Programación Declarativa:
Lógica y Restricciones

Programación Lógica Pura

Mari Carmen Suárez de Figueroa Baonza
mcsuarez@fi.upm.es

Programas Lógicos Puros: Información General

- Son programas que sólo hacen uso de la **unificación**
- Son completamente lógicos: el conjunto de respuestas computadas es exactamente el conjunto de consecuencias lógicas
  - Respuestas computadas: todas las llamadas que se ejecutan con éxito
- Permiten la programación declarativa: declarar el problema (especificaciones como programas)
- Tienen una potencia de cálculo completa

Contenidos

- Programación de bases de datos
- Programación recursiva
  - Tipos
  - Aritmética
  - Manipulación de estructuras de datos

Programación de Bases de Datos (I)

- Una **base de datos lógica** es un conjunto de hechos y reglas
  - Es decir, un programa lógico
- Ejemplo:
  - father_of(john,peter) <- .
  - father_of(john,mary) <- .
  - father_of(peter,michael) <- .
  - mother_of(mary, david) <- .

Hechos

  - grandfather_of(L,M) <- father_of(L,N), father_of(N,M).
  - grandfather_of(X,Y) <- father_of(X,Z), mother_of(Z,Y).

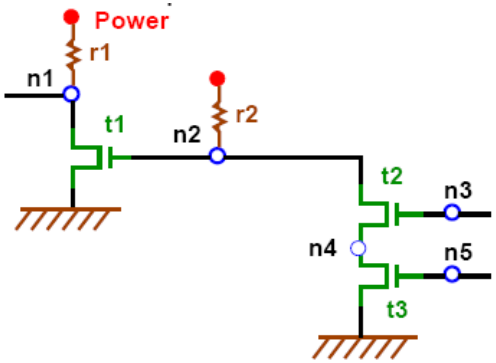
Reglas

Programación de Bases de Datos (II)

- Dada una base de datos lógica, un sistema de programación lógica puede responder **preguntas** (*queries*)
- Ejemplo:
  - <- father_of(john, peter). Respuesta: *Yes*
  - <- father_of(john, david). Respuesta: No
  - <- father_of(john, X). Respuesta: {X = peter}, {X = mary}
  - <- grandfather_of(X, michael). Respuesta: {X = john}
  - <- grandfather_of(X, Y). Respuesta: {X = john, Y = michael}, {X = john, Y = david}
  - <- grandfather_of(X, X). Respuesta: No
- Ejercicio:
  - ¿Qué reglas habría que escribir para el predicado "*grandmother_of(X, Y)*"?

Programación de Bases de Datos: Ejemplo (I)

- resistor(power,n1) <- .
- resistor(power,n2) <- .
- transistor(n2,ground,n1) <- .
- transistor(n3,n4,n2) <- .
- transistor(n5,ground,n4) <- .

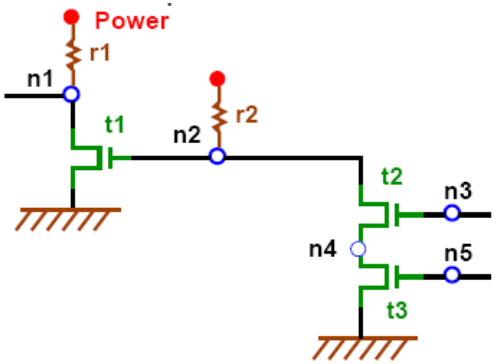


- Los inversores se construyen con un transistor y una resistencia
  - inverter(Input,Output) <-
    transistor(Input,ground,Output), resistor(power,Output).
- Una puerta *nand* se compone de dos transistores y una resistencia
  - nand_gate(Input1,Input2,Output) <-
    transistor(Input1,X,Output), transistor(Input2,ground,X),
    resistor(power,Output).

Programación de Bases de Datos: Ejemplo (II)

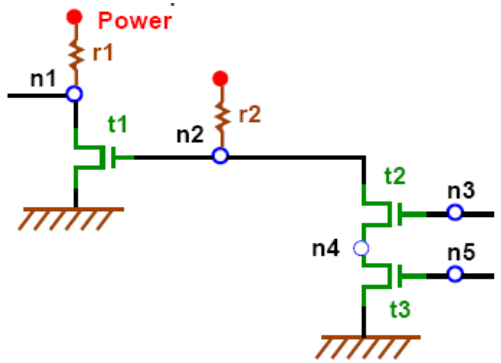
- Una puerta *and* consiste en una puerta *nand* y un inversor
  - and_gate(Input1,Input2,Output) <-
    nand_gate(Input1,Input2,X), inverter(X, Output).

- Consulta:
  - <- and_gate(In1,In2,Out).
- Respuesta:
  - {In1=n3, In2=n5, Out=n1}



Datos Estructurados y Abstracción de Datos (I)

- Ejemplo: Nueva versión para el ejemplo del circuito
  - resistor(r1,power,n1) <- .
  - resistor(r2,power,n2) <- .
  - transistor(t1,n2,ground,n1) <- .
  - transistor(t2,n3,n4,n2) <- .
  - transistor(t3,n5,ground,n4) <- .

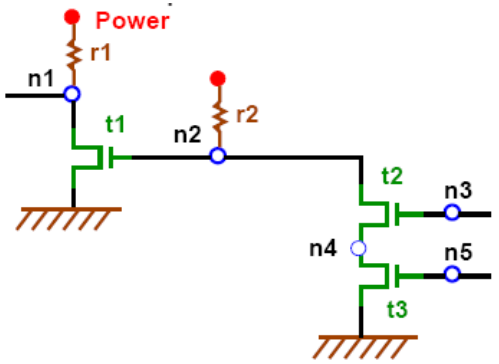


  - inverter(inv(T,R),Input,Output) <-
    transistor(T,Input,ground,Output), resistor(R,power,Output).
  - nand_gate(nand(T1,T2,R),Input1,Input2,Output) <-
    transistor(T1,Input1,X,Output), transistor(T2,Input2,ground,X),
    resistor(R,power,Output).

Datos Estructurados y Abstracción de Datos (II)

- Nueva versión para el ejemplo del circuito
  - and_gate(and(N,I),Input1,Input2,Output) <-

    nand_gate(N,Input1,Input2,X), inverter(I,X,Output).



Consulta:

<- and_gate(G,In1,In2,Out).

Respuesta:

{G=and(nand(t2,t3,r2),inv(t1,r1)),In1=n3,In2=n5,Out=n1}

Programas Lógicos y Modelo Relacional de Bases de Datos (I)

Tradicional	Modelo Relacional (Codd)	
Fichero	Relación	Tabla
Registro	Tupla	Fila
Campo	Atributo	Columna

- Ejemplo:

Persona

Name	Age	Sex
Brown	20	M
Jones	21	F
Smith	36	M

Vive_En

Name	Town	Years
Brown	London	15
Brown	York	5
Jones	Paris	21
Smith	Brussels	15
Smith	Santander	5

- El orden de las filas es inmaterial
- Las filas duplicadas no están permitidas

Programas Lógicos y Modelo Relacional de Bases de Datos (II)

Base de Datos Relacional	Programación Lógica
Nombre de relación	Símbolo de predicado
Relación	Procedimiento que consiste en hechos cerrados (hechos sin variables)
Tupla	Hecho cerrado (*ground fact*)
Atributo	Argumento de predicado

- Ejemplo:
  - person(brown,20,male) <- .
  - person(jones,21,female) <- .
  - person(smith,36,male) <- .

