Programación Declarativa:
Lógica y Restricciones

Conceptos Básicos de la Programación en Prolog

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

Contenidos

- Unificación
- Estructuras de datos
- Recursividad, *backtracking* y búsqueda
- Control de ejecución

Unificación (I)

- La **unificación** es el mecanismo que se encarga de resolver las igualdades lógicas y de dar valor a las variables lógicas
  - En la unificación no se evalúan expresiones
  - Para evaluar expresiones existe un operador especial "*is*"
    - Antes de realizar la unificación evalúa la parte derecha como si se tratase de una expresión aritmética

Unificación (II): Reglas Generales

- Si T1 y T2 son constantes, entonces T1 y T2 unifican si son idénticas
- Si T1 y T2 son variables, entonces T1 y T2 unifican siempre
- Si T1 es una variable y T2 es cualquier tipo de término, entonces T1 y T2 unifican y T1 se instancia con T2
- Si T1 y T2 son términos complejos, unifican si:
  - Tienen el mismo functor y aridad
  - Todos los argumentos unifican

Unificación (III). Ejemplos

- *¿pepe(A,rojo)=jose(B,rojo)?*
  - *pepe!=jose → No unifica*
- *¿pepe(A,rojo)=pepe(Z,rojo)?*
  - *pepe=pepe, A=Z, rojo= rojo → Si unifican*
- *¿X=a?* x is a
- *¿X=Y?* X=Y
- *¿f(a)=f(X)?* a = X
- *¿f(X,a)=f(b,Y)?* X is b, a=Y
- *¿f(X,a,X)=f(b,Y,c)?* X is b, a=Y,X is c no unifican
- *¿X=f(X)?* no unifican

Unificación (IV): =/2

- = simboliza el predicado *"unifica"*:
  - El predicado =/2 está *predefinido* en ISO-Prolog, no es necesario programar la unificación, que es una característica básica del demostrador automático
  - Sus dos argumentos son las dos expresiones a unificar
  - Este predicado es *verdadero* si sus dos argumentos unifican
    - a = a ; f(a,b) = f(a,b) ; X = a ; f(a,b) = X ; X = Y ; f(a,X) = f(a,b) ; X = f(Y)
  - Y es *falso* si sus dos argumentos no unifican
    - a = b ; f(a,b) = f(b,a) ; X = f(X) ; f(X,X) = f(g(Y),Y)

Unificación Implícita y Variable Anónima

- La unificación de una variable puede realizarse:
  - Explícitamente, empleando =/2 como un objetivo más
  - Implícitamente, dado que Resolución = Corte + Unificación

```
% "Juan es amigo de cualquiera que sea rico"
% (x)(rico(x) -> amigo(juan,x))
amigo(juan,X):- rico(X).
```

    - Es necesario unificar el segundo argumento de amigo/2, para que su valor se emplee en la prueba de rico/1

- La variable anónima:
  - Sintácticamente: '_' ; '_Anónima'; '_X'
  - Semánticamente: no se unifica, no toma nunca valor

```
% "Pepe es amigo de todo el mundo"
% (x) amigo(pepe,x)
amigo(pepe,_).
```

    - Es innecesario unificar el segundo argumento de amigo/2, cualquier valor es aceptable y no se emplea en ulteriores objetivos

Papel de la Unificación en Ejecución (I)

- La unificación se usa para acceder a los datos y para dar valor a las variables
  - Ejemplo: Considerando la consulta "?- animal(A), named(A,Name).", con

    animal(dog(barry)).

    named(dog(Name),Name).
    - La ejecución de "animal(A)" asigna un valor (ground) a A
    - La ejecución de "named(A,Name)" asigna un valor (ground) a Name, accediendo a los datos en el subcampo de la estructura dog/1

Papel de la Unificación en Ejecución (II)

- La unificación también se usa para pasar parámetros en llamadas a procedimientos y para devolver valores a la salida de dichos procedimientos
  - Ejemplo:
    - "?- animal(A), named(A,Name)." devuelve un valor a la salida de animal(A)
    - "?- named(dog(barry),Name)." pasa un valor en el primer argumento de la llamada named/2
    - "Name = barry" devuelve un valor en el segunda argumento a la salida de named/2

