, -ListaR).
% es cierto cuando ListaR unifica con una lista que
% contiene los mismos elementos que Lista ordenados
% de menor a mayor.
%-----
ordena_burbuja(Lista, Lista) :- ordenada(Lista).
ordena_burbuja(Lista, RT) :-
    append(Ini, [E1, E2|Fin], Lista),
    E1 > E2,
    append(Ini, [E2, E1|Fin], R),
    ordena_burbuja(R, RT).
%-----
% ordenada(+Lista)
% es cierto cuando Lista unifica con una lista
% que contiene sus elementos ordenados de menor a
% mayor.
%-----
ordenada([]).
ordenada([_]).
ordenada([Cab1, Cab2|Resto]) :-
    Cab1 =< Cab2,
    ordenada([Cab2|Resto]).
```

4) Ordenación por inserción

```
%-----
% inserta_en_list_ord(+Elem, +Lista, -ListaR).
%es cierto cuando ListaR unifica con una lista
%que contiene los elementos de la lista ordenada
%Lista, con el elemento Elem insertado de forma
%ordenada.
%-----
inserta_en_list_ord(Elem, [], [Elem]).
inserta_en_list_ord(Elem, [Cab|Resto], [Elem,
Cab|Resto]):-
    Elem =< Cab.
inserta_en_list_ord(Elem, [Cab|Resto], [Cab|R]):-
    Elem > Cab,
    inserta_en_list_ord(Elem, Resto, R).
%-----
% ordena_insercion(+Lista, -ListaR).
%es cierto cuando ListaR unifica con una lista que
%contiene los mismos elementos que Lista ordenados
%de menor a mayor.
%-----
ordena_insercion([], []).
ordena_insercion([Cab|Resto], RT):-
    ordena_insercion(Resto, R),
    inserta_en_list_ord(Cab,R, RT).
```

5) Ordenación por Quicksort

```
%-----
% divide(+Elem, +Lista, -Menores, -Mayores)
%es cierto cuando Menores unifica con una lista que
%contiene los elementos de Lista que son menores
%o iguales que Elem y Mayores unifica con una lista
%que contiene los elementos de Lista que son
%mayores que Elem.
%-----
divide(_, [], [], []).
divide(Elem, [Cab|Resto], Menores, [Cab|Mayores]):-
    Cab > Elem,
    divide(Elem, Resto, Menores, Mayores).
divide(Elem, [Cab|Resto], [Cab|Menores], Mayores):-
    Cab =< Elem,
    divide(Elem, Resto, Menores, Mayores).
%-----
% ordena_quick(+Lista, -ListaR).
%es cierto cuando ListaR unifica con una lista que
```

```
%contiene los mismos elementos que Lista ordenados
%de menor a mayor.
%-----
ordena_quick([], []).
ordena_quick([Cab|Resto], R):-
    divide(Cab, Resto, Men, May),
    ordena_quick(Men, RMen),
    ordena_quick(May, RMay),
    append(RMen, [Cab|RMay], R).
```

6) Encontrar los primos entre X e Y

```
%-----
% lista_divisores(+X, +Y, -ListaR).
%es cierto cuando ListaR unifica con una lista
%que contiene a los números cuyo resto
%de la división entera de X entre Z es igual a 0
%para valores de Z entre 1 e Y.
lista_divisores(_, 1, [1]).
lista_divisores(X, Y, [Y|R]):-
    Y > 1,
    Y2 is Y-1,
    lista_divisores(X, Y2, R),
    0 is X mod Y.
lista_divisores(X, Y, R):-
    Y > 1,
    Y2 is Y-1,
    lista_divisores(X, Y2, R),
    Z is X mod Y, Z \== 0.
%-----
% primo(+X)
%es cierto si X unifica con un número primo.
%-----
primo(X):- lista_divisores(X, X, [X, 1]).
%-----
% primosEntrexy(+X, +Y, -ListaR)
%es cierto cuando ListaR unifica con una lista
%que contiene a los primos que van desde X hasta
%Y ambos incluidos en orden ascendente.
%-----
primosEntrexy(X, X, []).
primosEntrexy(X, Y, [X|R]):- X<Y,
    X2 is X+1,
    primosEntrexy(X2, Y, R),
    primo(X).
primosEntrexy(X, Y, R):- X<Y,
    X2 is X+1,
    primosEntrexy(X2, Y, R),
    \+ primo(X).
```

7) Permutar los elementos de una lista

```

%-----
% selecciona_uno(+Lista, -Elem, -Resto)
%es cierto cuando Elem unifica con cualquier
%elemento de la lista Lista y Resto unifica
%con una lista que contiene los elementos de
%Lista, en el mismo orden menos el elemento
%Elem.
%-----
selecciona_uno([Ca|R], Ca, R).
selecciona_uno([Ca|Co], E, [Ca|R]):-
    selecciona_uno(Co, E, R).
%-----
% permuta(Lista, ListaR).
%es cierto cuando ListaR unifica con una lista
%que contiene los elementos de Lista en orden
%distinto. Este predicado genera todas las
%listas posibles por backtracking.
%-----
permuta([], []).
permuta(L, [E|RP]):-
    selecciona_uno(L, E, R),
    permuta(R, RP).

```

8) Encontrar el elemento que aparece más veces en una lista

```

%-----
% mas_veces(+Lista, -Elem, -Num)
%es cierto cuando Elem unifica con el elemento
%que se repite más veces en la lista Lista
%y Num unifica con el número de veces que se
%repite dicho elemento.
%-----
mas_veces([],_,0).
mas_veces([Ca|Co], Ca, N2):-
    mas_veces(Co,El,N),
    Ca=El,
    N2 is N+1.
mas_veces([Ca|Co], El, N):-
    mas_veces(Co,El,N),
    Ca\=El.

```


PROGRAMACIÓN LÓGICA CON PROLOG

0. Introducción

1. Unificación

2. Tipos de datos

2.1 Listas

2.2 Árboles

2.3 Grafos

3. Control de la ejecución

4. Problemas de estados

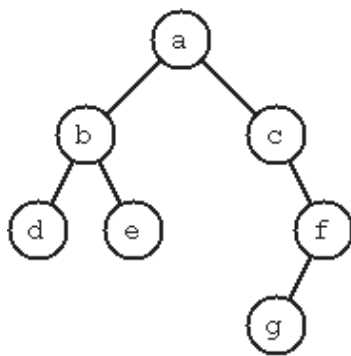
SECCIÓN 2.TIPOS DE DATOS (CONTINUACIÓN)

2.2 Árboles

Árboles binarios

Un árbol binario es

- Árbol nulo, o bien,
- Una estructura compuesta por un elemento y dos sucesores que son árboles binarios.



En Prolog representaremos un árbol nulo con el átomo 'nil' y el árbol no vacío con el término a(Et, HI, HD), donde "Et" representa la etiqueta de la raíz y HI y HD son los subárboles Izquierdo y derecho respectivamente.

A1=a(a,a(b,a(d,nil,nil),a(e,nil,nil)),a(c,nil,a(f,a(g,nil,nil),nil)))

Otros ejemplos son un árbol que sólo contiene un nodo

A2 = a(a,nil,nil) o el árbol vacío A3 = nil

Veamos un ejemplo con el predicado cuenta_nodos para árboles binarios.

```

/* cuenta_nodos(+Arbol_binario, ?Num_nodos)

es cierto cuando Num_nodos unifica con el
numero de nodos del árbol "Arbol_binario" */

cuenta_nodos(nil, 0).

cuenta_nodos(a(_, HI, HD), R):-
    cuenta_nodos(HI, RI),
    cuenta_nodos(HD, RD),
    R is RI + RD + 1.

dato(a(a,a(b,a(d,nil,nil),a(e,nil,nil)),a(c,nil,a(f,a(
g,nil,nil),nil))))).

/* 1 ?- dato(A), cuenta_nodos(A, N).
A = a(a, a(b, a(d, nil, nil), a(e, nil, nil)), a(c,
nil, a(f, a(g, nil, nil), nil)))

```

Otro ejemplo es el predicado `lista_hojas` para árboles binarios

```

/* lista_hojas(+Arbol_binario, ?Lista_hojas)

   es cierto cuando Lista_hojas unifica con una lista que con-
   tiene las etiquetas de las hojas del árbol "Arbol_binario"
   */

lista_hojas(nil, []).

lista_hojas(a(_, nil, HD), LD):-
HD \= nil,
lista_hojas(HD, LD).  lista_hojas(a(_, HI, nil), LI):-
HI \= nil,
lista_hojas(HI, LI).
  lista_hojas(a(_, HI, HD), LR):-
HI \= nil, HD \= nil,
lista_hojas(HI, LI),
lista_hojas(HD, LD),
append(LI, LD, LR).  lista_hojas(a(Et, nil, nil), [Et]).

dato(a(a(a(b,a(d,nil,nil)),a(e,nil,nil)),a(c,nil,a(f,a(g,ni
l,nil),nil))))).

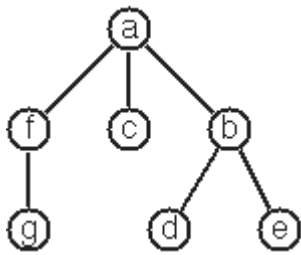
/* 1 ?- dato(A), lista_hojas(A, R).

A = a(a, a(b, a(d, nil, nil), a(e, nil, nil)), a(c, nil,
a(f, a(g, nil, nil), nil)))
R = [d, e, g] */

```

Árboles genéricos

Un árbol genérico está compuesto por una raíz y por una lista de sucesores que son a su vez árboles genéricos. Aunque podríamos establecer una representación en nuestra aproximación, para simplificar consideraremos que un árbol genérico no estará nunca vacío, es decir, no tendremos un equivalente al árbol nulo de los árboles binarios.



En Prolog representaremos un árbol genérico por el término `a(Et, ListaHijos)`, donde `Et` es la etiqueta de la raíz y `ListaHijos` es una lista con los árboles descendientes. El árbol del ejemplo lo representaremos con el siguiente término Prolog.

```
A = a(a,[a(f,[a(g,[])]),a(c,[]),a(b,[a(d,[]),a(e,[])])])
```

Metodología para la implementación de árboles genéricos

Con la idea de simplificar la implementación de los predicados de árboles genéricos, escribiremos unas cláusulas que unifiquen con una estructura de árbol y otra/s cláusula/s que unificarán con listas de árboles.

En el caso de las cláusulas para listas de árboles, seguiremos el mismo principio utilizado en las listas.

- Un caso base, que será por lo general, lista vacía.
- Un caso recursivo, que separa el primero del resto, realiza la llamada recursiva sobre el resto de la lista (en este caso de árboles) y, a partir del resultado, construye la solución.

Implementaremos el predicado `cuenta_nodos` para árboles genéricos como ejemplo:

```

/* cuenta_nodos(+Arbol_generico, ?Num_nodos)

es cierto cuando Num_nodos unifica con el
numero de nodos del árbol "Arbol_generico" */

%cláusula para un árbol genérico
cuenta_nodos(a(_, Lista_hijos), R):-
cuenta_nodos(Lista_hijos, N),
R is N + 1.

% cláusulas para lista de árboles
cuenta_nodos([], 0).
cuenta_nodos([Cab|Resto], _):-
cuenta_nodos(Cab, NCab),
cuenta_nodos(Resto, NResto),
R is Ncab + Nresto.

dato(a(a,[a(f,[a(g,[[]])]),a(c,[[]]),a(b,[a(d,[[]]),a(e,[[]])])])).

/* dato(A), cuenta_nodos(A,N).
A = a(a, [a(f, [a(g, [])]), a(c, []), a(b, [a(d, []), a(e, [])])])
N = 7 */

```

Otro ejemplo, lo encontramos en el predicado profundidad para árboles genéricos:

```

/*
profundidad_ag(+Arbol_generico, ?P)
    es cierto cuando P unifica con la profundidad del
    árbol genérico "Arbol_genérico"

*/

profundidad_ag(a(_, Lista_hijos), R):-
    profundidad_ag(Lista_hijos, PH),
    R is PH+1.

profundidad_ag([], 0).

profundidad_ag([Cab|Resto], PCab):-
    profundidad_ag(Cab, PCab),
    profundidad_ag(Resto, PResto),
    PCab >= PResto.

profundidad_ag([Cab|Resto], PResto):-
    profundidad_ag(Cab, PCab),
    profundidad_ag(Resto, PResto),
    PCab < PResto.

dato(a(a, [a(f, [a(g, [])]), a(c, [])], a(b, [a(d, []), a(e, [])]))).

/*
dato(A), profundidad_ag(A,N).