Name	Age	Sex
Brown	20	M
Jones	21	F
Smith	36	M

  - lived_in(brown,london,15) <- .
  - lived_in(brown,york,5) <- .
  - lived_in(jones,paris,21) <- .
  - lived_in(smith,brussels,15) <- .
  - lived_in(smith,santander,5) <- .

Name	Town	Years
Brown	London	15
Brown	York	5
Jones	Paris	21
Smith	Brussels	15
Smith	Santander	5

Programas Lógicos y Modelo Relacional de Bases de Datos (III)

- Las operaciones del modelo relacional se implementan fácilmente como reglas
- **Unión**: R o S
  - r_union_s(X1, ... ,Xn) ← r(X1, ... ,Xn).
  - r_union_s(X1, ... ,Xn) ← s(X1, ... ,Xn).
- **Diferencia**: R – S
  - r_diff_s(X1, ... ,Xn) ← r(X1, ... ,Xn), not s(X1, ... ,Xn).
  - r_diff_s(X1, ... ,Xn) ← s(X1, ..., Xn), not r(X1, ... ,Xn).

Nota: La negación se explica más adelante

- **Producto cartesiano**: R x S
  - r_X_s(X1, ... ,Xm,Xm+1, ... ,Xm+n) ←
    r(X1, ... ,Xm),s(Xm+1, ..., Xm+n).

Programas Lógicos y Modelo Relacional de Bases de Datos (IV)

- **Proyección**: Equivale a seleccionar sólo ciertas columnas
  - r13(X1,X3) ← r(X1,X2,X3).
- **Selección**: Selecciona las tuplas que cumplen unas ciertas condiciones
  - r_selected(X1,X2,X3) ← r(X1,X2,X3), ≤(X2,X3).

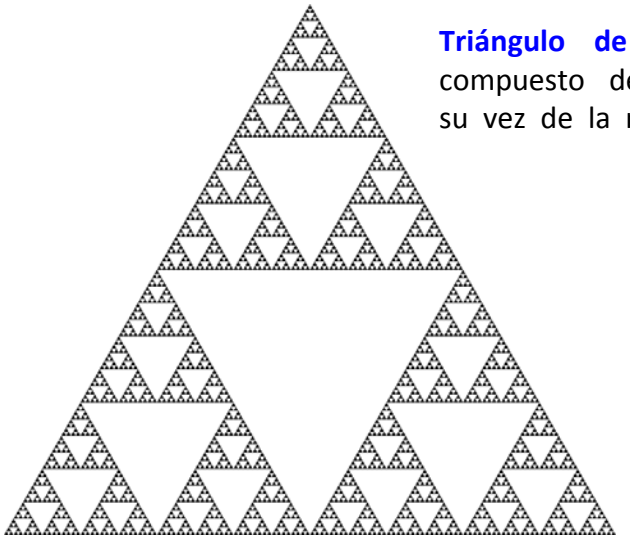
Nota: La definición de ≤/2 se explica más adelante

- **Operaciones derivadas**: Algunas se pueden expresar de forma más directa usando programación lógica
  - Intersección: R y S
    - r_meet_s(X1,. . .,Xn) ← r(X1,. . .,Xn), s(X1,. . .,Xn).
  - Join:
    - r_joinX2_s(X1,. . .,Xn) ← r(X1,X2,X3,. . .,Xn), s(X′1 ,X2,X′3,. . .,X′n).
- Existe el problema de los duplicados
  - Nota: Más adelante veremos “setof” (ISO Prolog)

Bases de Datos Deductivas

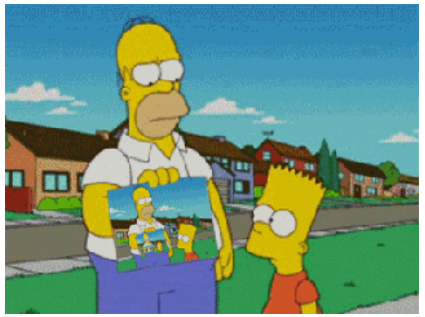
- Las **bases de datos deductivas** utilizan estas ideas para desarrollar la bases de datos basadas en lógica
  - A menudo se utilizan restricciones sintácticas (un subconjunto de programas definidos)
    - Ejemplo: En Datalog no hay funtores ni hay variables existenciales
  - Se usan variaciones de una estrategia de ejecución "*bottom-up*"
    - Por ejemplo, se usa el operador *Tp* para calcular el modelo y restringir la consulta

Recursividad (I)



Triángulo de Sierpinski: cada triángulo está compuesto de otros más pequeños, compuestos a su vez de la misma estructura recursiva

Matyroshkas (muñecas rusas): cada muñeca esconde en su interior otra muñeca, que esconde en su interior otra muñeca, hasta llegar a una muñeca que ya no esconde nada



Recursividad (II)

- Cualidad de recursivo
- **Recursivo**:
  - Sujeto a reglas o pautas recurrentes
  - Dicho especialmente de un proceso: Que se aplica de nuevo al resultado de haberlo aplicado previamente
  - Dicho de una unidad o una estructura: Que puede contener como constituyente otra del mismo tipo

- Técnica de programación que permite definir un objeto (problema, estructura de datos, etc.) en términos de si mismo
  - Una definición recursiva hace referencia a un caso similar más sencillo (no exactamente a sí mismo)

Recursividad (III)

- Ejemplos:
  - Factorial de n:
    - El factorial de 0 es, por definición, 1
    - Los factoriales de números mayores se calculan mediante la multiplicación de 1·2·..., incrementando el número de 1 en 1 hasta llegar al número para el que se está calculando el factorial
  - Término n-ésimo de la sucesión de Fibonacci
  - Potencia *n*-ésima de un número
  - Capital acumulado:
    - C0:= Capital inicial
    - C1:= (1+ % intereses anuales) · C0
    - Ci:= (1+ % intereses anuales) · Ci-1
  - Estructuras de datos recursivas: árboles, listas, grafos, conjuntos, etc.

Recursividad (IV)

- La **recursividad** es una técnica para resolver problemas que se basa en solucionar versiones más pequeñas de un problema, para obtener la solución general del problema planteado
  - Se suele utilizar para resolver problemas cuya solución se puede hallar resolviendo el mismo problema, pero para un caso de tamaño menor

Recursividad (V)

- La aplicación de esta técnica necesita la formulación de:
  - **Caso base**: del cual se conoce directamente la solución
    - Problema trivial que se puede resolver sin cálculo
    - Deben existir algún/algunos casos base que se puedan resolver sin recursión
  - **Caso general**: que debe poder resolverse en función del caso base y un caso general de menor tamaño (que progrese hacia una solución más sencilla (caso base))
    - Se da la solución del problema en función de un problema del mismo tipo más sencillo
    - Los casos que se resuelven de forma recursiva deben progresar siempre hacia algún caso base

Recursividad (VI)

- Propiedades de las definiciones recursivas:
  - No debe generar una secuencia infinita de llamadas a sí misma
    - Ha de existir al menos un caso base
  - Cada llamada recursiva se debe definir sobre un problema de menor complejidad
    - Sobre un problema más fácil de resolver

Recursividad: Ejemplo

- 4!=4*3!