Modos de uso (I)

- En la definición de un procedimiento no hay parámetros predefinidos de "entrada" y/o "salida". El modo de uso de cada parámetro depende de la llamada o pregunta que se haga en cada momento al procedimiento
  - Ejemplo: Considerar las consultas “?- pet(spot).” vs. “?- pet(X).”
- Tras la llamada a un procedimiento, cualquier argumento puede ser cerrado (ground), libre, o parcialmente instanciado

Modos de uso (II)

- Por tanto, los procedimientos se pueden usar en diferentes modos (diferentes conjuntos de argumentos son de entrada o de salida en cada modo)
  - Ejemplo: Consideramos las siguientes consultas
    - ?- named(dog(barry),Name). **% entrada, salida**
    - ?- named(A,barry). **% salida, entrada**
    - ?- named(dog(barry),barry). **% entrada, entrada**
    - ?- named(A,Name). **% salida, salida**
- Un argumento podría ser incluso de entrada y salida
  - Ejemplo: Consideramos la consulta "?- struct(f(A,b))." con struct(f(a,B)).
- El hecho de que los predicados se puedan llamar de cualquier modo es una consecuencia directa de su naturaleza lógica

Acceso a los Datos (I)

- Acceso a subcampos de registros
  - Ejemplo:
    - day(date(Day,_Month,_Year),Day).
    - month(date(_Day,Month,_Year),Month).
    - year(date(_Day,_Month,Year),Year).
- Nombrando subcampos
  - Ejemplo:
    - date(day, date(Day,_Month,_Year),Day).
    - date(month,date(_Day,Month,_Year),Month).
    - date(year, date(_Day,_Month,Year),Year).
- Inicialización de variables
  - Ejemplo: ?- init(X), ...
    - init(date(9,6,2011)).
- Comparación de variables
  - Ejemplo: ?- init_1(X), init_2(Y), equal(X,Y).
    - equal(X,X).

    o simplemente: ?- init_1(X), init_2(X).

Datos Estructurados (I)

- Las estructuras de datos se crean usando términos completos
- Estructurar los datos es importante
  - course(complog,wed,18,30,20,30,'F.','Bueno',new,5102). **course/10**
  - ¿Cuándo es el curso de Lógica Computacional (complog)?

    ?- course(complog,Day,StartH,StartM,FinishH,FinishM,C,D,E,F).
- Versión estructurada
  - course(complog,Time,Lecturer, Location) :- **course/4**

    Time = t(wed,18:30,20:30),

    Lecturer = lect('F.','Bueno'),

    Location = loc(new,5102).

Nota: "X=Y" es equivalente a "=(X,Y)",donde el predicado =/2 se define como el hecho "=(X,X)." (plain unification)

- Version equivalente
  - course(complog, t(wed,18:30,20:30), lect('F.','Bueno'), loc(new,5102)).

Datos Estructurados (II)

- Suponiendo
  - course(complog,Time,Lecturer, Location) :-

    Time = t(wed,18:30,20:30),

    Lecturer = lect('F.','Bueno'),

    Location = loc(new,5102).
  - ¿Cuándo es el curso de Lógica Computacional (complog)?
    - ?- course(complog,Time, A, B).
    - Tiene como solución: {Time=t(wed,18:30,20:30), A=lect('F.','Bueno'), B=loc(new,5102)}
  - Si usamos la variable anónima
    - ?- course(complog,Time,_ ,_).
    - Tiene como solución: {Time=t(wed,18:30,20:30)}

Estructuras de Datos (I)

- Las estructuras en programas lógicos son básicamente registros
- Los arrays son básicamente registros con acceso por índice
  - Ejemplos:
    - index(1,array(X,_,_,...),X).
    - index(2,array(_,X,_,...),X).
    - index(3,array(_,_,X,...),X).
    - ...
  - Prolog proporciona un predicado predefinido para hacer esto