A = a(a, [a(f, [a(g, [])]), a(c, [])], a(b, [a(d, []), a(e, [])]))
N = 3

```

Ejercicio:

Plantear cómo se podría extender la representación de árboles genéricos propuesta para representar árboles genéricos nulos.

PROGRAMACIÓN LÓGICA CON PROLOG

0. Introducción

1. Unificación

2. Tipos de datos

2.1 Listas

2.2 Árboles

2.3 Grafos

3. Control de la ejecución

4. Problemas de estados

SECCIÓN 2. TIPOS DE DATOS (CONTINUACIÓN)

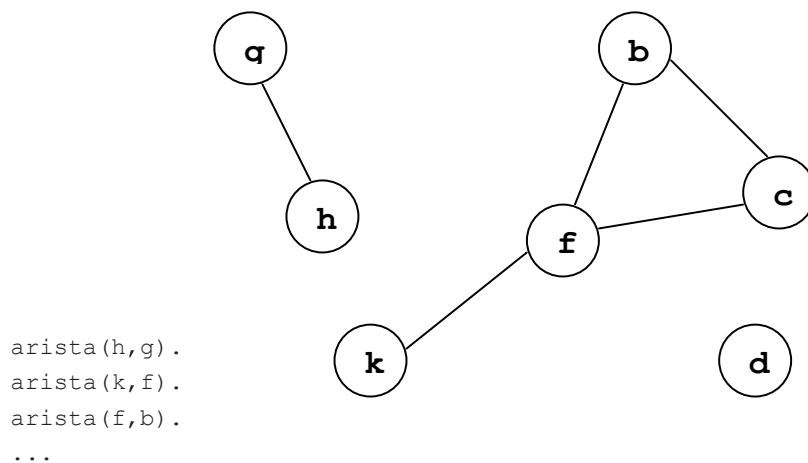
2.3 Grafos

Referencia: Ninety-Nine Prolog Problems:

<https://prof.ti.bfh.ch/hew1/informatik3/prolog/p-99>

Un **grafo** se define como un conjunto de nodos y un conjunto de aristas, donde cada arista une a dos nodos.

Existen diferentes modos de representar un grafo en Prolog. Uno de estos métodos consiste en representar cada arco con una cláusula (hecho) independiente. Para representar el siguiente grafo con este método escribiremos:



Podemos denominar a esta representación “*Representación Cláusula-arista*”. Con este método no podemos representar los nodos aislados (en el ejemplo anterior el nodo “d”). Esta representación puede hacer la implementación del camino algo más sencilla, sin embargo complica bastante tareas como recorrer todas las aristas o visitar todos los nodos.

Otro método consiste en representar todo el grafo en un único objeto. Siguiendo la definición dada anteriormente de un grafo como un par de dos conjuntos (aristas y nodos), podemos utilizar el siguiente término en Prolog para representar nuestro grafo ejemplo:

```

grafo([b,c,d,f,g,h,k],[arista(b,c), arista(b,f), arista(c,f), arista(f,k),
arista(g,h)])

```

o de una forma más compacta

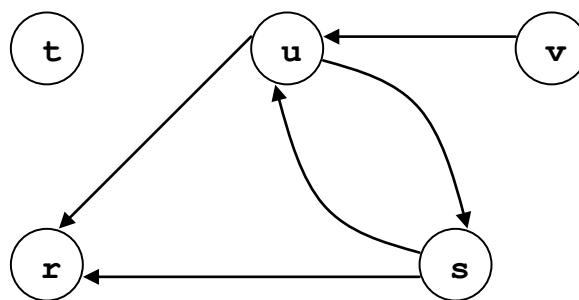
```

grafo([b,c,d,f,g,h,k],[a(b,c), a(b,f), a(c,f), a(f,k), a(g,h)])

```

Denominaremos a esta representación “*Representación Término-Grafo*”. Tendremos en cuenta que el conjunto se representa con una lista sin elementos repetidos. Cada arista aparece sólo una vez en la lista de aristas; por ejemplo, una arista desde el nodo x al nodo y se representa como $a(x,y)$ y el término $a(y,x)$ no aparece en la lista. La *Representación Término-Grafo* es la representación que utilizaremos por defecto.

En SWI Prolog existen predicados predefinidos para trabajar con conjuntos. Ejecute `help(11-1)` en SWI Prolog para obtener información sobre la sección “Set Manipulation”.



Pueden consultarse otras representaciones en “Ninety-Nine Prolog Problems”

<https://prof.ti.bfh.ch/hew1/informatik3/prolog/p-99/>

Las aristas pueden ser dirigidas, cuando sólo enlazan a los nodos implicados en sentido origen -> destino y no en el contrario; o no dirigidas cuando enlazan a los nodos en ambos sentidos.

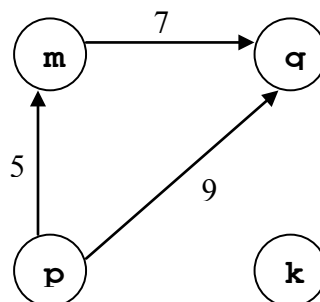
Cuando las aristas están dirigidas llamaremos al grafo Grafo dirigido.

Dependiendo del problema a resolver necesitaremos usar un tipo de grafo u otro.

Los nodos y las aristas de los grafos dirigidos y no dirigidos pueden incorporar información adicional. Es posible sustituir el nombre del nodo por una estructura que contenga, por ejemplo el nombre de la ciudad y el código postal ciudad(‘Huelva’,27002). Por otro lado, es posible también añadir información a las aristas, como por ejemplo el coste de recorrerlas.

Denominaremos a los grafos que añaden información adicional a vértices o aristas “Grafos etiquetados”.

Ejemplo:



Representación Cláusula-Arista

```
arista(m,q,7).
```

```
arista(p,q,9).
```

```
arista(p,m,5).
```

Representación Término-Grafo

```
grafo([k,m,p,q],[arista(m,p,7),arista(p,m,5),arista(p,q,9)])
```

Construcción de un camino en grafos

Gran cantidad de los problemas de grafos se resuelven construyendo un camino que una dos nodos del grafo.

Los predicados que encuentran un camino en un grafo son recursivos.

El predicado camino tendrá la siguiente cabecera

```
camino(+Inicio, +Fin, +Grafo, +Visitados, ?Camino,<?Peso_total>, <?Camino2>)
```

Inicio: Representa al nodo inicial del camino

Fin: Representa al nodo final del camino

Grafo: Grafo en el que buscamos el camino

Visitados: Lista de nodos o de aristas (dependerá del caso), que se utiliza para evitar que el predicado se quede iterando en un ciclo.

Camino: Lista de nodos o de aristas (dependiendo del caso) que representa el camino.

Los campos anteriores son necesarios en la mayoría de los problemas de grafos.

<Peso_total> : En ocasiones es necesario indicar cual es el peso total del camino.

<Camino2> : En ocasiones es necesario indicar tanto la lista de nodos visitados como la lista de aristas visitados. Ejemplo carreteras visitadas, ciudades visitadas y kilómetros recorridos.

Caso base:

Primera posibilidad y más habitual.

```
camino(Fin, Fin, _, []).
```

Hemos indicado antes que la lista de visitados tiene como única función evitar que el predicado caiga en un bucle infinito. En el caso base, al no tener llamada recursiva, la lista de Visitados es irrelevante. Podemos hacer el siguiente razonamiento: **Si ya estoy en el final del camino he terminado, independientemente de los puntos por los que haya pasado.**

La variable de salida **Camino**, en este caso será una lista vacía o bien una lista con el nodo Fin si lo que deseo devolver es la lista de nodos que he visitado.

Caso base alternativo:

```
camino(Inicio, Fin, _, [arista(Inicio, Fin)]):-
    arista(Inicio, Fin).
```

Lo utilizaremos sólo cuando sea imprescindible. Por lo general el caso base anterior funciona bien y es más sencillo. Un tipo de problemas en el que utilizaremos este caso base alternativo es el de la obtención de los ciclos de un grafo.

Caso recursivo:

Para construir el caso recursivo buscaremos avanzar a algún vértice TMP y a partir de ese punto buscaremos un camino hasta el final.

```
Inicio ----> TMP -----> Fin

camino(Inicio, Fin, grafo(Vertices, Aristas),
Visitados, [arista(Inicio,TMP)|Camino]):-
    conectado(Inicio, TMP, Aristas),
    \+ visitado(Inicio, Fin, Visitados),
    camino(TMP, Fin, [arista(Inicio, TMP)|Visitados],
Camino).
```

Dependiendo del tipo de grafo y del objetivo perseguido, los predicados conectado y visitado podemos implementarlos de forma distinta.

Predicado conectado en grafos no dirigidos:

```
% conectado(+Inicio, +Fin, +Aristas)
conectado(Inicio, Fin, Aristas):- member(arista(Inicio, Fin), Aristas).

conectado(Inicio, Fin, Aristas):- member(arista(Fin, Inicio), Aristas).
```

En este predicado el corte “!” (que veremos con detalle en la sección 3) tiene como finalidad evitar que Prolog intente buscar una solución por un camino que sabemos no devolverá ninguna solución válida.

Predicado conectado en grafos dirigidos:

```
% conectado(+Inicio, +Fin, +Aristas)
conectado(Inicio, Fin, Aristas):- member(arista(Inicio, Fin), Aristas).
```

Predicado visitado en grafos no dirigidos:

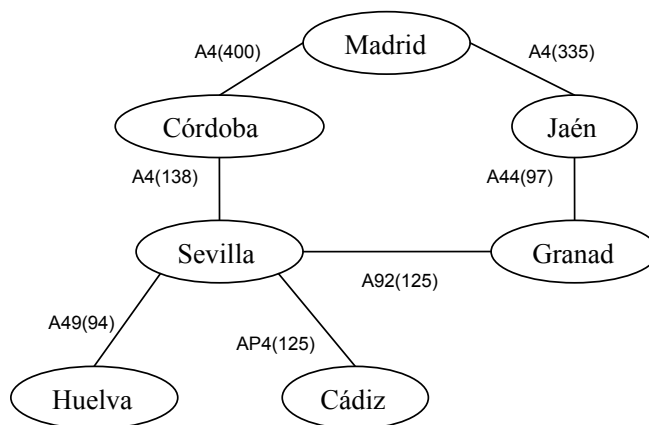
```
% visitado(+Inicio, +Fin, +Visitados)
visitado(Inicio, Fin, Aristas):- member(arista(Inicio, Fin), Visitados).

visitado(Inicio, Fin, Aristas):- member(arista(Fin, Inicio), Visitados).
```

Predicado visitado en grafos dirigidos:

```
% visitado(+Inicio, +Fin, +Visitados)
visitado(Inicio, Fin, Aristas):- member(arista(Inicio, Fin), Visitados).
```

Ejemplos:



En este caso, se trata de un grafo no dirigido.

```
dato(grafo([madrid, cordoba, sevilla, jaen, granada,
huelva, cadiz],
[arista(huelva, sevilla, a49, 94),
arista(sevilla, cadiz, ap4, 125),
arista(sevilla, granada, a92, 256),
arista(granada, jaen, a44, 97),
arista(sevilla, cordoba, a4, 138),
arista(jaen, madrid, a4, 335),
arista(cordoba, madrid, a4, 400)]
)).
```

```
% conectado(+Inicio, +Fin, +Aristas, -Carretera, -Kilometros)
conectado(Inicio, Fin, Aristas, C, K):-
member(arista(Inicio, Fin, C, K), Aristas).
```

```

conectado(Inicio, Fin, Aristas, C, K):-member(arista(Fin, Inicio,C,K), Aris-
tas).

% visitado(+Inicio, +Fin, +Visitados)
visitado(Inicio, Fin, Visitados):-member(arista(Inicio, Fin,_,_), Visita-
dos).
visitado(Inicio, Fin, Visitados):- member(arista(Fin,Inicio,_,_), Visita-
dos).

%camino(Inicio, Fin, Grafo, Visitados, Ciudades,Carreteras, Kilometros)
camino(Fin, Fin, _, _, [Fin], [], 0).

camino(Inicio, Fin, G, Visitados, [Inicio|Ciudades],[Carretera|Carreteras],
K2):-
    G = grafo(_, Aristas),
    conectado(Inicio, TMP, Aristas, Carretera, K),
    \+ visitado(Inicio, TMP, Visitados),
    camino(TMP, Fin, G, [arista(Inicio,TMP,_,_)|Visitados], Ciudades, Carrete-
ras, Kilometros),
    K2 is Kilometros + K.

% dato(G), camino(huelva, madrid, G, [],C,Ca,K).

```

Ejecutaremos el objetivo camino para encontrar las soluciones. En este caso queremos encontrar las alternativas para llegar desde Huelva a Córdoba. La primera opción pasa por Granada y Madrid recorriendo 1182 kilómetros y la segunda tiene una longitud de 232 kilómetros.