Recursividad (VII)

- Preguntas a tener en cuenta
  - Sobre el caso base
    - ¿Existe una caso no recursivo (o caso base) para el problema?
    - ¿Es un caso correcto?
  - Sobre la reducción del tamaño del problema inicial
    - ¿Cada llamada recursiva se refiere a un caso más pequeño del problema original?
  - Sobre el caso general (o caso recursivo)
    - ¿Es correcta la solución en aquellos casos no base?

Recursividad (VIII)

- Usando analogías

Muñecas Rusas	Solución Recursiva
Una muñeca puede abrirse para ver su interior	Un problema puede **descomponerse** en subproblemas
Si se abre una muñeca grande, se encuentran muñecas más pequeñas en su interior	Si descomponemos un problema, obtenemos subproblemas con la **misma estructura** que el problema inicial pero que tratan con datos más pequeños
La muñeca más pequeña ya no contiene otras muñecas	Existen **casos simples** cuya solución no requiere descomposición
Sólo hay dos tipos de muñecas: (1) las que contienen otras en su interior y (2) las más pequeñas que no contienen nada	Todas las posibilidades de solución se cubren con los **casos simples** y con los **casos recursivos**

Recursividad (IX)

- **Pensando de forma recursiva**
  - Primero pensamos donde no se aplica una solución recursiva y si podemos utilizar esto como un caso base
    - Se definen los casos base: escenarios en los que la aplicación de una recursión no tiene sentido
    - Ejemplos:
      - Listas vacías
      - Nodos que no tienen hijos (árboles)
  - Segundo pensamos en la descomposición de los datos y en que elementos se debe aplicar la definición recursiva (caso recursivo)
    - Se definen los casos recursivos (o generales)
      - Se utiliza un elemento y la definición recursiva aplicada al resto de elementos
    - Ejemplos:
      - Un sumatorio es la suma del primer elemento más la suma del resto de elementos
      - El tamaño de una lista es 1 más el tamaño del resto de la lista

Programación Recursiva: Ejemplo

- Antepasados
  - parent(X,Y) <- father(X,Y).
    parent(X,Y) <- mother(X,Y).
  - ancestor(X,Y) <- parent(X,Y).
    ancestor(X,Y) <- parent(X,Z), parent(Z,Y).
    ancestor(X,Y) <- parent(X,Z), parent(Z,W), parent(W,Y).
    ancestor(X,Y) <- parent(X,Z), parent(Z,W), parent(W,K), parent(K,Y).
    ...

Programación Recursiva: Ejemplo

- Antepasados
  - parent(X,Y) <- father(X,Y).
    parent(X,Y) <- mother(X,Y).
  - ancestor(X,Y) <- parent(X,Y).
    ancestor(X,Y) <- parent(X,Z), parent(Z,Y).
    ancestor(X,Y) <- parent(X,Z), parent(Z,W), parent(W,Y).
    ancestor(X,Y) <- parent(X,Z), parent(Z,W), parent(W,K), parent(K,Y).
    ...

  ¿Cómo definir 'antepasado' de manera recursiva?
  - ancestor(X,Y) <- parent(X,Y).
    ancestor(X,Y) <- parent(X,Z), ancestor(Z,Y).

Tipos

- **Tipo**: un conjunto de términos (posiblemente infinito)
- **Definición de Tipo**: un programa que define un tipo
- Ejemplo: Día de la semana
  - Conjunto de términos para representar Lunes, Martes, etc.
  - Definición de tipo:
    - es_DiaSemana('lunes') <- .
    - es_DiaSemana('martes') <- .
    - Etc.
- Ejemplo: Fecha (día de la semana y día del mes):
  - Conjunto de términos para representar "fecha(lunes,23)", "fecha(martes,24)", etc.
  - Definición de tipo:
    - es_Fecha(fecha(S,D)) <- es_DiaSemana(S), es_DiaMes(D).
    - es_DiaMes(1) <- . es_DiaMes(2) <- .
    - Etc.

Programación Recursiva: Tipos Recursivos (I)

- **Tipos recursivos**: definidos por programas lógicos recursivos
- Ejemplo: Los números naturales (el tipo de datos recursivo más simple)
  - Conjunto de términos para representar: 0, s (0), s (s (0)), etc.
  - Definición de Tipo:
    - nat (0) <- .

      nat (s (X)) <- nat (X).

Nota: Un predicado recursivo mínimo es aquel que contiene una cláusula unitaria y una cláusula recursiva (con un solo cuerpo literal)

- Se puede analizar la complejidad, para una determinada clase de consultas ("modo")
  - Ejemplo: Para el modo nat(ground), la complejidad es lineal en el tamaño del número

Programación Recursiva: Tipos Recursivos (II)

- Ejemplo: Los enteros
  - Conjunto de términos para representar: 0, s (0), -s (0), etc.
  - Definición de Tipo:
    - integer(X) <- nat(X).

      integer(-X) <- nat(X).

Programación Recursiva: Aritmética (I)

- Definición del **orden de los números naturales** (≤):
  - menor_o_igual(0,X) <- nat(X).
    menor_o_igual(s(X),s(Y)) <- menor_o_igual(X,Y).

  Múltiples usos:
  - menor_o_igual (s (0), s (s (0))), menor_o_igual (X, 0), etc.

  Múltiples soluciones:
  - menor_o_igual (X, s (0)), menor_o_igual (s (s (0)), Y), etc.
- Definición de la **suma de números naturales**:
  - suma(0, X, X) <- nat (X).
    suma (s (X), Y, s (Z)) <- suma (X, Y, Z).

  Múltiples usos:
  - suma (s (s (0)), s (0), Z), suma (s (s (0)), Y, s (0)), etc.

  Múltiples soluciones:
  - suma (X, Y, s (s (s (0)))), etc.