Estructuras de Datos (II)

- Las listas son básicamente registros con una estructura recursiva y un acceso secuencial a los elementos
  - Caso base: la lista vacía
  - Caso recursivo: un par (X,Y), donde un argumento es un elemento de la lista y el otro (normalmente el segundo – Y) es (recursivamente) una lista (el resto de la lista)

Listas

- Son estructuras binarias: el primer argumento es un elemento y el segundo es el resto de la lista
- Se necesita
  - Un símbolo constante: la lista vacía que se denota con la constante []
  - Un functor de aridad 2: tradicionalmente el punto "."
- *Syntactic sugar*: el término .(X,Y) se denota con [X|Y] (X es la cabeza, Y es la cola)

Formal object	*Cons pair syntax*	*Element syntax*
.(a,[ ])	[a|[ ]]	[a]
.(a,.(b,[ ]))	[a|[b|[ ]]]	[a,b]
.(a,.(b,.(c,[ ])))	[a|[b|[c|[ ]]]]	[a,b,c]
.(a,X)	[a|X]	[a|X]
.(a,.(b,X))	[a|[b|X]]	[a,b|X]

- Hay que tener en cuenta que
  - [a,b] y [a|X] unifica con {X = [b]}
  - [a] y [a|X] unifica con {X = [ ]}
  - [a] y [a,b|X] no unifica
  - [ ] y [X] no unifica

Strings (Listas de Códigos) (y Comentarios)

- *Strings* (de caracteres): se representan entre comillas ("...")
  - Si la doble comilla (") pertenece al *string*, entonces es necesario duplicarla
  - Ejemplos: "Prolog" "Esto es un ""string"""
- *Syntactic sugar*: En realidad un *string* equivale a la lista de los códigos ASCII de cada caracter
  - Ejemplo: "Prolog" ≡ [80,114,111,108,111,103]
- Comentarios
  - Si se usa “%”: El resto de la línea es un comentario
  - Si se usa “/* ... */”: Todo lo que hay entre medias es un comentario

Miembro de una Lista (I)

- member(X,Y) es cierto si y sólo si X es un miembro de la lista Y
- Por generalización

```
member(a,[a]).  member(b,[b]).  etc.          ⇒ member(X,[X]).
member(a,[a,c]).  member(b,[b,d]).  etc.      ⇒ member(X,[X,Y]).
member(a,[a,c,d]).  member(b,[b,d,l]).etc.    ⇒ member(X,[X,Y,Z]).

⇒ member(X,[X|Y]).


member(a,[c,a]), member(b,[d,b]).  etc.       ⇒ member(X,[Y,X]).
member(a,[c,d,a]).  member(b,[s,t,b]).  etc.  ⇒ member(X,[Y,Z,X]).

⇒ member(X,[Y|Z]) :- member(X,Z).
```

- Por tanto, se obtiene la definición
  - member(X,[X|]).
  - member(X,[|T]) :- member(X,T).

Miembro de una Lista (II)

- Definición
  - member(X,[X|]).
  - member(X,[|T]) :- member(X,T).
- Usos de member(X,Y)
  - Comprobar si un elemento está en una lista
    - ?- member(b,[a,b,c]).
  - Encontrar un elemento en una lista
    - ?- member(X,[a,b,c]).
  - Encontrar una lista que contiene un elemento
    - ?- member(a,Y).
- Ejercicios:
  - select(X,Ys,Zs) : X es un elemento de la lista Ys y Zs es la lista del resto de elementos de Ys
  - include(X,Ys,Zs) : Zs es la lista resultante de incluir el elemento X en la lista Ys (en cualquier lugar)

Concatenación de Listas (I)

- append(X,Y,Z) es cierto si y sólo si Z = X.Y (suponiendo "." un operador de concatenación de listas)
- Por generalización (recursión en el primer argumento):
  - Base case:

```
append([],[a],[a]).   append([],[a,b],[a,b]).   etc.
                                          ⇒ append([],Ys,Ys).
```