```

1 ?- dato(G), camino(huelva, cordoba, G, [], Ca,C,K).
G = grafo([madrid, cordoba, sevilla, jaen, granada,
huelva, cadiz], [arista(huelva, sevilla, a49, 94),
arista(sevilla, cadiz, ap4, 125), arista(sevilla,
granada, a92, 256), arista(granada, jaen, a44, 97),
arista(sevilla, cordoba, a4, 138), arista(jaen, madrid,
a4, 335), arista(cordoba, madrid, a4, 400)]),
Ca = [huelva, sevilla, granada, jaen, madrid, cordoba],
C = [a49, a92, a44, a4, a4],
K = 1182 ;
G = grafo([madrid, cordoba, sevilla, jaen, granada,
huelva, cadiz], [arista(huelva, sevilla, a49, 94),
arista(sevilla, cadiz, ap4, 125), arista(sevilla,
granada, a92, 256), arista(granada, jaen, a44, 97),
arista(sevilla, cordoba, a4, 138), arista(jaen, madrid,
a4, 335), arista(cordoba, madrid, a4, 400)]),
Ca = [huelva, sevilla, cordoba],
C = [a49, a4],
K = 232 ;
false.

```


PROGRAMACIÓN LÓGICA CON PROLOG

0. Introducción

1. Unificación

2. Tipos de datos

2.1 Listas

2.2 Árboles

2.3 Grafos

3. Control de la ejecución

4. Problemas de estados

SECCIÓN 2. CONTROL DE LA EJECUCIÓN

El *corte*

Notaremos el corte con un cierre de admiración “!”. Dada la siguiente cláusula que contiene el corte:

H:- B1, B2, ..., Bm, !, Bm+1, ..., Bn.

Consideremos que fue invocada por un objetivo **G** que unifica con **H**.

Primer efecto del corte:

En el momento en el que se alcanza el corte, el sistema ya ha encontrado soluciones para los términos B1,B2,...,Bm. **Las soluciones encontradas para B1,B2,..Bm se mantienen pero cualquier otra alternativa para estos objetivos es descartada.** Para el resto de los objetivos Bm+1,...,Bn, sí se aplicarán las técnicas de *backtracking* habituales.

Segundo efecto del corte:

Cualquier intento de satisfacer el objetivo G con otra cláusula es descartada.

Existen dos modos de uso del corte. Lo que algunos autores llaman cortes verdes y cortes rojos.

Los *cortes verdes*, son aquellos que **no modifican las soluciones del programa**. Es decir, que con corte y sin corte el programa produce las mismas soluciones. Se utiliza para mejorar la eficiencia de ejecución.

Los *cortes rojos* son aquellos que **modifican las soluciones del programa** (al quitar el corte las soluciones son distintas). Este tipo de cortes hacen los programas menos declarativos y deben utilizarse con reservas. En programación declarativa no importa el orden en que se escriban las cláusulas, las soluciones deben ser las mismas. Sin embargo el uso de cortes rojos impide que las cláusulas se puedan cambiar de orden.

Ejemplo:

```
1. cuenta_hojas(nil,0).
2. cuenta_hojas(a(_,nil,nil), 1):- !.
3. cuenta_hojas(a(_,HI,HD), R):-
    cuenta_hojas(HI, RI),
    cuenta_hojas(HD, RD),
    R is RI+RD.
```

En este ejemplo, si eliminamos el corte, para la consulta `cuenta_hojas(a(1,nil,nil), R)`, el predicado daría dos soluciones una correcta ($R=1$) y otra incorrecta ($R=0$).

Otra versión del mismo problema con cortes “verdes”:

```

1. cuenta_hojas(nil,0).
2. cuenta_hojas(a(_,nil,nil), 1):- !.

3. cuenta_hojas(a(_,HI,HD), _):-
    HI \= nil, HD \= nil, !,
    cuenta_hojas(HI, RI).
    cuenta_hojas(HD, RD),
    R is RI + RD.

4. cuenta_hojas(a(_, nil, HD), RD):-
    HD \= nil, !,
    cuenta_hojas(HD, RD).

5. cuenta_hojas(a(_, HI, nil), RI):-
    HD \= nil,
    cuenta_hojas(HI, RI).

```

En este otro ejemplo, al eliminar los cortes el programa obtiene las mismas soluciones. En este caso, el corte evita que se realicen algunas comprobaciones. Por ejemplo, si llegamos al corte de la cláusula 2, no tiene sentido buscar una solución en las cláusulas que aparecen a continuación.

Árbol de resolución

También conocido como árbol de deducción, representa el modo en el que Prolog resuelve una determinada consulta.

Algoritmo de resolución.

Repetir mientras queden términos:

- 1) **Buscamos** en la base de conocimiento de arriba abajo **unificaciones del término objetivo en las cabezas de las reglas y en los hechos**.

a) Abrimos **tantas ramas de ejecución como unificaciones** y empezamos a resolver por la que está más a la izquierda.

b) Si no **hubiese ninguna unificación, se produce un fallo y se explora la siguiente rama inmediatamente a la derecha** en el mismo nivel. Si no existiesen ramas al mismo nivel, pasaríamos al nivel superior empezando por la rama que esté más a la izquierda.

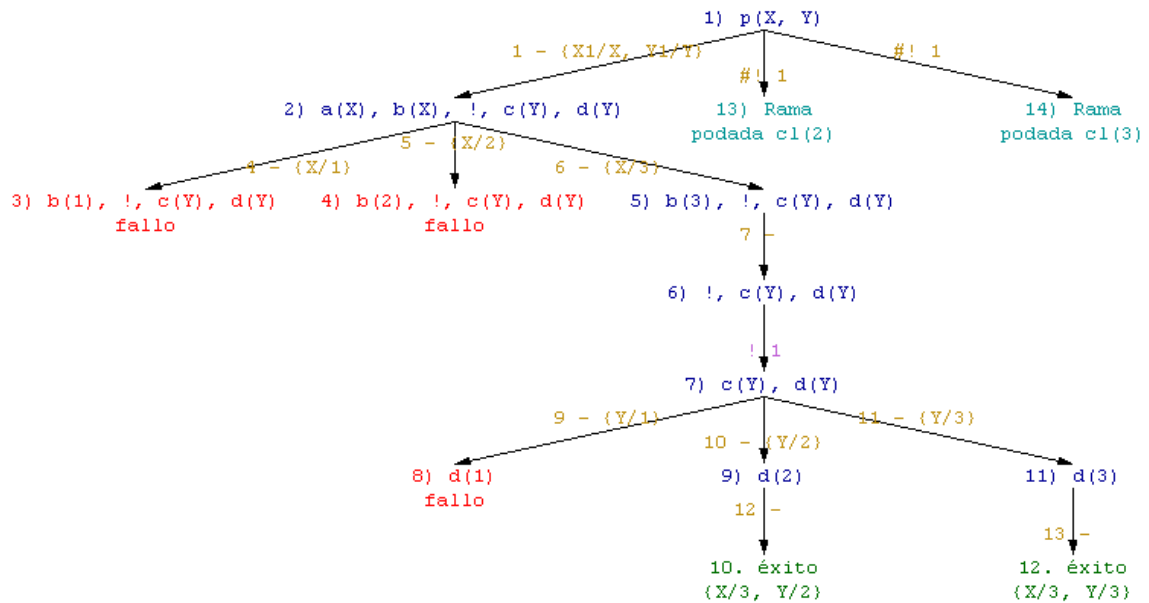
- 2) En la rama actual, **sustituimos la cabeza por el cuerpo de la regla** **instanciando las variables**. Fijamos como término objetivo actual el que está más a la izquierda y volvemos al paso 1.

Ejemplo:

```

1. p(X,Y):- a(X), b(X), !, c(Y), d(Y).
2. p(1,Y):- c(Y).
3. p(X,2):- c(X).
4. a(1).
5. a(2).
6. a(3).
7. b(3).
8. b(4).
9. c(1).
10. c(2).
11. c(3).
12. d(2).
13. d(3).

```



Pasos para la creación de los árboles de resolución

- 1) **Numeraremos las cláusulas** del programa de arriba abajo en orden ascendente.
- 2) Escribiremos el objetivo propuesto y **buscaremos unificaciones del objetivo en las cabezas de las reglas y en los hechos.**
- 3) **Dibujaremos tantas ramas como unificaciones haya y etiquetaremos cada rama con el número de la cláusula.** Si no existiese ninguna unificación anotaremos un fallo para el objetivo.
- 4) **Empezaremos resolviendo por la rama que está más a la izquierda.**
- 5) Anotaremos la sustitución de variables en la rama y **sustituiremos la cabeza por el cuerpo de la regla**, sin olvidar los términos anteriores si los hubiese.
- 6) **Resolveremos los términos de izquierda a derecha.** Recuerde que para que el consecuente sea cierto deben ser ciertos todos los términos del antecedente. Si uno de los términos fuese false, no seguiremos comprobando el resto de términos.
- 7) **Cuando un término se hace cierto, lo eliminamos y seguimos resolviendo el resto de izquierda a derecha.**
- 8) **Cuando hagamos todos los términos, marcaremos esa rama con “éxito” y anotaremos los valores de las soluciones** para las variables del objetivo.

Tratamiento del corte en los árboles de resolución

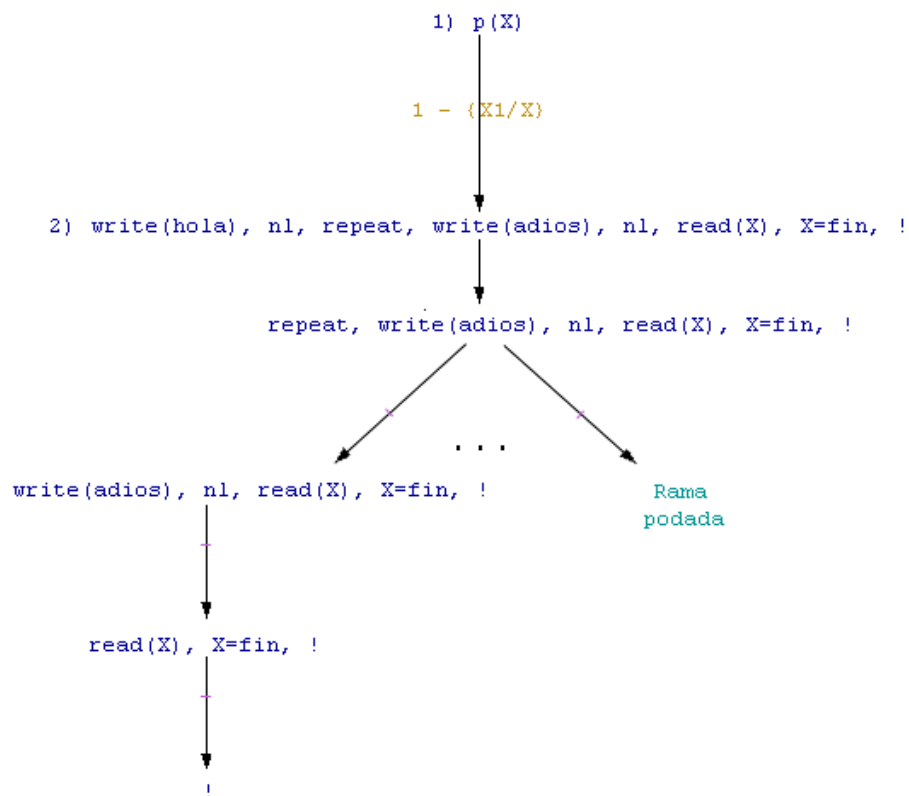
Cuando alcancemos un corte en un árbol de resolución, seguiremos los siguientes pasos:

- 1) **Identificaremos la cláusula que contiene al corte** y los elementos H, G, los términos que están antes del corte **B1, B2,..., Bm** y los que están después **Bm+1, ..., Bn**. Tal y como indica la definición del corte **H:- B1, B2,...,Bm,!, Bm+1, ..., Bn**.
- 2) **Comprobaremos si existen términos antes del corte.** Si existen tendremos que aplicar el primer efecto del corte, marcando con ‘**Rama podada**’ las ramas correspondientes a las alternativas para los términos que están antes del corte.
- 3) **Identificaremos al objetivo que intentábamos satisfacer (G) que unificó con la cabeza (H) de la regla que contiene al corte y, aplicando el segundo efecto, marcaremos como ‘Rama podada’ las ramas correspondientes al intento de satisfacer el objetivo G con otras cláusulas.**

Término “repeat”

El término repeat, tiene un efecto similar a un término que tuviese infinitas unificaciones. Veamos un ejemplo:

```
1. p(X):- write('hola'), repeat, write('adios'), read(X), X='fin', !.
```



Este ejemplo muestra una vez `hola`, escribe `adios` y lee un término por teclado. Si el término es igual a `fin`, se alcanza el corte y se eliminan todas las ramas del `repeat` y el programa termina. Si no se introduce `fin`, el programa no termina.

Elemento ‘fail’

Cuando se alcanza el término fail en una cláusula, ésta se hace falsa.

Ejemplo para probar:

```
1. q([C|R],0):-s(C),q(R,1).
2. q([C,next(a)],1):-!,r([C]).
3. q([C,[N]],1):-r([C]),!,N is 2+3.
4. r(1).
5. r([]).
6. r([next(a)]).
7. s(R):-!,t(R).
8. s(1):-q([],1).
9. t(1):-fail.
```

SLD-Draw

Puede utilizarse la herramienta SLD-Draw (<http://www.lcc.uma.es/~pacog/sldDraw/>) para dibujar los árboles de resolución que no contengan ‘repeat’. Si existen dudas de si un árbol de resolución es correcto, la mejor opción es implementarlo, introducirlo en Prolog y solicitar todas las soluciones utilizando “;”. Prolog nos dará en todos los casos todas las soluciones válidas. SLD-Draw en algunas ocasiones no produce todas las soluciones válidas.

PROGRAMACIÓN LÓGICA CON PROLOG

0. Introducción

1. Unificación

2. Tipos de datos

2.1 Listas

2.2 Árboles

2.3 Grafos

3. Control de la ejecución

4. Problemas de estados

SECCIÓN 3. PROBLEMAS DE ESTADOS

Un método para resolver problemas haciendo uso de la programación lógica consiste en trazar un camino que transcurre por diferentes situaciones (estados) hasta llegar a una solución.

Una transición de un estado a otro se produce realizando un movimiento. De esta forma, la clave para resolver un problema con esta metodología consistirá en definir un estado y unos movimientos; y en base a estos dos elementos se construirá un camino que parte de un estado inicial y tras una secuencia de movimientos alcanza un estado final.

Construiremos la solución en cuatro pasos:

- 1) Definición de un estado genérico y de los estados inicial y final.**
- 2) Implementación de los movimientos.**
- 3) Implementación del camino.**
- 4) Implementación del predicado solución.**

Explicaremos la resolución de problemas de estados utilizando el ejemplo de los caníbales y los misioneros.

Caníbales y misioneros

Enunciado del problema:

- Hay tres misioneros y tres caníbales en la orilla izquierda de un río
- Tanto los misioneros como los caníbales, desean cruzar a la orilla derecha de río.
- Para cruzar, disponen de una barca que puede transportar sólo a dos personas a la vez.
- El número de caníbales en cada orilla nunca debe ser mayor al número de misioneros, en otro caso los misioneros se convertirían en la cena de los caníbales.
- Planee una secuencia de movimientos para cruzar a los misioneros y los caníbales de forma que al final estén todos sanos y salvos en la orilla de la derecha.

Representaremos el problema como un conjunto de:

- Estados: que representan una instantánea del problema.
- Movimientos: que transforman un estado en otro.

Encontramos una relación entre los problemas de estados y los grafos asociando nodos con estados y aristas con movimientos. Si recordamos la forma de construir un camino en los grafos, la secuencia de movimientos de un problema de estados tendrán básicamente la misma estructura.

1) Definición de un estado genérico y de los estados inicial y final.

- Un estado representa un instante del problema.
- Utilizando la información del término **estado(...)** debemos ser capaces de representar el problema sin perder información relevante evitando duplicar información.
- El estado no incluir en el estado información de las acciones.

Volviendo ahora al problema de los caníbales y los misioneros, la información que necesitamos para representar el problema es:

- El número de misioneros en la orilla de la izquierda
- El número de caníbales en la orilla de la izquierda
- El lugar en el que se encuentra la barca.

Toda la información necesaria para resolver el problema, podemos obtenerla a partir de estos tres datos. En Prolog, podemos representar este estado con un término de 3 argumentos.

```
estado(Canibales_izq, Misioneros_izq, Pos_barca)
```

Utilizando esta representación, definiremos los estados inicial y final:

```
inicial(estado(3,3,izq)).
final(estado(0,0,dch)).
```

2) Implementación de los movimientos.

El siguiente paso consiste en definir los movimientos posibles. En el caso del problema de los caníbales y los misioneros tendríamos los siguientes movimientos:

- Pasar un misionero a la izquierda o a la derecha.
- Pasar un caníbal a la izquierda o a la derecha.
- Pasar dos caníbales a la izquierda o a la derecha.
- Pasar dos misioneros a la izquierda o a la derecha.
- Pasar un misionero y un caníbal a la izquierda o a la derecha.

El predicado mover, escrito ‘mov’, tendrá por lo general, la siguiente cabecera:

```
/*mov(?Movimiento,?Estado_anterior,?Estado_posterior)es cierto cuando Movimiento unifica con un Movimiento válido, Estado_anterior unifica con un estado válido y Estado_posterior unifica con el estado resultante de aplicar el movimiento "Movimiento" al estado "Estado_anterior" */
```

Es posible que para algún tipo de problema sea necesario incluir algún argumento más. Por ejemplo, si quisiéramos calcular cuanto tiempo es necesario para llegar a la solución, podemos asociar un coste en tiempo a cada movimiento.

Para poder implementar los movimientos, es necesario pensar en las condiciones necesarias antes de realizar el movimiento, es decir como debe ser el estado anterior al movimiento. Por ejemplo, para poder pasar un misionero a la izquierda es necesario que exista al menos un misionero a la izquierda y que la barca esté a la derecha.

```
mov(pasar_un_mis_izq, estado(MI, CI, dch), estado(MI2, CI)):-
    MI < 3, MI2 is MI + 1.
```

No estamos comprobando si el estado nuevo es un estado válido. Cuando tenemos muchos movimientos (en este caso hay 10 movimientos) interesa no incluir en los movimientos las comprobaciones de si el estado nuevo es válido ya que repetiríamos en cada movimiento la misma comprobación. Es mejor incluir un predicado `valido(Estado)` que será cierto, cuando Estado sea un estado válido.

Otra forma de escribir los movimientos es hacerlos más genéricos para evitar escribir varios movimientos muy parecidos. Sin embargo, esta simplificación no siempre es fácil de encontrar. A continuación se propone una simplificación para los movimientos que los reduce sólo a dos cláusulas más un conjunto de 10 hechos. Utilizaremos un predicado `pasar` que representaremos mediante 10 hechos.

`pasar(?Num_misioneros, ?Num_canibales, ?Lugar)` será cierto cuando `Num_misioneros` y `Num_canibales` unifica con una combinación de misioneros y caníbales válida según la especificación del problema y cuando `lugar` unifica con 'izq' o 'dch'.

```
pasar(1,0,izq).
pasar(1,0,dch).
pasar(0,1,izq).
pasar(0,1,dch).
pasar(2,0,izq).
pasar(2,0,dch).
pasar(0,2,izq).
pasar(0,2,dch).
pasar(1,1,izq).
pasar(1,1,dch).
```

Los movimientos quedarían de la siguiente forma

```
mov(pasar(M, C, izq), estado(MI, CI, dch), estado(MD, CD, izq)):-
    pasar(M,C,izq),
    NT is M + C, NT <= 2, NT >= 1,
    M <= MI, C <= CI,
    MD is MI + M, CD is CI + C.

mov(pasar(M, C, dch), estado(MI, CI, izq), estado(MD, CD, dch)):-
    pasar(M,C,dch),
    NT is M + C, NT <= 2, NT >= 1,
    M <= MI, C <= CI,
    MD is MI -M, CD is CI - C.
```

3) Implementación del camino.

La solución a este tipo de problemas viene dada por una secuencia de movimientos que, partiendo de un estado inicial, hace que el problema evolucione hasta el estado final deseado.

La implementación de este camino es básicamente la misma que utilizamos en el recorrido de los grafos, utilizando estados en lugar de nodos y movimientos en lugar de aristas.

El predicado camino tendrá como mínimo los siguientes argumentos:

```
/* camino(+Estado_inicial, +Estado_final, +Visitados, -Camino)
es cierto cuando Estado_inicial y Estado_final unifican con estados válido, Vi-
sitados unifica con una lista de estados visitados. */
```

```
camino(Inicio, Inicio, _, []).
```

```
camino(Inicio, Fin, Visitados, [Mov|Camino]):-
    length(Visitados, L), L < 10,
    mov(Mov, Inicio, Int),
    \+ member(Int, Visitados),
    camino(Int, Fin, [Int|Visitados], Camino).
```

Dependiendo del problema, el predicado camino puede tener algún argumento más si queremos, por ejemplo, guardar el coste total del camino.

Este camino tiene la particularidad de limitar como mucho a 11 (10 visitados + 1 paso final) la longitud del camino. Esta limitación tiene sentido cuando las posibilidades de exploración sean muchas y sepamos que en un número de pasos como máximo está la solución.

Podemos asimismo incluir la comprobación de si el nuevo estado es correcto después de realizar el movimiento. Esta comprobación la haremos con un predicado de validación que llamaremos valido(Estado). En este caso la comprobación se realiza dentro del predicado mov.

Por último, indicar que la lista de visitados contendrá estados, ya que es posible repetir el mismo movimiento siempre que no se pase dos veces por un mismo estado (para evitar caer en un bucle infinito).

4) Implementación del predicado solución

Este predicado hace uso de camino y los estados inicial y final y su implementación es muy sencilla. Su objetivo es hacer la llamada de forma correcta al predicado camino. Es importante remarcar que la lista de estados visitados inicialmente debe contener al estado inicial.

```
solucion(Camino):- inicial(Ei), final(Ef), camino(Ei, Ef, [Ei], Camino).
```

A continuación se muestra la implementación completa de la solución:

```

/* estado(Canibales_izq, Misioneros_izq, Pos_barca) */
inicial(estado(3,3,izq)).
final(estado(0,0,dch)).
/* pasar(?Num_misioneros, ?Num_canibales, ?Lugar)
   es cierto cuando Num_misioneros y Num_canibales unifica con una combinación
   válida de misioneros y misioneros válida según la especificación del problema y
   cuando lugar unifica con 'izq' o 'dch'. */
pasar(1,0,izq).
pasar(1,0,dch).
pasar(1,1,izq).
pasar(1,1,dch).
pasar(0,1,izq).
pasar(0,1,dch).
pasar(2,0,izq).
pasar(2,0,dch).
pasar(0,2,izq).
pasar(0,2,dch).

/*      mov(?Movimiento,      ?Estado_anterior,      ?Estado_posterior)
   es cierto cuando Movimiento unifica con un Movimiento válido, Estado_anterior
   unifica con un estado válido y Estado_posterior unifica con el estado resultante
   de aplicar el movimiento "Movimiento" al estado "Estado_anterior" */

mov(pasar(M, C, izq), estado(MI,CI, dch), estado(MD, CD, izq)):-
    pasar(M,C,izq),
    NT is M + C, NT <= 2, NT >= 1,
    M <= MI, C <= CI,
    MD is MI + M, CD is CI + C.

mov(pasar(M, C, dch), estado(MI, CI, izq), estado(MD, CD, dch)):-
    pasar(M,C,dch),
    NT is M + C, NT <= 2, NT >= 1,
    M <= MI, C <= CI,
    MD is MI - M, CD is CI - C.

/*      camino(+Estado_inicial,      +Estado_final,      +Visitados,      -Camino)
   es cierto cuando Estado_inicial y Estado_final unifican con estados válido, Visi-
   tados unifica con una lista
   */
camino(Inicio, Inicio, _, []).

camino(Inicio, Fin, Visitados, [Mov|Camino]):-
    length(Visitados, L), L < 10,
    mov(Mov, Inicio, Int),
    \+ member(Int, Visitados),
    camino(Int, Fin, [Int|Visitados], Camino).

```

Referencias: Problem Solving in Prolog: Bratko capítulo 12

<http://www.cse.unsw.edu.au/~billw/cs9414/notes/mandc/mandc.html>

PRÁCTICAS

PRÁCTICAS CON HASKELL

INTRODUCCIÓN AL ENTORNO DE PROGRAMACIÓN HASKELL HUGS

En esta primera práctica nos familiarizaremos con el entorno de programación de Hugs y veremos los tipos de datos básicos.

SECCIÓN 1. CUENTA DE USUARIO

Cada alumno puede disponer de una cuenta de usuario. La cuenta tiene asociado un espacio en disco. Se recomienda no dejar ningún material importante en esta cuenta. Utilizaremos este espacio, sólo como espacio de almacenamiento temporal. Después de cada sesión guardaremos los ficheros en una memoria USB o bien los enviaremos a nuestra cuenta de correo electrónico.

SECCIÓN 0. EMPEZANDO

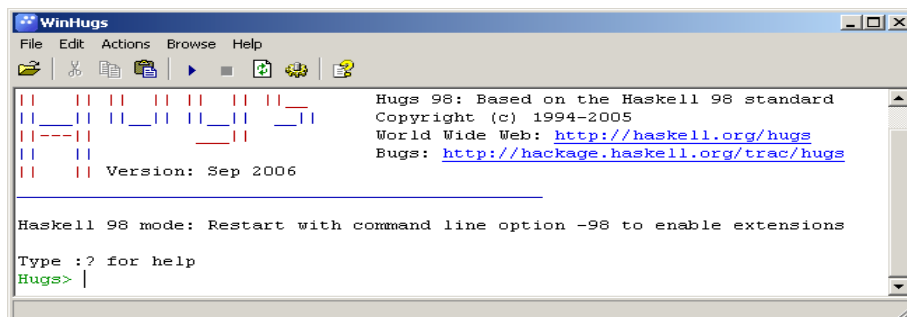
- Seleccionar el arranque del sistema Operativo Windows XP.
- Introducir el login proporcionado por el profesor/a. La primera vez que inicie el sistema debe cambiar el password. Recuerde su password, ya que no será posible consultar su password con posterioridad

SECCIÓN 1. EJECUTAR HUGS

- En el menú “Inicio” de Windows seleccionar el programa WinHugs



- Aparecerá una ventana como esta:
- Desde esta ventana cargaremos los programas y haremos las consultas. El modo de funcionamiento de esta ventana es muy similar al de una calculadora. Cuando escribimos una expresión en esta ventana y pulsamos ☐ , pedimos al interprete que nos muestre el resultado de la expresión.



Ejecutar en Hugs:

```
Hugs> 1+1
```

comprobar que el intérprete muestra “2”.

ejecutar

```
Hugs> :type $$
```

```
Hugs muestra "1+1 :: Num a => a"
```

La expresión inicial no se ha reducido, se ha utilizado una función “show” que muestra el resultado de la expresión, dejando esta expresión invariante.

SECCIÓN 2. ESCRIBIR UN PROGRAMA

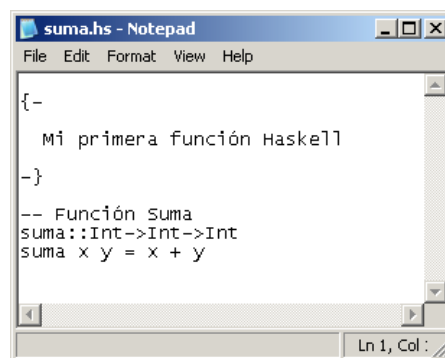
Para escribir un programa utilizaremos un editor de texto que no introduzca caracteres especiales. Nos servirá el bloc de notas de Windows. Hugs tiene configurado éste como editor por defecto. Podemos cambiar este editor utilizando el comando “:set”.

Antes de empezar con la edición de los ficheros, crearemos el directorio “H:\declarativa”.

La forma más sencilla de escribir un nuevo programa es introduciendo :edit <nombre_fichero> en el prompt de Hugs:

```
Hugs> :edit "H:\\declarativa\\suma.hs"
```

Este comando editará el fichero suma.hs. Escribiremos lo siguiente en el fichero suma.hs:



Guardaremos el contenido del fichero y cerraremos la ventana. Durante la edición del fichero no podremos acceder a la ventana de Hugs.

A continuación cargaremos el fichero. Para ello, escribiremos :load <nombre_fichero>:

```
Hugs> :load "H:\\declarativa\\suma.hs"
```

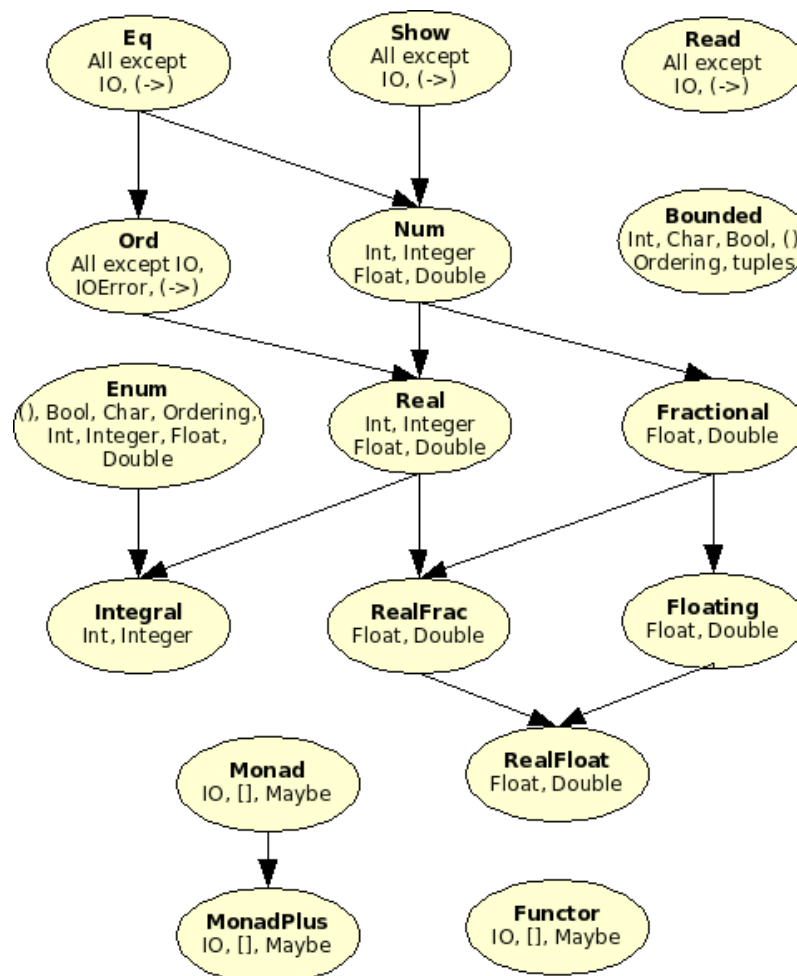
Podemos comprobar si se han cargado las funciones en memoria ejecutando `:info <nombre_función>`:

```
Hugs> :info suma
suma :: Int -> Int -> Int
```

El resultado es la cabecera de la función suma.

SECCIÓN 4. TIPOS DE DATOS

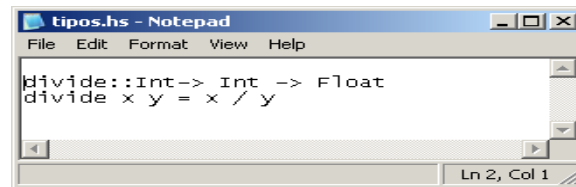
En el siguiente gráfico se muestra la relación entre las clases que agrupan los diferentes tipos de datos:



La clase Num, por ejemplo incluye a los tipos Int, Integer, Float y Double. La clase Fractional incluye al tipo Float y al tipo Double. A continuación veremos algunos ejemplos que muestran particularidades de Haskell con los tipos de datos:

```
Hugs> :edit "H:\\declarativa\\tipos.hs"
```

Escribiremos lo siguiente:



A continuación guardaremos el fichero y lo cargaremos en memoria:

```
Hugs> :load "H:\\declarativa\\tipos.hs"
```

Al cargar el fichero aparece el siguiente mensaje de error:

```
ERROR file:h:\declarativa\tipos.hs:3 - Type error in explicitly typed binding
*** Term           : divide
*** Type           : Int -> Int -> Int
*** Does not match : Int -> Int -> Float
```

Este error indica que no coincide la definición que hemos dado a la cabecera con la cabecera que Hugs ha deducido.

Para solucionar el problema, veamos qué tipo de dato esperaba la función “/” (divide). Ejecutaremos antes :load sin argumentos para que elimine de la memoria el fichero con el error y podamos proseguir.

```
Hugs> :load
```

```
Hugs> :info /
```

```
infixl 7 /
```

```
(/) :: Fractional a => a -> a -> a -- class member
```

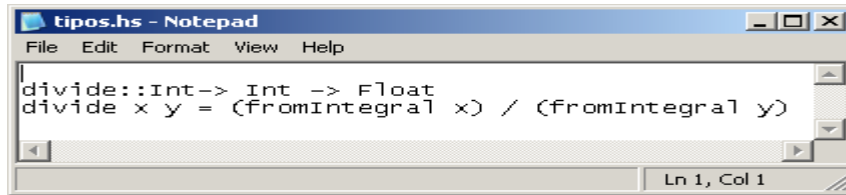
La cabecera indica que el tipo de dato que acepta la función “/” es de tipo `Fractional`, es decir `Float` o `Double` y que todos los argumentos son del mismo tipo. Por esta razón no podemos definir la función `divide` del tipo `Int->Int->Float`

Podemos solucionar este problema de dos formas.

a) La primera es dejando igual la cabecera y utilizando una de las funciones que convierten tipos.

En este caso utilizaríamos la función `fromIntegral`:

```
Hugs> :edit "H:\\declarativa\\tipos.hs"
```

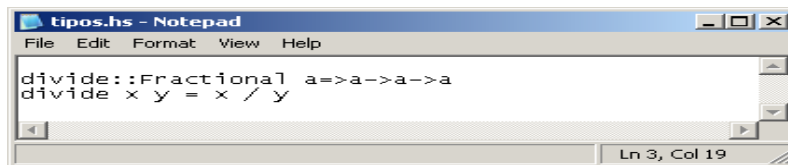


```
Hugs> :load "H:\\declarativa\\tipos.hs"
Main> divide 4 2
2.0
```

Nótese que el prompt **Hugs>** indica que no hay ningún módulo cargado y que el prompt **Main>** indica que hay al menos un fichero cargado en memoria.

- b) La segunda solución pasa por cambiar la cabecera para que acepte valores de entrada de tipo Fractional:

```
Main> :edit "H:\\declarativa\\tipos.hs"
```



```
Main> :load "H:\\declarativa\\tipos.hs"
Main> divide 4 2
2.0
```

Para terminar con los tipos de datos, recordar que aunque Haskell es capaz de inferir una cabecera para la función a partir de la implementación, nosotros escribiremos siempre la cabecera.

SECCION 3. RESUMEN

:t	Muestra el tipo de dato de la expresión
:type	Muestra el tipo de dato de la expresión
:?	Muestra la lista de comandos básicos
\$\$	Última expresión introducida
:load	Cargar un fichero
:edit	Abre el editor por defecto
:r	Actualiza los módulos cargados
:reload	Actualiza los módulos cargados
:set	Muestra la lista de opciones para activar/desactivar
:set +/-opción	Activa o desactiva una opción.
:set +t	Activa la opción que muestra el tipo de dato
:q	Salir de Hugs

Material complementario

- Utilizar el manual: “A tour of the Haskell Prelude”
<http://www.cs.ut.ee/~varmo/MFP2004/PreludeTour.pdf>
 para consultar las funciones más usuales definidas por defecto en Hugs.

Ejercicios

1. Implementar la función `doble` que recibe un número y devuelve el doble.
2. Implementar la función `cuadrado` que recibe un número y devuelve su cuadrado.
3. Implementar la función `esPositivo` que sea verdad cuando su único argumento sea mayor que 0
4. Dada la siguiente definición del máximo de dos números: $\max(x, y) = \frac{(x+y) + |x-y|}{2}$ implementar una función que devuelva el mayor de dos números de tipo `Float`.
5. Utilizando la función anterior, definir una nueva función que calcule el máximo de tres números.
6. Escribir la función `entre 0 y 9` que tome un entero y devuelva `True` si está entre 0 y 9 y `False` en otro caso.
7. Implementar la función `esMultiploDe3` que tome un entero y devuelva `True` si éste es múltiplo de 3 o `False` en otro caso.
8. Definir la función `loscuatroiguales::Int->Int->Int->Int->Bool` que devuelva `True` si los cuatro argumentos son iguales.
9. Utilizar la función anterior para crear una nueva llamada `lostresiguales` que reciba tres enteros y sea cierta si los tres enteros son iguales.
10. Definir la función `cuantosiguales::Int->Int->Int->Int` que recibe tres enteros y devuelve un entero indicando cuántos de los tres argumentos de entrada son iguales.
11. Implementar la función del ejercicio 4 utilizando sólo valores enteros.

Listas en Haskell

En esta sesión veremos algunos mecanismos de Haskell para definir listas.

SECCIÓN 1. LISTAS

En Haskell, una lista es una secuencia de valores del mismo tipo, por ejemplo `[1,2,3,4,5]` es una lista de 5 enteros (tipo `Int`), por otro lado `[True, False, False, True]` es una lista de 4 elementos booleanos (tipo `Bool`).

Haskell permite definir listas infinitas, por ejemplo `[1..]` es una lista infinita de enteros.

Una cadena de texto (tipo `String`), es una lista de caracteres (tipo `Char`). Si pedimos a Haskell que evalúe la lista `['a', 'b', 'c']` responderá que el resultado es `"abc"`.

Haskell proporciona un mecanismo para definir fácilmente listas.

Evaluar las siguientes listas en el prompt de Hugs:

- | | |
|------------------------------|------------------------------|
| a) <code>[1..10]</code> | d) <code>['q' .. 'z']</code> |
| b) <code>[15..20]</code> | e) <code>[14 .. 2]</code> |
| c) <code>[15..(20-5)]</code> | |

¿Los resultados son los esperados? Para números, caracteres y otros tipos enumerados (cualquier tipo en el que tenga sentido hablar del sucesor), la expresión `[m..n]` genera la lista `[m,m+1,m+2,...,n]`. Si `n` es menor que `m` (como en el ejemplo e) anterior), Haskell devuelve una lista vacía.

De forma más general, la expresión `[m,p..n]` genera la lista:

$$[m, m+(p-m), m+2(p-m), m+3(p-m) \dots n']$$

dónde `n'` es el mayor valor de la forma $m+k*(p-m) \leq n$. Con el fin de aclarar esta notación, ejecutar los siguientes ejemplos en Hugs:

- | | |
|-------------------------------|--------------------------------|
| a) <code>[1,3..10]</code> | d) <code>['a','d'..'z']</code> |
| b) <code>[15,15.5..20]</code> | e) <code>[10,8 .. -3]</code> |
| c) <code>[1,1.2..2.0]</code> | f) <code>[1,2 .. 0]</code> |

¿Qué sucederá al evaluar esta expresión: `[1,1..5]`?

Utilizar el menú Actions -> Stop, si fuese necesario.

SECCIÓN 2. NOTACIÓN EXTENDIDA DE LISTAS

La definición de una lista con esta definición consta de tres partes: 1) Generador, 2) Restricciones (puede que no haya ninguna) y 3) Transformación. El generador produce elementos de una lista, las restricciones filtran algunos elementos de los generados y la transformación utiliza los elementos seleccionados para generar una lista resultado.

Ejemplo:

```
[(x,True) | x <- [1..20], even x, x < 15]
```

En la expresión anterior, `x <- [1..20]` es el generador, las restricciones son `even x` y `x < 15`; y `(x, True)` es la transformación. Esta expresión genera una lista de pares `(x,True)` donde `x` está entre 1 y 20, es par y menor que 15 (`even` es una función definida en el módulo `Prelude` estándar de Haskell). Compruebe como los elementos aparecen en el orden generado.

Comprobar el resultado anterior, con la evaluación de:

```
[(x,True)|x<-[20,19..1],even x, x < 15]
```

De nuevo, los elementos aparecen en el orden que fueron generados.

Evaluar:

```
[[m+n] | (m,n) <- [(3,6), (7,3), (8,4), (1,3), (4,8)], n==2*m]
```

Esta expresión, genera una lista de listas de enteros, con aquellos pares cuya segunda componente es el doble que la primera. Esta expresión demuestra como utilizar plantillas `(m,n)` para aislar una determinada componente de la tupla.

Ahora probar la siguiente expresión:

```
[(x,y) | x <- [1..5], y == 1]
```

En este caso, Haskell muestra el siguiente error `Undefined variable “y”`. Las variables se definen en el generador. Si una variable no aparece en esta sección, no es posible utilizarla en otra.

Podemos utilizar la notación extendida de listas para definir funciones más generales. Por ejemplo, la siguiente función es una generalización de la expresión anterior:

```
aniadirPares::[(Int,Int)] -> [[Int]]
aniadirPares listaPares = [[m+n] | (m,n) <-
    listaPares, n == 2*m]
```

La declaración de tipos de la función especifica que acepta una lista de pares de enteros y devuelve una lista de listas de enteros.

Probar la función con los siguientes argumentos:

```
aniadirPares [(3,6),(7,3),(8,4),(1,3),(4,8)]
aniadirPares [(3,7),(8,9),(1,3)]
```

Por último, el siguiente ejemplo muestra cómo utilizar la notación extendida de listas como parte de una función un poco más compleja. Las siguientes funciones indican cuándo una lista contiene sólo valores pares (o sólo valores impares).

```
todosPares, todosImpares :: [Int] -> Bool
todosPares lista = (lista == [x | x <- lista, x `mod` 2 == 0])
todosImpares lista = ([ ] == [x | x <- lista, x `mod` 2 == 0])
```

Por ejemplo, evaluar las siguientes expresiones:

```
todosPares [1 .. 20]
todosImpares [1 .. 20]
todosPares [1,3 .. 20]
todosImpares [1,3 .. 20]
todosPares [ ]
```

SECCIÓN 3. UTILIZACIÓN DE PLANTILLAS CON LISTAS

La siguiente función utiliza la notación extendida de listas. Recibe una lista como argumento y devuelve otra lista con todos sus elementos multiplicados por 2.

```
doblarTodos :: [Int] -> [Int]
doblarTodos lista = [2*x | x <- lista]
```

Esta definición de lista incluye el generador `x <- lista` y una transformación `2*x`. El resultado de la lista contiene siempre el mismo número de elementos que la lista original puesto que la definición no incluye ninguna restricción. Por ejemplo, `doblarTodos [1,2,3,4,5]` devuelve una lista con cinco elementos.

A continuación, una versión de `doblarTodos` utilizando recursividad:

```
doblarTodos :: [Int] -> [Int]
doblarTodos [] = []
doblarTodos (cab:resto) = 2*cab : doblarTodos
    resto
```

A continuación, una versión recursiva de la función `aniadirPares`, implementada en la sección anterior:

```
aniadirPares [] = []
aniadirPares ((m,n):resto) = [m+n] : aniadirPares
    resto
```

Un vez más, podemos comprobar cómo la definición recursiva utiliza exactamente la misma transformación que la implementación utilizando notación extendida de listas.

SECCIÓN 4. SELECCIONANDO ELEMENTOS

Hemos visto que utilizando la notación extendida de lista podemos filtrar algunos elementos de la lista original añadiendo restricciones. Por ejemplo, la siguiente función acepta una lista de enteros y devuelve el doble de todos los elementos menores que 10, eliminando el resto:

```
doblarAlgunos :: [Int] -> [Int]
doblarAlgunos lista = [ 2*x | x <- lista, x < 10]
```

A continuación una versión de la función `doblarAlgunos` utilizando recursividad:

```
doblarAlgunos [] = []
doblarAlgunos (cab:resto)
  | cab < 10 = 2*cab : doblarAlgunos resto
  | otherwise = cab : doblarAlgunos resto
```

Esta función es similar a la función `doblarTodos` excepto porque no se añaden todos los elementos $2*x$. En esta versión sólo añadimos a la lista aquellos elementos que satisfacen que $x < 10$.

Otro ejemplo similar es esta implementación de la función `aniadirPares` utilizando recursividad:

```
aniadirPares [] = []
aniadirPares (m,n):resto
  | n == 2*m = [m+n] : aniadirPares resto
  | otherwise = aniadirPares resto
```

SECCIÓN 5. COMBINANDO RESULTADOS

La notación extendida de listas es útil para convertir listas en otras nuevas listas, sin embargo, no proporciona un modo de convertir listas en algo que no sea una lista. Por ejemplo, la notación extendida de listas no sirve para encontrar el máximo de una lista.

Podemos utilizar para este propósito el mismo tipo de plantillas que hemos visto en las funciones recursivas. La idea es que podemos combinar un elemento de la lista con el resultado obtenido por la llamada recursiva.

Por ejemplo, consideremos la siguiente función que suma los elementos de una lista de enteros:

```
sumaLista :: [Int] -> Int
sumaLista [] = 0
sumaLista (cab:resto) = cab + sumaLista resto
```

Podemos comprender cómo esta definición es similar a la que hemos visto anteriormente, salvo por el uso del operador “+” para combinar los resultados, en vez del operador “:”; y por que no tiene sentido devolver una lista vacía en la primera ecuación, ya que sumaLista devuelve un entero.

A continuación veremos otros dos ejemplos de funciones recursivas que combinan el primer elemento de la lista con el resultado de evaluar el resto de misma. En la primera función se utiliza el operador “+” para sumar el cuadrado (cabeza*cabeza) del primer elemento:

```
sumaCuadrados :: [Int] -> Int
sumaCuadrados [] = 0
sumaCuadrados (cab:resto) = cab*cab + sumaCuadrados
    resto
```

En el segundo ejemplo se utiliza el operador “++” para concatenar la primera de las listas con el resultado de evaluar el resto de una lista de listas:

```
concatena :: [[a]] -> [a]
concatena [] = []
concatena (cab:resto) = cab ++ concatena resto
```

Ejercicios

Generar cada una de las siguientes listas utilizando la notación extendida de listas, con la lista [1..10] como generador. Es decir, cada solución debe tener la siguiente forma, donde se deben completar los blancos y sólo los blancos:

[_____ | x <- [1 .. 10] _____]

De forma más explícita, la respuesta debe utilizar el generador x<- [1.. 10], y no debe añadir a esta definición ninguna llamada a función: por ejemplo, no utilizar reverse [x | x <- [1 .. 10]] para crear la lista [10,9,8,7,6,5,4,3,2,1]. De la misma forma, modificaciones del tipo [x|x <- [10,9..1]] y [x|x <- reverse[1 ..10]] también están prohibidas.

1. [11,12,13,14,15,16,17,18,19,20]
2. [[2],[4],[6],[8],[10]]
3. [[10],[9],[8],[7],[6],[5],[4],[3],[2],[1]]
4. [True,False,True,False,True, False,True,False,True,False]
5. [(3,True),(6,True),(9,True),(12,False),(15,False),(18,False)]
6. [(5,False),(10,True),(15,False),(40,False)]
7. [(11,12),(13,14),(15,16),(17,18),(19,20)]
8. [[5,6,7],[5,6,7,8,9],[5,6,7,8,9,10, 11],[5,6,7,8,9,10,11,12,13]]
9. [21,16,11,6,1]
10. [[4],[6,4],[8,6,4],[10,8,6,4],[12,10,8,6,4]]

Patrones, tuplas, recursividad y notación extendida de listas en Haskell

En esta sesión realizaremos ejercicios utilizando **patrones, tuplas, recursividad y notación extendida de listas**.

1. Implementar la función divisores que recibe un argumento entero y que devuelva la lista de sus divisores.

```
Main> divisores 9
[1,3,9]
```

2. Utilizando la función anterior, programar la función primo que devuelva verdadero en caso de que su único argumento entero sea un número primo. No consideraremos al número 1 como primo.
3. Crear una expresión con la que se obtengan los primos entre 1 y 100. Utilizar la notación extendida de listas para este ejercicio.
4. Averiguar cómo funcionan las funciones map y filter e implementarlas utilizando la notación extendida de listas. Llamar a las nuevas funciones mapea y filtra.
5. Programar una función evaluaciones con la siguiente cabecera
`evaluaciones::[a]->[(a->b)]->[[b]]`

La lista de listas resultante contiene listas con los resultados de aplicar a cada uno de los valores de la primera lista las funciones de la segunda lista. Por ejemplo:

```
Main> evaluaciones [1,2,3] [doble, triple]

[[2,3],[4,6],[6,9]]
```

6. Utilizar la función anterior para evaluar si los siguientes valores $[0, (3.14/2), ((-3.14)/2), 3.14, (-3.14)]$ cumplen que el seno es mayor que 0, el coseno es 0 y la tangente es 0. Componer la lista de funciones utilizando el operador de composición “.”. El resultado para este ejemplo será:

```
[ [False, False, True], [True, False, False], [False, False, False], [True, False, False], [False, False, False] ]
```

7. Implementar una función que devuelva la descomposición en factores primos de un número entero. La función devolverá una lista de tuplas tal que la primera componente será el factor primo y la segunda será el número de veces que dicho factor primo divide al argumento original.

Ejemplo:

```
Main> descomposicion 60
[(2,2),(3,1),(5,1)]
```

- 8.** Averiguar qué devuelven las funciones `takeWhile` y `dropWhile` e implementar su propia versión. Llamar a las nuevas funciones `tomarMientras` y `eliminarMientras`.

```
Main> tomarMientras (<5) [1..10]
[1,2,3,4]
```

```
Main> eliminarMientras (<5) [1..10]
[5,6,7,8,9,10]
```

- 9.** Programar la función `quita_blanco` que elimine los blancos iniciales de una cadena de caracteres.