Programación Recursiva: Aritmética (II)

- Otra definición de **suma**:
  - suma(X,0,X) <- nat(X).

    suma(X,s(Y),s(Z)) <- suma(X,Y,Z).
- El significado de suma es el mismo si se combinan ambas definiciones
  - Sin embargo, no se recomienda: se tienen varios árboles de búsqueda para una misma consulta
    - No es eficiente, no es concisa
    - Se buscan siempre axiomatizaciones mínimas
- El arte de la programación lógica consiste en encontrar formulaciones compactas y (computacionalmente) eficientes

Programación Recursiva: Aritmética (III)

- Definición de **mod(X,Y,Z)**: Z es el resto de dividir X entre Y
  - (∃ Q t.q. X = Y*Q + Z y Z < Y)
  - mod(X,Y,Z) <- less(Z, Y), times(Y,Q,W), plus(W,Z,X).
  - less(0,s(X)) <- nat(X).

    less(s(X),s(Y)) <- less(X,Y).
- Otra posible definición:
  - mod(X, Y, X) <- less(X, Y).

    mod(X, Y, Z) <- plus(X1, Y, X), mod(X1, Y, Z).
- La segunda definición es mucho más eficiente que la primera
  - Si se compara el tamaño de los árboles de búsqueda

Programación Recursiva: Aritmética. Ejercicio

- Un **numeral** es una cadena de cifras usada para denotar un número
  - Los numerales "21", "2", "3", "4" y "500" representan en el sistema arábico los mismos números que los respectivos numerales "XXI", "II", "III", "IV" y "D" en el sistema romano
- Se pide un predicado que, utilizando la notación de Peano,
  - Devuelva el numeral en notación arábiga de cualquier dígito
    - Por ejemplo "4" es el numeral en notación arábiga del dígito representado por s(s(s(s(0))))
  - Devuelva el primer y segundo dígito en notación arábiga de cualquier decena
  - Devuelva el primer, segundo y tercer dígito en notación arábiga de cualquier centena

aritmetica_logica.pl

Programación Recursiva: Aritmética y Funciones

- La función Ackermann:
  - ackermann(0,N) = N+1
    ackermann(M,0) = ackermann(M-1,1)
    ackermann(M,N) = ackermann(M-1,ackermann(M,N-1))
- La función Ackermann usando la aritmética de Peano:
  - ackermann(0,N) = s(N)
    ackermann(s(M),0) = ackermann(M,s(0))
    ackermann(s(M),s(N)) = ackermann(M,ackermann(s(M),N))
- En programación lógica, se puede definir esta función como:
  - ackermann(0,N,s(N)) <- .
    ackermann(s(M),0,Val) <- ackermann(M,s(0),Val).
    ackermann(s(M),s(N),Val) <- ackermann(s(M),N,Val1),
    ackermann(M,Val1,Val).

Programación Recursiva: Aritmética y Funciones

- En general, las funciones se pueden codificar como un predicado con un argumento más, que representa la salida
  - A menudo existe '*syntactic sugar*'
- Existe soporte sintáctico disponible
  - Véase, por ejemplo, el paquete de funciones Ciao

Programación Recursiva: Listas (I)

- Definición de tipo (sin usar '*syntactic sugar*'):
  - list([]) <- .

    list(.(X,Y)) <- list(Y).
- Definición de tipo (usando '*syntactic sugar*'):
  - list([]) <- .

    list([X|Y]) <- list(Y).
- Ejemplos:
  - [s(0),s(0),s(s(0)),s(s(s(0))),s(0)]
  - [ana, luis, pepe]
  - [capital(madrid,espagna), capital(paris, francia), capital(roma,italia)]

Programación Recursiva: Listas (II)

- Definición de **miembro de una lista**

  Supongamos que tenemos la lista de las personas que asistirán a una reunión y queremos averiguar si una determinada persona irá a ella

  La forma de programarlo usando programación lógica consiste en:

  - Analizar si dicha persona es la primera de la lista o
  - Analizar si está en el resto de la lista
    - Esto implica analizar de nuevo si es la primera del resto o está en el resto de dicho resto, y así sucesivamente
    - Si se llega al final de la lista, es decir, a la lista vacía, entonces la persona buscada no está en la lista de asistentes

Programación Recursiva: Listas (II)

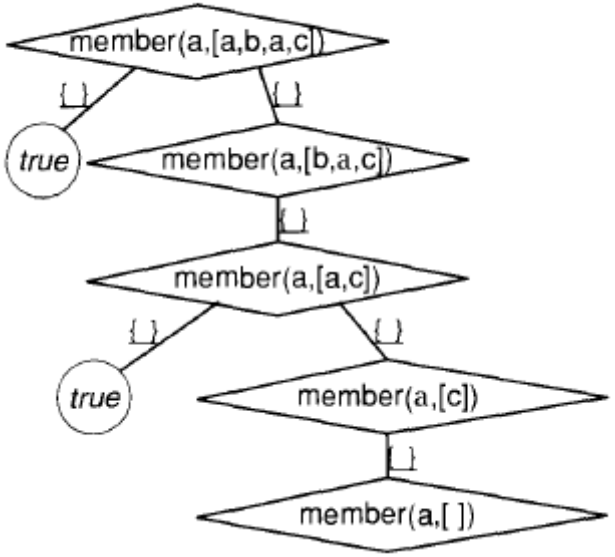
- Definición de **miembro de una lista** (miembro(X, L), significa X es miembro de la lista L)
  - miembro(X, [X|Y]) <-.

    miembro(X, [Y|Z]) <- miembro(X,Z).
    - La primera regla del predicado indica "X es miembro de la lista que tiene X como cabeza"
    - La segunda regla expresa "X es miembro de una lista si X es miembro del resto de dicha lista"
  - El predicado establece declarativamente "cómo" recorrer una lista, de principio a fin, para buscar un elemento
  - Una posible pregunta al programa es
    - ? miembro(pedro, [maria, pedro, juan, ana]).

Programación Recursiva: Listas (III)

- Definición de **miembro de una lista**
  - miembro(X, [X|Xs]).

    miembro(X, [Y|Ys]) :- miembro(X,Ys).
- Árbol de búsqueda para la consulta
  - ?- miembro(a, [a,b,a,c]).



- Nota: Consultas con variables funcionan correctamente

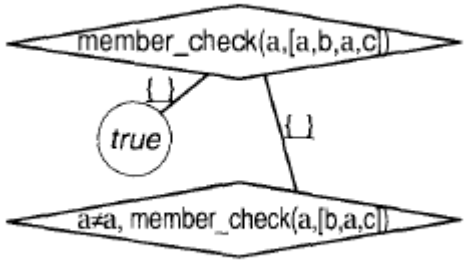
Programación Recursiva: Listas (IV)

- Otra definición de **miembro de una lista**
  - miembro2(X, [X|Xs]).

    miembro2(X, [Y|Ys]) :-

    X \= Y, % X e Y no son unificables

    miembro2(X,Ys).
- Árbol de búsqueda para la consulta
  - ?- miembro2(a, [a,b,a,c]).



- Nota: Consultas con variables no funcionan correctamente
  - Se deben usar únicamente términos cerrados (ground)

Programación Recursiva: Listas (V)

- Definición de **concatenación de listas**

  Dadas dos listas L1 y L2, obtener la lista L3 formada por la concatenación (o yuxtaposición) de los elementos de L1 con los de L2

  Ejemplo: Dadas L1 = [a, b, c] y L2 = [f, e], la concatenación de L1 con L2 es L3, donde L3 = [a, b, c, f, e]

- El predicado concatenar(L1, L2, L3) significa "L3 es la concatenación de la lista L1 con la lista L2"
  - concatenar([],L,L) <-.

    concatenar([X|Y],Z,[X|U]) <- concatenar(Y,Z,U).

Programación Recursiva: Listas (VI)

- Definición de **último elemento de una lista**

  Dada una lista L1, obtener el último elemento de dicha lista

  Ejemplo: Dada L1 = [a, b, c], U = c es el último elemento

- ultimo(X,[X]).