  - Rest of cases (first step):

```
append([a],[b],[a,b]).
append([a],[b,c],[a,b,c]).   etc.
                                    ⇒  append([X],Ys,[X|Ys]).
append([a,b],[c],[a,b,c]).
append([a,b],[c,d],[a,b,c,d]).   etc.
                                  ⇒ append([X,Z],Ys,[X,Z|Ys]).
```

- Esta solución todavía es infinita
  - Se necesita generalizar más

Concatenación de Listas (II)

- Segunda generalización

```
append([X],Ys,[X|Ys]).
append([X,Z],Ys,[X,Z|Ys]).
append([X,Z,W],Ys,[X,Z,W|Ys]).
                    ⇒ append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

- Por tanto, se obtiene la definición
  - append([],Ys,Ys).
  - append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).

- Usos de append(X,Y,Z)
  - Concatenar dos listas dadas: ?- append([a,b],[c],Z).
  - Encontrar diferencias entre listas: ?- append(X,[c],[a,b,c]).
  - Dividir una lista: ?- append(X,Y,[a,b,c]).

Recursión e Inducción

- La recursión es inducción lógica
- append(Xs,Ys,Zs) por inducción (en uno de sus argumentos, por ejemplo Xs)
  - Base: Xs=[]
    - Zs=Xs.Ys si Zs=Ys
  - Hipótesis: Xs=[X|Xs1] y se tiene que Zs1=Xs1.Ys1
  - Paso: Xs=[X|Xs1] y se tendría Zs=Xs.Ys si:
    - Ys=Ys1
    - Zs=[X|Zs1]
- Por tanto, se obtiene la definición
  - append([],Ys,Ys).
  - append([X|Xs1],Ys,[X|Zs1]) :- append(Xs1,Ys,Zs1).

Nota: La inducción es una forma de razonamiento que consiste en establecer una conclusión general a partir de la observación de hechos o casos particulares.

Inversa de una Lista (I)

- reverse(Xs,Ys) es cierto si y sólo si Ys es la lista que se obtiene de invertir los elementos de la lista Xs
- Pensando computacionalmente
  - Se necesita 'voltear' la lista Xs
  - Para cada elemento X de Xs, se debe colocar X al final del resto de Xs (lista ya invertida)
    - reverse([X|Xs],Ys) :-

      reverse(Xs,Zs),

      append(Zs,[X],Ys).
  - ¿Cómo paramos?
    - reverse([],[]).

Inversa de una Lista (II)

- Tal y como se ha definido, reverse(Xs,Ys) es muy ineficiente
- Otra posible definición

  Uso de parámetros de acumulación

  - reverse(Xs,Ys) :- reverse(Xs,[],Ys).

  - reverse([],Ys,Ys).

    reverse([X|Xs],Acc,Ys) :- reverse(Xs,[X|Acc],Ys).

Listas: Ejercicios (I)

- Definir prefijo(X,Y): la lista X es un prefijo de la lista Y
  - prefijo([a,b], [a,b,c,d]).
- Definir sufijo(X,Y): la lista X es un sufijo de la lista Y
  - sufijo([c,d], [a,b,c,d]).
- Definir sublista(X,Y): la lista X es una sublista de la lista Y. Proporcionar una solución recursiva y una no recursiva
  - sublista([b,c],[a,b,c,d]).
- Definir longitud(X,N): N es la longitud de la lista X (utilizando notación de Peano)

Listas: Ejercicios (II)

- Palíndromos
  - Definir el predicado palindromo/1 tal que:
    - palindromo(X) es cierto si la lista X es palíndromo, es decir, puede leerse de la misma manera al derecho y al revés
    - Ejemplos: palindromo([r,o,t,o,r]) es verdadero

      palindromo([r,o,t,a,r]) es falso