```
Main> quitaBlancos "   bcd fgh"
"bcd fgh"
```

- 10.** Revisar los ejercicios realizados e intentar localizar los puntos en los que se utilizó alguna de las características más importante de Haskell: *currificación y funciones de orden superior*.

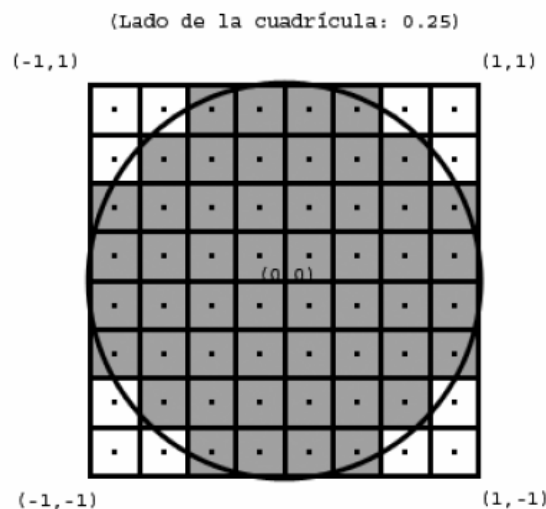
Ejercicios de Haskell de exámenes anteriores

En esta última sesión de Haskell realizaremos algunos ejercicios aparecidos en exámenes anteriores.

1. Implementar una función que aproxime el cálculo de la integral de una función en el intervalo $[a, b]$ y dado un factor de precisión t .

`integral funcion a b t`

2. **(Febrero 2006)** Implementar una función que aproxime el valor de π . Para obtener el valor aproximado de π utilizaremos un cuadrado de lado 2. En el centro del cuadrado fijaremos en centro de referencia $(0,0)$. Haremos una rejilla dentro del cuadrado de lado t . Por último, contaremos cuantos centros de los cuadrados de la rejilla están dentro del círculo. Esto nos dará un valor aproximado de π . Cuanto menor sea t , más precisión tendrá la aproximación.



Área del círculo:

$$A = \pi * r^2$$

Distancia entre los puntos (x_1, y_1) y (x_2, y_2) :

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

3. **(Noviembre de 2006)** Implementar en Haskell una función que calcule la clave hash de una cadena. La función hash utilizada es:

$$f([a_1, a_2, \dots, a_n]) = a_1 * p^n + a_2 * p^{n-1} + \dots + p^1 * a_n.$$

Donde los a_i son los caracteres incluidos en la lista y p_i son los i -ésimos números primos ($p_1=2, p_2=3, p_3=5, p_4=7 \dots$).

4. (Septiembre 2007) Suponiendo que se implementan en Haskell los conjuntos como LISTAS SIN REPETICION, implementar las funciones:

pertenece elemento conjunto: devuelve True si un elemento pertenece a ese conjunto, False en caso contrario.

subconjunto conjunto1 conjunto2: devuelve True si el primer argumento es subconjunto del segundo.

iguales conjunto1 conjunto2: devuelve True si dos conjuntos son iguales.

union conjunto1 conjunto2: devuelve una lista que es la unión de los dos conjuntos (SIN REPETICIONES).

PRÁCTICAS CON PROLOG

Introducción al entorno SWI-Prolog

En esta primera práctica nos familiarizaremos con el entorno de programación de SWI-Prolog.

SECCIÓN 1. EMPEZANDO

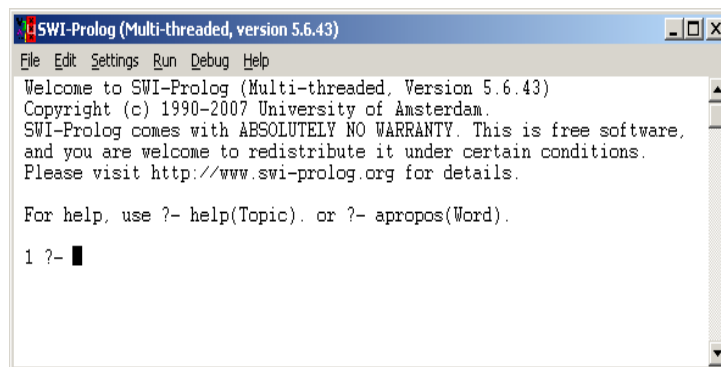
- Seleccionar el arranque del sistema Operativo Windows XP.

SECCIÓN 4. EJECUTAR SWI-PROLOG

- En el menú “Inicio” de Windows seleccionar el programa SWI-Prolog



- Aparecerá una ventana como ésta:



- Desde esta ventana cargaremos los programas y haremos las consultas.

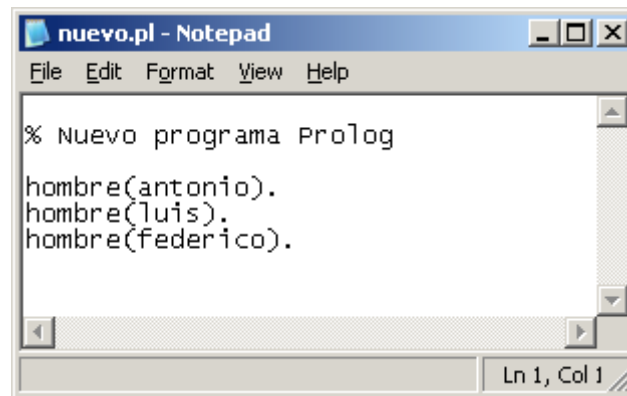
SECCIÓN 5. ESCRIBIR UN PROGRAMA

Para escribir un programa utilizaremos un editor de texto que no introduzca caracteres especiales. Nos servirá el bloc de notas de Windows. SWI-Prolog tiene configurado este editor por defecto.

La forma más sencilla de escribir editar un nuevo programa es escribir

```
edit(file('nombre_fichero.pl')).
en el prompt de SWI-Prolog:
1? - edit(file('nuevo.pl')).
```

Este comando editará el fichero ‘nuevo.pl’. Utilizaremos la extensión “.pl” para los programas Prolog. El fichero se guardará por defecto en el directorio “Mis documentos\Prolog”. Escribiremos lo siguiente teniendo en cuenta que los predicados y las constantes empiezan con minúscula:



Guardaremos el contenido del fichero y cerraremos la ventana. Durante la edición del fichero no podemos acceder a la ventana de SWI-Prolog.

Cargaremos el fichero con la orden `consult('<nombre_fichero>')`.

```
? - consult('nuevo.pl').
```

Podemos comprobar si, efectivamente se cargó el fichero utilizando la orden `listing`, que mostrará el programa cargado.

```
? - listing.
hombre(antonio).
hombre(luis).
hombre(federico).
Yes
?-
```

A partir de este momento podemos empezar a hacer consultas

```
? - hombre(antonio).
```

Yes

```
? - hombre(federico).
```

Yes

```
? - hombre(manuel).
```

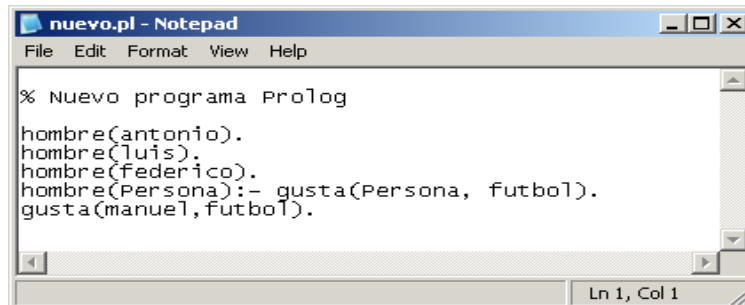
No.

Por la *hipótesis del mundo cerrado*, lo no definido es falso.

Modificaremos el programa anterior para incluir las siguientes líneas:

```
hombre(persona):- gusta(Persona,futbol).
gusta(manuel,futbol).
```

Tendremos en cuenta que *Persona* es una variable y la escribiremos en mayúscula.



```
% Nuevo programa Prolog
hombre(antonio).
hombre(luis).
hombre(federico).
hombre(Persona):- gusta(Persona, futbol).
gusta(manuel,futbol).
```

Después de modificar un fichero, actualizamos los cambios ejecutando la orden “make”.

```
?- make.
```

Realizaremos ahora la consulta anterior ¿Hombre manuel?

```
? hombre(manuel).
```

```
Yes
```

Operadores

Los operadores más utilizados en Prolog son “=”, “==”, “is” y “:=”.

“=” leeremos unificación.

Dos términos unifican:

- 1) Si no tienen variables, unifican si son idénticos (iguales carácter a carácter).
- 2) Si tienen variables, unificarán si es posible encontrar una sustitución de las variables de forma que lleguen a ser idénticos.

El operador de unificación no evalúa operaciones aritméticas. Antes de unificar una variable a un valor diremos que la variable está “libre” o “sin instanciar”. Tras unificar un valor a una variable, la variable deja de estar libre y diremos que la variable está instanciada. Una variable, una vez que se instancia, no cambia de valor.

```
?- X = 1+1.
```

```
X = 1+1.
```

```
Yes
```

“==” identidad

```
?- 2 == 1+1.
```

No.

”is” evalúa expresiones aritméticas. Evalúa a la derecha y unifica con la izquierda.

```
?- X is 1+1.
```

```
X = 2
```

```
?-
```

“:=” evalúa expresiones aritméticas y compara. Evalúa expresiones aritméticas a derecha e izquierda y es cierto si el resultado de las evaluaciones es el mismo valor.

```
?- 2 := 1+1.
```

Yes

SECCIÓN 6. ÁMBITO DE UNA VARIABLE

En la mayoría de los lenguajes de programación, si la variable se define dentro de una función, la variable puede ser referenciada desde cualquier punto de la función. Decimos que el ámbito de la variable es la función en la que está definida.

En Prolog el ámbito de una variable está restringida a una cláusula. Fuera de una cláusula la variable no puede ser referenciada. Esta es una de las diferencias más importantes de Prolog respecto a la mayoría de lenguajes de programación. De este modo, en el programa Prolog:

```
nombre_numero(X, 0):- X = cero.
nombre_numero(X, 1):- X = uno.
```

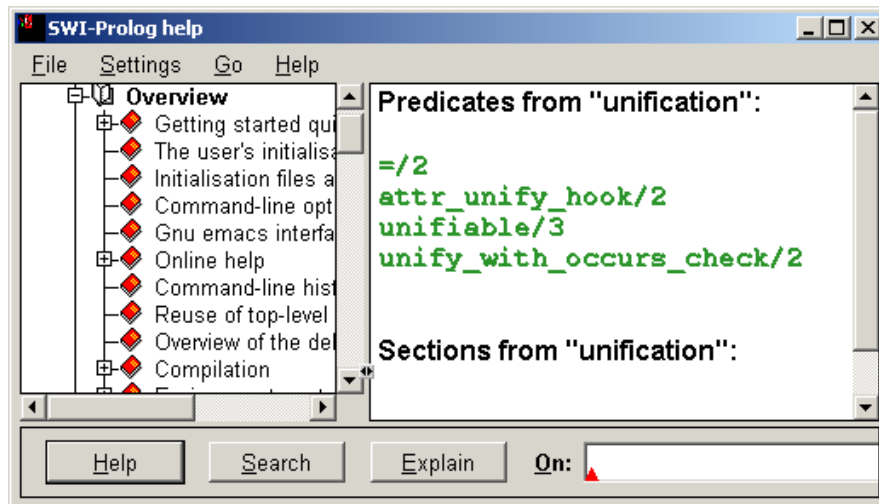
la variable X que aparece en el consecuente de la primera cláusula es la misma que la que aparece en el antecedente de esa misma cláusula pero es diferente de la variable X que aparece en consecuente y antecedente de la segunda cláusula.

SECCIÓN 7. PREDICADOS DE AYUDA

Podemos consultar información sobre cualquier predicado con el comando “apropos”

```
apropos(unification).
```


Aparecerá la siguiente ventana:



Invocaremos a esta misma ventana con el predicado `help`. Los predicados en Prolog se nombran de la siguiente forma `nombre_predicado/aridad`. La aridad es el número de argumentos que tiene el predicado. Por ejemplo, el predicado unificación `=/2`, tiene aridad 2.

SECCIÓN 8. DEPURADOR

Habilitaremos el depurador gráfico con el predicado “`guitracer`”.

```
?- guitracer.
```

```
% The graphical front-end will be used for subsequent tracing
```

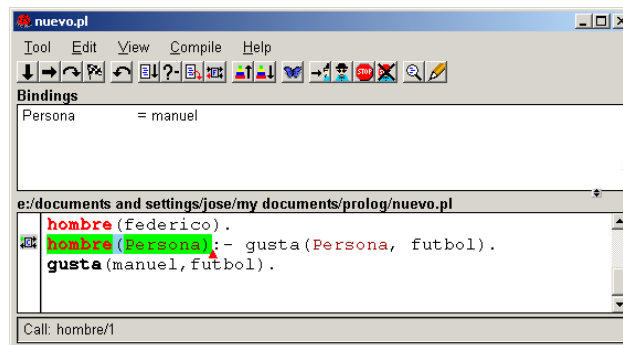
```
Yes
```

Para depurar un programa Prolog utilizaremos el comando “`trace`”. La siguiente consulta que realicemos después del predicado “`trace`” se ejecutará paso a paso. Si el depurador gráfico está habilitado se mostrará la siguiente ventana. Para avanzar la ejecución pulsaremos la barra espaciadora o utilizaremos los iconos de avance y retroceso.