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

Programación Recursiva: Listas (VII)

- Definición de **inversa de una lista**

  Dada una lista L1, obtener su inversa

  Ejemplo: Dada L1 = [a, b, c], L2 = [c, b, a] es su inversa

- inversa([],[]).
  inversa([X|R],L):-
  inversa(R,L1),
  concatenar(L1,[X],L).

Listas: Ejemplo

- **Componentes de una bicicleta**: Supongamos una fábrica de bicicletas, donde se necesita tener un inventario de los componentes de bicicletas. Si queremos montar una bicicleta para un cliente, hemos de saber que componentes necesitamos del almacén

  Cada componente puede estar compuesto de subcomponentes

  - Por ejemplo, cada rueda puede tener radios, una llanta y un plato; un plato puede constar de un eje y de piñones

  Consideraremos una base de datos que nos permita hacer preguntas sobre componentes que se requieren para montar (una parte de) una bicicleta

Listas: Ejemplo

- Existen dos clases de componentes: básicos y compuestos
  - Los componentes básicos no se componen de otros menores, simplemente se combinan entre ellos para formar ensamblajes
  - Los componentes compuestos (ensamblajes) consisten en un conjunto de componentes básicos y/o ensamblajes (por ejemplo, la rueda que consta de radios, llanta y plato)
- Podemos representar
  - las piezas básicas como hechos
  - los ensamblajes pueden representarse también como hechos, mediante un predicado de dos argumentos
    - Por ejemplo, ensamblaje (bici, unir([rueda, rueda, cuadro])) indica que una bicicleta es un ensamblaje formado por la unión de dos ruedas y un cuadro

Listas: Ejemplo

- La base de datos de componentes básicos y ensamblajes necesaria sería la siguiente:
  - componenteBasico(llanta). componenteBasico(cuadro_trasero).
  - componenteBasico(piñones). componenteBasico(tuerca).
  - componenteBasico(radios). componenteBasico(manillar).
  - componenteBasico(tornillo). componenteBasico(horquilla).

  - ensamblaje(bici, unir([rueda, rueda, cuadro])).
  - ensamblaje(rueda, unir([radios, llanta, plato])).
  - ensamblaje(cuadro, unir([cuadro_trasero, cuadro_delantero])).
  - ensamblaje(cuadro_delantero, unir([horquilla, manillar])).
  - ensamblaje(plato, unir([piñones, eje])).
  - ensamblaje(eje, unir([tornillo, tuerca])).

Listas: Ejemplo

- Para escribir el programa que, dado un componente, devuelva la lista de componentes básicos que se necesitan para construirlo, hemos de tener en cuenta:
  - Si el componente que queremos montar es un componente básico, entonces la lista que contiene dicho componente es la respuesta
  - Si queremos montar un componente de ensamblaje, tenemos que aplicar el proceso de búsqueda de los componentes básicos para cada componente del ensamblaje
- Definimos un predicado **componentesDe(X, Y)**, donde X es el nombre de un componente e Y es la lista de componentes básicos que se necesitan para construir X
  - componentesDe(X, [X]) :- componenteBasico(X).
    componentesDe(X, P) :-
    ensamblaje(X, unir(Subcomponentes)),
    listacomponentesde(Subcomponentes, P).

Listas: Ejemplo

- listacomponentesde([], []).

  listacomponentesde([P|Resto], Total):-
  componentesDe(P,Componentescabeza),
  listacomponentesde(Resto,Componentesresto),
  concatenar(Componentescabeza,Componentesresto ,Total).

- El proceso que debe seguir **listacomponentesde/2** (en el caso recursivo) consiste en
  - obtener los componentes básicos de la cabeza de la lista (mediante componentesDe/2)
  - obtener los componentes básicos del resto de la lista (mediante listacomponentesde/2)
  - concatenar ambos resultados en la lista (Total) de componentes básicos buscados

Listas: Ejercicio (I)

- **Sumar elementos de una lista**: definir un predicado sumaElementosLista(L,N), que tenga éxito si N es la suma de los elementos de la lista L
- Ejemplo:
  - ?- sumaElementosLista([s(0),s(s(0)),s(s(s(0))),0,s(0)],N).

    N = s(s(s(s(s(s(0))))))
  - ?- sumaElementosLista([s(0),s(s(0)),s(s(s(0))),0],s(0)).

    no

Listas: Ejercicio (II)

- **Duplicar elementos de una lista**: definir un predicado duplicarLista(L1,L2), que tenga éxito si L2 contiene todos los elementos de L1 duplicados
- Ejemplo:
  - ?- duplicarLista([a,b,c],L).

    L = [a,a,b,b,c,c] ?
  - ?- duplicarLista([ana,pablo,juan],[ana,ana,pablo,juan,juan]).

    no

Listas: Ejercicio (III)

- **Producto Cartesiano**: Suponiendo una representación de conjuntos como listas, definir un predicado "cartesiano(L1,L2,L3)" que tenga éxito si L3 es el producto cartesiano de L1 y L2
- Ejemplo:
  - ?- cartesiano([a, b], [1, 2, 3], C).

    C = [(a, 1), (a, 2), (a, 3), (b, 1), (b, 2), (b, 3)]

Programación Recursiva: Árboles Binarios (I)

- Los **árboles binarios** se representan mediante
  - Un funtor ternario "*tree(Elemento, Izquierda, Derecha)*"
  - El árbol vacío se representa por "*void*"
- Definición de tipo:
  - binary_tree(void) <- .

    binary_tree(tree(Element,Left,Right)) <-

    binary_tree(Left),

    binary_tree(Right).
- Definición de miembro de árbol binario (*tree_member(Element,Tree)*):
  - tree_member(X,tree(X,Left,Right)) <- .

    tree_member(X,tree(Y,Left,Right)) <- tree_member(X,Left).

    tree_member(X,tree(Y,Left,Right)) <- tree_member(X,Right).

Programación Recursiva: Árboles Binarios (II)

- Definición de pre_order(Tree,Order): (nodo raíz, nodo izquierda, nodo derecha)
  - pre_order(void,[]) <- .
  - pre_order(tree(X,Left,Right),Order) <-

    pre_order(Left,OrderLeft),

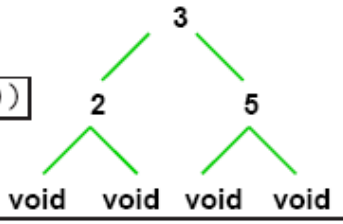
    pre_order(Right,OrderRight),

    append([X|OrderLeft],OrderRight,Order).
  - *Consulta*:
    - ?- pre_order(tree(a,tree(b,tree(d,void,void),tree(e,void,void)),tree(c,void,void)),L).
- Ejercicio:
  - Definir in_order(Tree,Order) y post_order(Tree,Order)

Ejemplo: Árbol Binario en Prolog y en Pascal

- In Prolog:

```
T = tree(3, tree(2,void,void), tree(5,void,void))
```



- In Pascal:

```
type tree = ^treerec;
        treerec = record
                data : integer;
                left : tree;
                right: tree;
        end;

var t : tree;
```

```
...
new(t);
new(t^left);
new(t^right);
t^left^left := nil;
t^left^right := nil;
t^right^left := nil;
t^right^right := nil;
t^data := 3;
t^left^data := 2;
t^right^data := 5;
...
```