      palindromo([r,o,t|X]) es verdadero con {X = [o,r]}, o {X = [t,o,r]} o {X = [_A,t,o,r]} o ...
- Primero y Último
  - Definir el predicado primeroultimo/1 tal que:
    - primeroultimo(X) es cierto si el primer y ultimo elementos de la lista X son el mismo
    - Ejemplos: primeroultimo([a]) es verdadero

      primeroultimo([a,f,t]) es falso

      primeroultimo([X,f,t,a]) es verdadero con {X = a}

Estructuras de Datos Incompletas

- Listas Diferencia: Un par X-Y donde X es una lista abierta acabada en Y, que es una variable libre
  - Ejemplo: [1,2,3,4|X]-X
    - [1, 2, 3, 5, 8] y [5, 8]
    - [1, 2, 3, 6, 7, 8, 9] y [6, 7, 8, 9]
    - [1,2,3] y [ ]
    - En realidad, el par generalmente no es explícito, en su lugar hay un par de argumentos que actúa como una lista diferencia
  - Permiten mantener un puntero al final de la lista
  - Permiten la concatenación en tiempo constante
    - append_dl(X-Y,Y-Z,X-Z).
  - Permiten manipular listas de forma más eficiente definiendo "patrones de listas"
- Otros ejemplos: árboles diferencia, listas abiertas, árboles abiertos, diccionarios, colas, etc.

Ordenación de Listas

- El mejor enfoque consiste en aplicar la técnica de divide y vencerás
  - Dividir la lista en dos partes
  - Ordenar cada parte de manera recursiva
  - Unir ambas partes (ordenadas)
- Dos posibilidades
  - División difícil y unión fácil
    - Algoritmo quicksort
  - Unión difícil y división fácil
    - Algoritmo merge sort
    - Algoritmo insertion sort

Ordenación de Listas: Quicksort (I)

- La ordenación rápida (*quicksort*) es un algoritmo, basado en la técnica de divide y vencerás, que permite ordenar n elementos
  - Se elige un elemento (arbitrario) de la lista
  - Se divide la lista tomando los elementos menores que el elegido y los elementos mayores que el elegido
  - Se compone el resultado con los menores ordenados, el elemento seleccionado, y los mayores ordenados

Ordenación de Listas: Quicksort (II)

- *Quicksort* estándar (usando *append*): qsort(Xs,Ys) es cierto si y sólo si Ys es una permutación ordenada de la lista Xs
  - qsort([],[]).
    qsort([X|L],SL) :-
      partition(L,X,Left,Right),
      qsort(Left,SLeft),
      qsort(Right,SRight),
      append(SLeft,[X|SRight],SL).

  - partition([],_B,[],[]).
    partition([E|R],C,[E|Left1],Right):-
      E < C,
      partition(R,C,Left1,Right).
    partition([E|R],C,Left,[E|Right1]):-
      E >= C,
      partition(R,C,Left,Right1).

Nota: El segundo argumento (SL) es la versión ordenada de [X|L] si (a) *Left* y *Right* son el resultado de particionar L usando X; (b) *SLeft* y *SRight* son el resultado de ordenar de manera recursiva *Left* y *Right*; y (c) SL es el resultado de concatenar [X|*SRight*] y *SLeft*

Ordenación de Listas: Quicksort (II)

- Quicksort (usando listas diferencia)
  - Más eficiente que el predicado anterior
    - Todas las operaciones de concatenación (realizadas para combinar resultados parciales) se pueden llevar a cabo de manera implícita