```
?- trace.
```

```
Yes
```

```
? - hombre(manuel).
```



Ejercicios

1. Unificación.

Comprobar si unifican los siguientes términos:

- (a) $X=1$.
- (b) $X = mujer(maria)$.
- (c) $X = 2 + 3$.
- (d) $X=1, X=2, X=Y$.
- (e) $X=2+X$.
- (f) $X=1, Y=1, X=Y$.
- (g) $X=2, Y=1+1, X = Y$.
- (h) $X=3+4, Y= 7, X = Y$.
- (i) $X = 2*6, X=2*Y$.
- (j) $5 = 2 + 3$.
- (k) $2 + 3 = 2 + Y$.
- (l) $f(X, b) = f(a, Y)$.
- (m) $2*X = Y*(3+Y)$.
- (n) $(1+2) = 1+2$.
- (o) $1+1 = +(1,1)$.
- (p) $(1+1)+1 = 1+(1+1)$.

2. Ejercicios de Aritmética y Unificación.

Comprobar si unifican los siguientes términos:

- (a) $X \text{ is } 2 + 3, X = 5$.
- (b) $X \text{ is } 2 + 3, X = 2 + 3$.
- (c) $2 + 3 \text{ is } X$.
- (d) $X \text{ is } 5, 6 \text{ is } X + 1$.
- (e) $X = 5, 6 = X + 1$.
- (f) $6 \text{ is } X + 1, X = 5$.
- (g) $Y = 2, 2*X \text{ is } Y*(Y+3)$.
- (h) $Y = 2, Z \text{ is } Y*(Y+3)$.

Predicados sencillos en Prolog

En esta sesión implementaremos algunos predicados sencillos utilizando recursividad en la mayoría de los casos.

Ejercicios

1. Implementar `natural(X)` que será cierto si `X` es un número natural.

```
?- nat(6).
yes
```

```
?- nat(-13).
no
```

2. Implementar el predicado Prolog `factorial(Número, Resultado)` que será cierto si `Resultado` unifica con el factorial de “Número”.

3. Implementar el predicado `fib(N, F)`. Que será cierto cuando “F unifique con el N-ésimo número de Fibonacci”. Estos números son:

```
fib(1)=1,
fib(2)=1,
fib(3)=2,
fib(4)=3,
fib(5)=5,
.....,
fib(n)=fib(n-1) + fib(n-2),
```

4. Dada una serie de platos, clasificados en primer plato, segundo plato y postre con su respectivo precio asociado, elaborar un predicado que encuentre un menú completo por un coste menor a `N`. Ejemplo:

```
menu(Primer_plato, Segundo_plato, Postre, Precio),
Precio < 50.

Primer_plato=sopa
Segundo_plato=jamón
Postre=helado
```

5. Implementar el predicado `suma(X,Y,Z)` que representa la suma de dos números naturales utilizando la representación del matemático italiano **Giuseppe Peano** (1858–1932) que está basada en la utilización del símbolo 0 y el predicado `s(X)` que representa el siguiente de `X`.

```
0=0
1 = s(0)
2 =s(s(0))
3=s(s(s(0)))
```

.....

`suma (s (0) , s (0) , Z) .`

`Z=s (s (0))`

`Yes`

6. Utilizando la representación del ejercicio anterior, implementar los predicados `resta(X,Y,Z)` y `producto(X,Y,Z)`.

7. Implementar el ejercicio 3 de forma que el predicado `fib(N,F)` sea reversible, es decir, que sea posible hacer la llamada dejando el primer argumento libre e instanciando el segundo. Este ejercicio es algo más complicado que los anteriores y para solucionarlo será necesario utilizar la aritmética de G. Peano.

Ejemplo:

`fib (N, s (s (0)))`

`N=s (s (s (0)))`

`Yes`

Predicados sobre listas en Prolog

En esta sesión implementaremos algunos predicados sobre listas.

Ejercicios

1. Tratar de predecir la unificación que resulta en los siguientes ejemplos y confirmar luego ejecutando cada uno:

```
?- [X| Y] = [a, b, c, d].
?- [X, Y| Z] = [a, b, c].
?- [X, Y| Z] = [a, b, c, d].
?- [X, Y, Z| A] = [a, b, c].
?- [X, Y, Z| A] = [a, b].
?- [X, Y, a] = [Z, b, Z].
?- [X, Y| Z] = [a, W].
```

2. Definir en Prolog los siguientes predicados recursivos para manejar listas:

- a) Dados dos elementos, crear una lista con esos dos elementos.
- b) Insertar un elemento en una lista:
 - 1) Al comienzo de la lista.
 - 2) Al final de la lista.
 - 3) En la posición N de una lista.
- c) Concatenar dos listas y usar ese predicado para implementar los predicados:
 - 1) prefijo
 - 2) sufijo
 - 3) sublista
- d) Invertir una lista.
- e) Borrar un elemento en una lista:
 - 1) Borrando sólo una aparición del elemento.
 - 2) Borrando todas las apariciones de ese elemento.
- f) Cambiar un cierto elemento por otro:
 - 1) Cambiar sólo una aparición del elemento.
 - 2) Cambiar todas las apariciones de ese elemento.
- g) Dada una lista de longitud n, generar otra lista de longitud 2n que sea un palíndromo.

h) Desplazar una posición a la derecha todos los elementos de una lista. Por ejemplo, pasar de la lista $[x_1, x_2, \dots, x_n]$ a la lista $[x_n, x_1, \dots, x_{n-1}]$.

i) Desplazar una posición a la izquierda todos los elementos de una lista. Por ejemplo, pasar de la lista $[x, x_2, \dots, x_n]$ a la lista $[x_2, \dots, x_n, x_1]$.

3. Definir un predicado `dividir(+N, +LOrig, -May, -Men)` que se cumpla si la lista `May` contiene los elementos mayores de un número `N` de la lista `LOrig` y `Men` contiene los elementos menores de un número `N` de la lista `LOrig`.

```
?-dividir(4, [3,1,2,5,0], Mayores, Menores).
Mayores = [5]
Menores = [1, 2, 3, 0]
```

4. Definir un predicado `mezclar_ord(+L1, +L2, -Resul)` de forma que, siendo `L1` y `L2` dos listas ordenadas, se cumple que la lista `Resul` contiene la mezcla de los elementos de las dos listas `L1` y `L2`, y es también una lista ordenada.

```
?-mezclar_ord([1,2,7], [0, 3,5], Resul).
Resul = [0, 1, 2, 3, 5, 7]
```

5. Definir un predicado `ordena(Lista, R)` que se cumple si la lista `R` contiene los elementos ordenados de `Lista`.

```
?-ordena([3,1,2], V).
V = [1, 2, 3]
```

a) Como permutaciones de una lista, hasta que encuentre la ordenada.

b) Como separación de listas, ordenación y mezcla.

6. Escribir un predicado `prod(+L, ?P)` que significa “`P` es el producto de los elementos de la lista de enteros `L`”. Debe poder generar la `P` y también comprobar una `P` dada.

7. Escribir un predicado `pEscalar(+L1, +L2, -P)` que significa “`P` es el producto escalar de los dos vectores `L1` y `L2`”. Los dos vectores vienen dados por las dos listas de enteros `L1` y `L2`. El predicado debe fallar si los dos vectores tienen una longitud distinta.

8. `divisores(+N, ?L)`. Dado un natural `N`, `L` es la lista de divisores de `N` en orden creciente. Por ejemplo, si `N` es 24, `L` será $[1, 2, 3, 4, 6, 8, 24]$. Debe contestar YES si una `L` dada es la lista de los divisores de un `N` dado y NO en caso contrario; y debe poder generar `L` para un `N` dado. Con `divisores(6, [2, 3, 6, 1])` ha de responder NO (la lista no esta ordenada!).

9. `permuta(+L, -P)`. “La lista `P` contiene una permutación de los elementos de la lista `L`”. La lista `L` inicialmente estará instanciada, y `P` no. Ejemplo: `L = [1, 2, 3]` daría `P = [1, 2, 3]`, `P = [1, 3, 2]`, `P = [2, 1, 3]`, etc. En el caso de que la lista a permutar tuviera elementos repetidos, deben permitirse permutaciones repetidas.

10. Un conjunto puede ser modelado mediante una lista de elementos sin repeticiones. Adoptando esta representación, implementar las siguientes operaciones sobre conjuntos en lenguaje Prolog.

- a) Determinar si un elemento pertenece a un conjunto.
- b) Incorporar un elemento a un conjunto.
- c) Unir dos conjuntos.
- d) Hallar la intersección de dos conjuntos.
- e) Calcular la diferencia entre dos conjuntos.
- f) Dada una lista de elementos con repeticiones, construir un conjunto que contenga todos los elementos de esa lista.

11. Un MULTI-conjunto puede ser modelado mediante una lista de elementos (elemento, multiplicidad). Adoptando esta representación, implementar las siguientes operaciones sobre conjuntos en lenguaje Prolog.

- a) Determinar si un elemento pertenece a un conjunto.
- b) Calcular la multiplicidad de un elemento.

Árboles en Prolog

Árboles binarios

1. Implementar el predicado `cuenta_nodos/2` que cuente el número de nodos que tiene un árbol binario.
2. Implementar el predicado `cuenta_internos/2` que cuente el número de nodos INTERNOS (que tienen al menos un hijo) que tiene un árbol binario.
3. Implementar el predicado `cuenta_hojas/2` que cuente el número de nodos HOJA (que no tienen ningún hijo) que tiene un árbol.
4. Implementar el predicado `suma_nodos/2` que sume el contenido de todos los nodos de un árbol.
5. Escribir el predicado `miembro/2` que indique si un elemento X, pertenece a un árbol.
6. Implementar los predicados `iguales/2`, `simetricos/2` e `isomorfo/2` que serán ciertos cuando dos árboles cumplan dichas propiedades.
7. Implementar el predicado `profundidad/2` que será cierto cuando el segundo argumento unifique con la profundidad del árbol.
8. Implementar el predicado `balanceado/1` que será cierto cuando el árbol esté balanceado.
9. Implementar los siguientes predicados que transforman los recorridos de un árbol en una lista.

```
inorden/2
preorden/2
postorden/2
anchura/2 (de izquierda a derecha)
```

Árboles genéricos

Repetir los ejercicios anteriores, excepto los recorridos en `inorden`, `preorden` y `postorden` del ejercicio 9, considerando árboles genéricos en lugar de binarios.

Árboles binarios de búsqueda,

Implementar en Prolog los siguientes predicados:

`crearArbolVacio(X)`: retorna en X un árbol vacío.

`insertar(E, A, NA)`: inserta el elemento E en el árbol A, retornando el nuevo árbol en NA (asumir que si el elemento a ser insertado ya pertenece al árbol, entonces se retorna el mismo árbol).

`altura(A, X)`: retorna en X la altura del árbol A.

`equilibrado(A)`: determina si el árbol A está equilibrado o no.

`recorridos(A, Pre, Post, In)`: a partir del árbol A retorna en Pre, Post e In sus recorridos en pre-orden, postorden e inorden respectivamente.

Implementar un predicado que construya un árbol binario de búsqueda a partir de una lista de números enteros.

Ejemplo:

```
?- construir([3,2,5,7,1],T) .  
T = t(3,t(2,t(1,nil,nil),nil),t(5,nil,t(7,nil,nil)))
```

Grafos en Prolog

1. Escribir un programa Prolog que pinte un sobre como el de la figura sin levantar el lápiz del papel, es decir, recorre el grafo pasando exactamente una vez por cada arco.

2. Representar el grafo de la figura en Prolog:

Escribir un programa Prolog que encuentre todos los caminos posibles entre Madrid y Oviedo, construyendo una lista de las ciudades por las que pasa.

Añadir los elementos necesarios para que calcule:

- La lista de las carreteras por las que pasa cada camino.
- La distancia recorrida por cada uno de ellos.

¿Qué ocurriría si el grafo propuesto no fuera dirigido y acíclico como el propuesto? Para analizar esta posible circunstancia, introducir dos arcos más en el grafo:

- León-Palencia (N 610,130 Km.)
- Palencia-Valladolid (N 620, 47 Km.)

y comprobar el comportamiento del programa.

Proponer una solución para evitar los bucles e implementarla.

3. Un grafo puede ser expresado como $G = (V, A)$, donde V es el conjunto (lista sin repetición) de vértices y A el conjunto de arcos (a_i, a_j) , etiquetados por la distancia que separa al vértice a_i del vértice a_j .

Crear un programa en Prolog con los predicados:

- `camino(G,N1,N2,Cam)`, que encuentre el camino entre dos nodos y devuelva una lista con los nodos por donde pasa. Mediante backtracking deberá encontrar todos los caminos posibles.
- `distancia(G,N1,N2,D)`, que calcule la distancia entre cualquier par de vértices. Implementar dos versiones: una que admita ciclos y otra que no.
- `ciclos(G,Ciclo)`, que encuentre los ciclos dentro de un grafo. Esto consiste en encontrar un camino que vaya de un nodo a él mismo.
- `conexo(G)`, que compruebe que un grafo es totalmente conexo. Se dice que un grafo es totalmente conexo cuando existen caminos entre todos sus nodos.
- `grado(G,N,Gr)` que nos diga el grado de un nodo dentro de un grafo. El grado de un nodo es el número de nodos con los que se conecta.
- `listagrados(G,L)` que genere una lista con todos los nodos y sus grados.
- `pintargrafo(G,LC,LNod)` que pinte el grafo G con una lista de colores LC y nos devuelva una lista $LNod$ con cada uno de los nodos y su color, de forma que dos nodos adyacentes no pueden tener el mismo color.