Programación Recursiva: Matrices

- Representación de **matrices**:
  - La estructura de datos con la que representaremos las matrices será una lista formada por las filas de la matriz.
  - De esta forma, el término que corresponde a la matriz

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

es el siguiente: [[a,b],[c,d]]

Recursividad (I)

- Uno de los peligros que conlleva la recursividad, es la de realizar definiciones circulares o que el intérprete de Prolog no sea capaz de resolver
- El ejemplo más simple de un caso problemático sería:
  - antecesor_de(X, Y) :- antecesor_de(X, Y).
    - que se puede reducir a la cláusula: P => P
  - Esta cláusula es declarativamente correcta, pero el intérprete de Prolog no podrá resolverla nunca y se quedará atrapado en un bucle infinito

Recursividad (II)

- Más casos problemáticos: por ejemplo, se define la siguiente base de hechos sobre las amistades de una persona:
  - amigos(pedro, antonio).
  - amigos(pedro, flora).
  - ...
  - amigos(fernando, pedro).
- Si consideramos que la relación de amistad es conmutativa, entonces posiblemente nos interesaría definir la relación de amistad de la siguiente manera:
  - amigos(X, Y) :- amigos(Y, X). % recurrencia

Recursividad (III)

- Esta definición aparentemente lógica provoca un error
- La forma correcta de definir esta situación sería usando un nuevo predicado:
  - son_amigos(X, Y) :- amigos(X, Y).
  - son_amigos(X, Y) :- amigos(Y, X).

Recursividad (IV)

- En lógica matemática no se impone un orden especial de las cláusulas y de los términos que componen los programas
- Sin embargo, en Prolog es necesario cuidar el orden de las cláusulas dentro de un programa, debido a que el intérprete unifica la reglas en el orden secuencial en el que le han sido proporcionadas

Recursividad (V)

- Ejemplo: Suponiendo la definición del predicado antecesor_de que consta de dos cláusulas, una de las cuales posee dos metas (u objetivos), esto permite generar cuatro definiciones distintas de antecesor_de, permutando el orden de los términos y de las cláusulas
  - antecesor1(X, Y) :- padre_de(X, Y).

    antecesor1(X, Y) :- padre_de(X, Z), antecesor1(Z, Y).
  - antecesor2(X, Y) :- padre_de(X, Z), antecesor2(Z, Y).

    antecesor2(X, Y) :- padre_de(X, Y).
  - antecesor3(X, Y) :- padre_de(X, Y).

    antecesor3(X, Y) :- antecesor3(Z, Y), padre_de(X, Z) .
  - antecesor4(X, Y) :- antecesor4(Z, Y), padre_de(X, Z) .

    antecesor4(X, Y) :- padre_de(X, Y).

Recursividad (VI)

- Con las definiciones anteriores se obtienen resultados distintos
  - antecesor1 es la más eficiente y funciona siempre
  - antecesor2 es menos eficiente
  - antecesor3 sólo funciona para algunos casos
  - antecesor4 no funciona nunca, siempre se queda atrapado en un bucle infinito
- Consultas:
  - ?- antecesor3(antonio, carlos).
    - Verifica que la relación es cierta
  - ?- antecesor3(carlos, maria).
    - La relación es falsa, pero el programa no puede comprobarlo, queda atrapado en un bucle infinito
  - ?- antecesor3(X, Y).
    - Es capaz de imprimir algunos resultados, pero cuando la primera regla que define antecesor3 falla, queda atrapado en un bucle infinito

Recursividad (VII)

- **Directivas para mejorar la eficiencia**: El siguiente conjunto de directivas se pueden utilizar para lograr que la ejecución de nuestro programa Prolog sea lo más eficiente posible
  - Primero los objetivos más sencillos
    - Ordenación de cláusulas:
      - 1º las más específicas
      - 2º las más generales (con recursividad)
      - Ejemplo: antecesor(X, Y) :- padre_de(X,Y).
antecesor(X, Y) :- padre_de(X,Z), antecesor(Z,Y).
    - Ordenación de términos dentro de una cláusula:
      - 1º los términos más específicos
      - 2º los términos más generales (recursivos)
      - Ejemplo: antecesor(X, Y) :- padre_de(X,Z), antecesor(Z,Y).
  - Acotar el espacio de búsqueda
    - Buscar primero en lo menos frecuente

Polimorfismo

- En un programa lógico se pueden usar **dos definiciones de un mismo predicado** de manera simultánea
- Ejemplo:
  - lt_member(X,[X|Y]) <- list(Y).
    lt_member(X,[_|T]) <- lt_member(X,T).

    Las listas sólo unifican con estas dos cláusulas
  - lt_member(X,tree(X,L,R)) <- binary_tree(L), binary_tree(R).
    lt_member(X,tree(Y,L,R)) <- binary_tree(R), lt_member(X,L).
    lt_member(X,tree(Y,L,R)) <- binary_tree(L), lt_member(X,R).

    Los árboles sólo unifican con estas tres cláusulas
  - <- lt_member(X,[b,a,c]).
    X = b ; X = a ; X = c
  - <- lt_member(X,tree(b,tree(a,void,void),tree(c,void,void))).
    X = b ; X = a ; X = c

Programación Recursiva: Manipulación de Expresiones Simbólicas

- Reconocimiento de **polinomios** en algún término X:
  - X es un polinomio en X
    - polynomial(X,X) <- .
  - Una constante es un polinomio en X
    - polynomial(Term,X) <- pconstant(Term).
  - Sumas, diferencias y productos de polinomios en X son polinomios
    - polynomial(Term1+Term2,X) <- polynomial(Term1,X), polynomial(Term2,X).
    - polynomial(Term1-Term2,X) <- polynomial(Term1,X), polynomial(Term2,X).
    - polynomial(Term1*Term2,X) <- polynomial(Term1,X), polynomial(Term2,X).
  - La división de un polinomio por una constante es un polinomio
    - polynomial(Term1/Term2,X) <- polynomial(Term1,X), pconstant(Term2).
  - La potencia de un número natural es un polinomio
    - polynomial(Term1^N,X) <- polynomial(Term1,X), nat(N).

Programación Recursiva: Grafos (I)

- Normalmente se hace uso de otra estructura de datos, por ejemplo, las listas
  - **Grafos** como listas de arcos (aristas)
- Otra solución es hacer uso de la base de datos del programa lógico
  - Declarar el grafo usando hechos en el programa
    - arista (a, b) <-.
    - arista (b, c) <-.
    - arista (c, a) <-.
    - arista (d, a) <-.

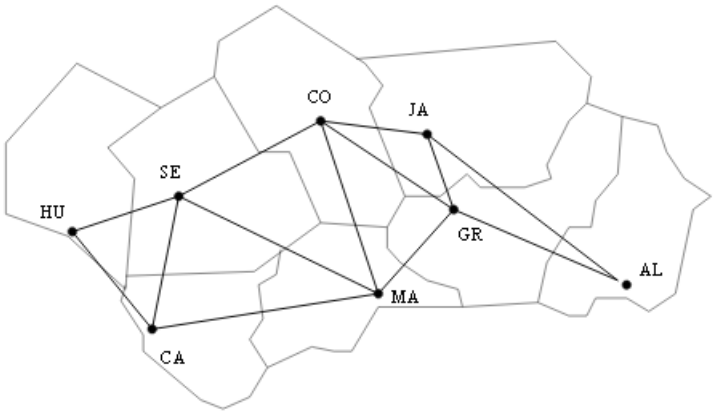
Programación Recursiva: Grafos (II)

- **Camino en un grafo**: camino(X, Y) si y sólo si existe un camino en el grafo para ir del nodo X al nodo Y
  - camino (A, B) <- arista (A, B).

    camino (A, B) <- arista (A, X), camino (X, B).
- **Circuito en un grafo** (un camino cerrado): circuito/0 si y sólo si existe un camino en el grafo de un nodo a sí mismo
  - circuito <- camino(A, A).

Programación Recursiva: Grafos. Ejemplo

- **Grafo de Andalucía**:
- arcos(L) se verifica si L es la lista de arcos del grafo
  - arcos([huelva-sevilla, huelva-cadiz, cadiz-sevilla, cadiz-malaga, malaga-granada, sevilla-malaga, sevilla-cordoba, cordoba-malaga, cordoba-granada, cordoba-jaen, jaen-granada, jaen-almeria, granada-almeria]).
- adyacente(X,Y) se verifica si X e Y son adyacentes
  - adyacente(X,Y) :-
    arcos(L), miembro(X-Y,L).
    adyacente(X,Y) :-
    arcos(L), miembro(Y-X,L).



Programación Recursiva: Grafos. Ejercicios

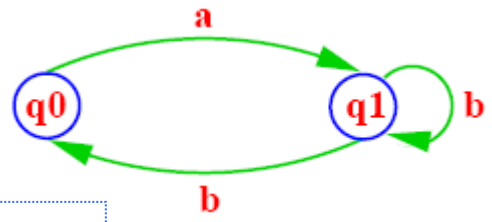
- Definir los siguientes predicados sobre grafos
  - camino1/3: predicado que dados dos nodos (X, Y) proporciona el camino (C) desde el nodo X al nodo Y
  - circuito1/1: predicado proporciona el circuito

Programación Recursiva: Grafos. Ejercicios

- Proponer una solución para manejar diferentes grafos en tu representación
  - arista, camino, circuito
- Proponer una representación adecuada para grafos como estructuras de datos
  - Definir los predicados anteriores para dicha representación
  - Considerar grafos dirigidos versus grafos no dirigidos

Programación Recursiva: Autómatas (Grafos) (I)

- **Reconocimiento de secuencias** de caracteres aceptadas por el siguiente autómata finito no determinista (NDFA)
  - donde q0 es tanto el estado inicial como el final



  - initial(q0) <-.
  - final(q0) <- .

Estados

initial(X) se verifica si X es el estado inicial
final(X) se verifica si X es el estado final

```
delta(q0,a,q1) <- .
delta(q1,b,q0) <- .
delta(q1,b,q1) <- .
```

Transiciones

delta/3 se verifica si se puede pasar del estado E1 al E2 usando la constante X

  - accept(S) <- initial(Q), accept_from(S,Q).

accept(S) se verifica si el autómata acepta la cadena S

  - accept_from([],Q) <- final(Q).

    accept_from([X|Xs],Q) <- delta(Q,X,NewQ), accept_from(Xs,NewQ).

Programación Recursiva: Autómatas (Grafos) (II)

- Autómata finito no determinista (basado en pila) (NDSFA):
  - accept(S) <- initial(Q), accept_from(S,Q,[]).
  - accept_from([],Q,[]) <- final(Q).

    accept_from([X|Xs],Q,S) <- delta(Q,X,S,NewQ,NewS),

    accept_from(Xs,NewQ,NewS).
  - initial(q0) <- .
  - final(q1) <- .
  - delta(q0,X,Xs,q0,[X|Xs]) <- .

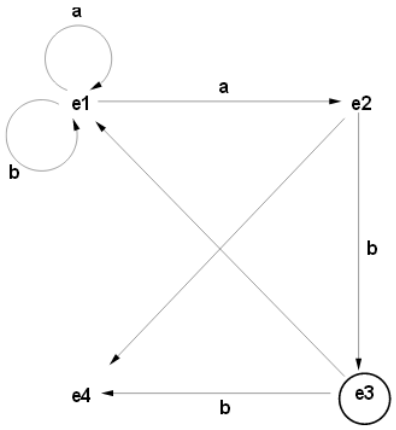
    delta(q0,X,Xs,q1,[X|Xs]) <- .

    delta(q0,X,Xs,q1,Xs) <- .

    delta(q1,X,[X|Xs],q1,Xs) <- .
- ¿Qué secuencia reconoce este autómata?

Programación Recursiva: Autómatas (Grafos) (III)

- Autómata no determinista (con estado final e3)
  - final(E) se verifica si E es el estado final
    - final(e3).
  - transicion(E1,X,E2) se verifica si se puede pasar del estado E1 al estado E2 usando la letra X
    - transicion(e1,a,e1).
    - transicion(e1,a,e2).
    - transicion(e1,b,e1).
    - transicion(e2,b,e3).
    - transicion(e3,b,e4).



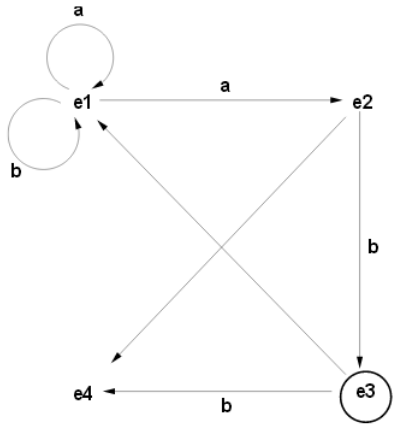
  - nulo(E1,E2) se verifica si se puede pasar del estado E1 al estado E2 mediante un movimiento nulo
    - nulo(e2,e4).
    - nulo(e3,e1).

Programación Recursiva: Autómatas (Grafos) (IV)

- Autómata no determinista (con estado final e3)
  - acepta(E,L) se verifica si el autómata, a partir del estado E acepta la lista L
  - Ejemplo:
    - acepta(e1,[a,a,a,b]) => Si
    - acepta(e2,[a,a,a,b]) => No
  - acepta(E,[]) :- final(E).

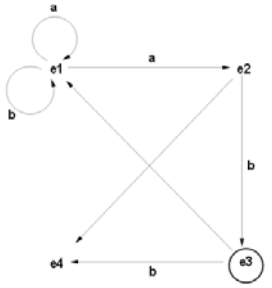
    acepta(E,[X|L]) :- transicion(E,X,E1), acepta(E1,L).

    acepta(E,L) :- nulo(E,E1), acepta(E1,L).



Programación Recursiva: Autómatas (Grafos) (V)

- Autómata no determinista (con estado final e3)



  - Determinar si el autómata acepta la lista [a,a,a,b]
    - ?- acepta(e1,[a,a,a,b]).

      Yes
  - Determinar los estados a partir de los cuales el autómata acepta la lista [a,b]
    - ?- acepta(E,[a,b]).

      E=e1 ;

      E=e3 ;

      no
  - Determinar las palabras de longitud 3 aceptadas por el autómata a partir del estado e1
    - ?- acepta(e1,[X,Y,Z]).

      X = a Y = a Z = b ;

      X = b Y = a Z = b ;

      no

Ejemplo: Las Torres de Hanoi (I)

- La **leyenda de las Torres de Hanoi**:

  "En la ciudad de Beranés existe un templo en el que el Dios Brahma, al crear el mundo, puso verticalmente tres agujas de diamante, colocando en una de ellas 64 anillos de oro: el más grande, en la parte inferior, y los demás por orden de tamaño uno encima de otro. Los sacerdotes del templo, debían, trabajando día y noche sin descanso, trasladar todos los anillos de la primera aguja a la tercera, utilizando la segunda como auxiliar, cuidando que nunca quedara un disco mayor sobre uno de menor tamaño. Los monjes se encuentran enteramente dedicados a esta tarea encomendada por su Dios, y continuan de generación en generación, reclutando cada vez a nuevos monjes que continuen la labor de los anteriores"

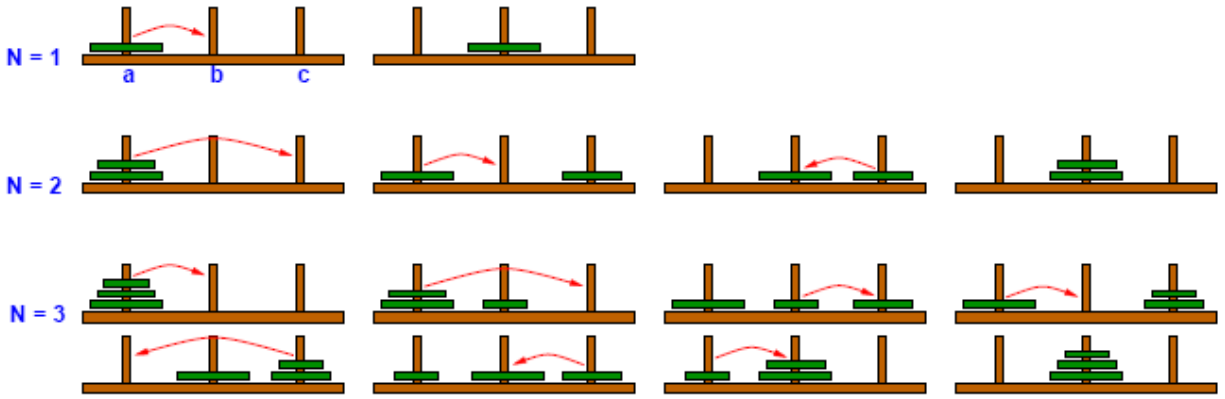
Ejemplo: Las Torres de Hanoi (II)

- Se pide un programa Prolog que resuelva el problema de las torres de Hanoi
  - Sobre una superficie de diamante, están colocadas tres agujas, en una de ellas, se encuentran 64 discos de oro de diferentes tamaños, cada uno de ellos de menor tamaño que el que le precede
  - La tarea encomendada por los dioses es la siguiente: se deben pasar todos los discos de la aguja número 1 a la aguja número 3 utilizando la aguja número 2 como auxiliar de tal manera que nunca quede un disco más grande sobre uno más pequeño

Ejemplo: Las Torres de Hanoi (III)

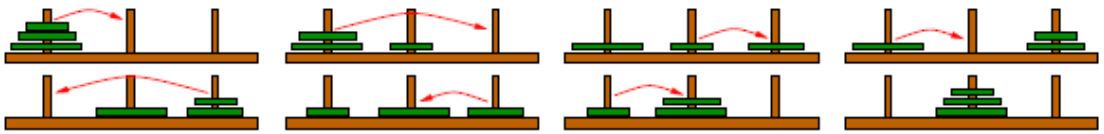
- **Objetivo**:
  - Mover N discos del palo a al palo b, usando como ayuda el palo c

- **Reglas**:
  - Mover un disco cada vez
  - Un disco mayor no puede estar nunca encima de un disco más pequeño



Ejemplo: Las Torres de Hanoi (IV)

- Predicado principal hanoi_moves(N,Moves)
  - *N* es el número de discos
  - *Moves* es la lista de movimientos
- Cada movimiento *move(A, B)* representa que el disco situado en la cima de A se debe mover a B
  - Ejemplo:



  - Esta situación se representa mediante:

    hanoi_moves (s(s(s(0))), [move(a,b), move(a,c), move(b,c), move(a,b), move(c,a), move(c,b), move(a,b)]).

Ejemplo: Las Torres de Hanoi (V)

- La regla general a utilizar es:

- Representamos esta regla en el predicado hanoi(N,Orig,Dest,Help,Moves) donde
  - Moves contiene los movimientos necesarios para mover una torre de N discos de un palo Orig a un palo Dest, usando como ayuda el palo Help
  - hanoi(s(0),Orig,Dest,_Help,[move(Orig, Dest)]) <- .

    hanoi(s(N),Orig,Dest,Help,Moves) <-

    hanoi(N,Orig,Help,Dest,Moves1),

    hanoi(N,Help,Dest,Orig,Moves2),

    append(Moves1,[move(Orig, Dest)|Moves2],Moves).

Ejemplo: Las Torres de Hanoi (VI)

- Por último llamamos a este predicado (para resolver el problema):
  - hanoi_moves(N,Moves) <- hanoi(N,a,b,c,Moves).

Ejercicio

- Existen 5 casas de diferentes colores. En cada una de las casas vive una persona con una nacionalidad diferente. Los 5 dueños beben una determinada bebida, fuman una determinada marca de cigarrillos y tienen una determinada mascota. Ningún dueño tiene la misma mascota, fuma la misma marca de cigarrillo o bebe la misma bebida. La pregunta es: ¿quién tiene el pez?
- **Claves:**
  - El británico vive en la casa roja
  - El sueco tiene como mascota un perro
  - El danés toma te
  - La casa verde está a la izquierda de la casa blanca
  - El dueño de la casa verde toma café
  - La persona que fuma Pall Mall tiene un pájaro
  - El dueño de la casa amarilla fuma Dunhill
  - El que vive en la casa del centro toma leche
  - El noruego vive en la primera casa
  - La persona que fuma Blends vive junto a la que tiene un gato
  - La persona que tiene un caballo vive junto a la que fuma Dunhill
  - El que fuma Bluemaster bebe cerveza
  - El alemán fuma Prince
  - El noruego vive junto a la casa azul
  - El que fuma Blends tiene un vecino que toma agua

Resumen

- Los programas lógicos puros permiten la programación declarativa pura
- Los programas lógicos puros tiene un elevado poder computacional

Programación Declarativa:
Lógica y Restricciones

Programación Lógica Pura

Mari Carmen Suárez de Figueroa Baonza
mcsuarez@fi.upm.es