  - La primera lista es una lista normal, la segunda se construye como una lista diferencia
  - dlqsort(L,SL) :- dlqsort_(L,SL\[]).
  - dlqsort_([],R\R).
  - dlqsort_([X|L],SL\R) :-

    partition(L,X,Left,Right),

    dlqsort_(Left,SL\[X|SR]),

    dlqsort_(Right,SR\R).
  - La partición que se realiza es la misma que en el caso anterior

Nota: El resultado de ordenar una lista vacía es una lista diferencia vacía

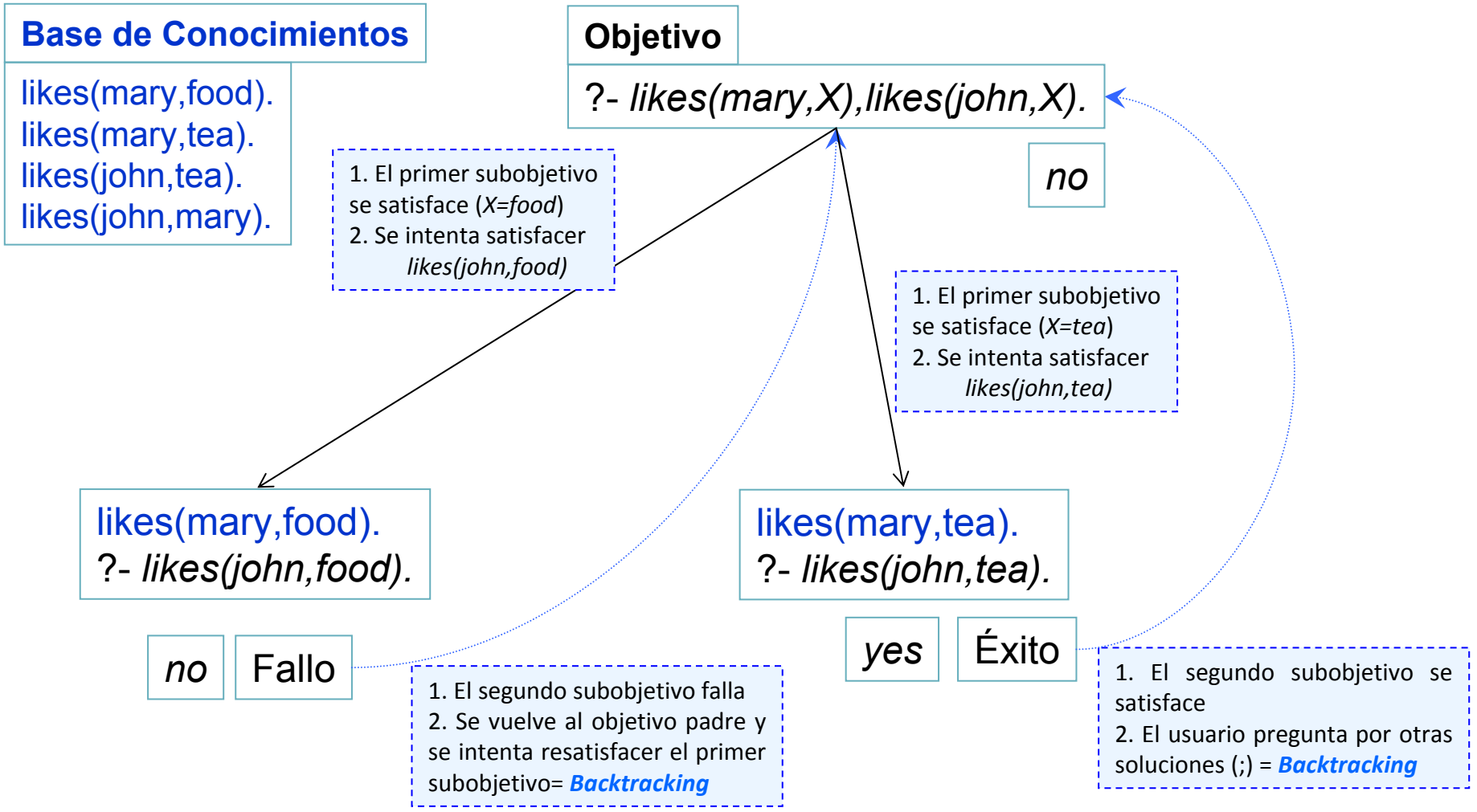
Backtracking (I)

- **Programa lógico**:
  - Base de conocimientos donde se expresan los hechos y las reglas de deducción de un dominio o problema
  - Motor de inferencia que aplica el algoritmo de resolución. Este algoritmo permite inferir nuevos datos relativos al mundo que estamos representando
    - Toma como **entrada** la base de conocimientos y el objetivo planteado
    - Ofrece como **salida** un resultado de verdadero o falso en función de si ha podido o no demostrar el objetivo según la base de conocimientos
    - Este algoritmo se basa en el uso de la técnica de ***backtracking***, de forma que la inferencia del objetivo planteado se realiza a base de prueba y error

Backtracking (II)

- Un hecho puede hacer que un objetivo se cumpla inmediatamente
- Una regla sólo puede reducir la tarea a la de satisfacer una conjunción de subobjetivos
- Si no se puede satisfacer un objetivo, se inicia un proceso de *backtracking*
  - Este proceso consiste en intentar satisfacer los objetivos buscando una forma alternativa de hacerlo
- El mecanismo de *backtracking* permite explorar los diferentes caminos de ejecución hasta que se encuentre una solución
  - *Backtracking* por fallo
  - *Backtracking* por acción del usuario

Backtracking (III). Ejemplo



Control de la Búsqueda

- Existen 3 formas principales de controlar la ejecución de un programa lógico
  - El orden de las cláusulas en un predicado
  - El orden de los literales en el cuerpo de una cláusula
  - Los operadores de poda (ej., '*cut*')
- El orden de las cláusulas en el programa y el orden de los literales en una cláusula son importantes
  - El orden afecta
    - al correcto funcionamiento del programa
    - al recorrido del árbol de llamadas, determinando, entre otras cosas, el orden en que Prolog devuelve las soluciones a una pregunta dada

Orden de las Cláusulas

- El orden de las cláusulas determina el orden en que se obtienen las soluciones
  - Varía la manera en que se recorren las ramas del árbol de búsqueda de soluciones
- Si el árbol de búsqueda tiene alguna rama infinita, el orden de las sentencias puede alterar la obtención de las soluciones, e incluso llegar a la no obtención de ninguna solución
- Regla heurística: es recomendable que los hechos aparezcan antes que las reglas del mismo predicado

Orden de las Cláusulas: Ejemplo (I)

- Dos versiones de miembro de una lista (miembro(X,L))
  - Ambas versiones tienen las mismas cláusulas pero escritas en distinto orden
- Versión 1:

```
1) miembro(X,[X|_]).
2) miembro(X,[_|Z]):- miembro(X,Z).
```

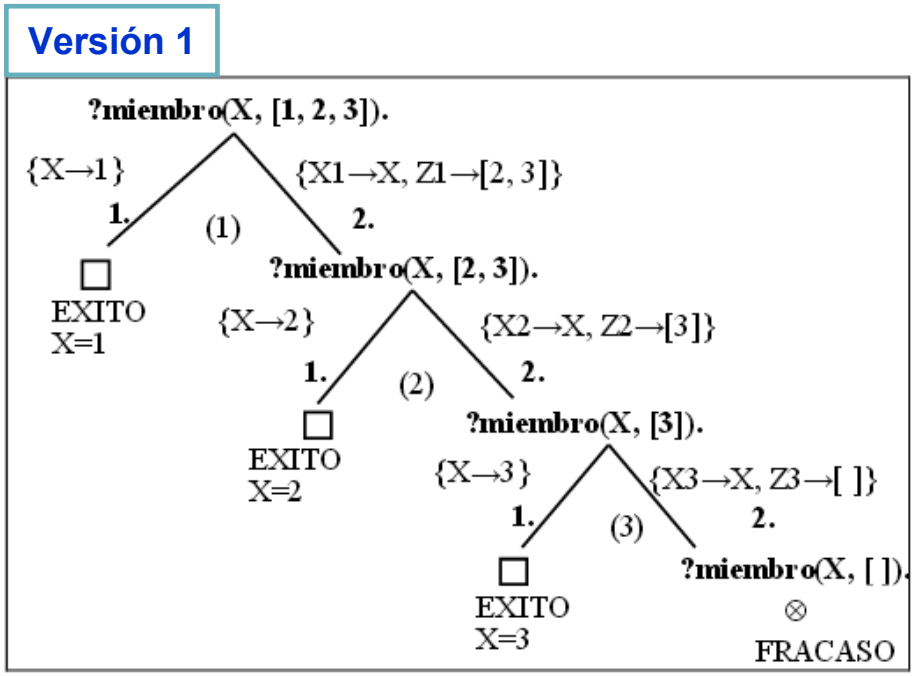
- Versión 2:

```
1) miembro(X,[_|Z]):- miembro(X,Z).
2) miembro(X,[X|_]).
```

Orden de las Cláusulas: Ejemplo (II)

- Realizamos la siguiente consulta
  - ?- miembro (X, [1,2,3]).

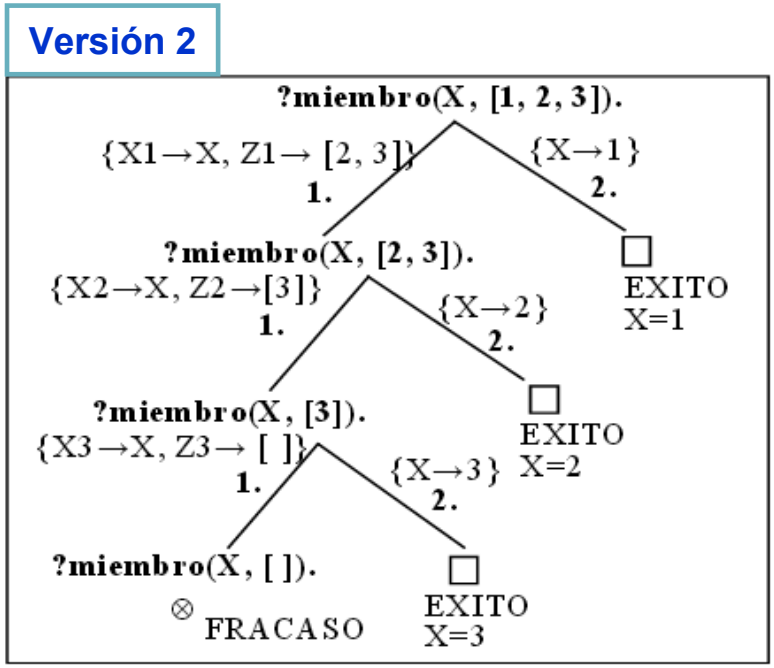
- Versión 1:
  1) miembro(X,[X|_]).
  2) miembro(X,[_|Z]):- miembro(X,Z).



Orden de las Cláusulas: Ejemplo (III)

- Realizamos la siguiente consulta
  - ?- miembro (X, [1,2,3]).

- Versión 2:
  1) miembro(X,[_|Z]):- miembro(X,Z).
  2) miembro(X,[X|_]).



Orden de los Literales (I)

- El orden de los literales dentro de una cláusula afecta al espacio de búsqueda y a la complejidad de los cómputos lógicos
- Distintas opciones en el orden de los literales pueden ser preferibles para distintos modos de uso
  - hijo(X, Y) :- hombre(X), padre(Y, X).
    - Para modo (in, out): Se comprueba primero que el X es hombre y después se busca a su padre Y
  - hijo(X, Y) :- padre(Y, X), hombre(X).
    - Para modo (out, in): Se buscan los hijos de Y y después se seleccionan si son hombres

Orden de los Literales (II)

- El orden de los literales en el cuerpo de una regla influye también en la terminación
  - inversa([ ], [ ]).
  - inversa([C|R], Z) :- inversa(R, Y), concatenar(Y, [C], Z).
    - Para preguntas en modo (in, out) termina
    - Para preguntas en modo (out, in) el árbol de búsqueda tiene una rama infinita, por lo que tras dar la respuesta correcta se queda en un bucle
  - ¿Qué sucede si intercambiamos los literales en la regla?

ordenLiterales.pl

Control con Poda: predicado *cut* (I)

- Prolog proporciona un predicado predefinido llamado cut (!/0) que influye en el comportamiento procedural de los programas
- Su principal función es reducir el espacio de búsqueda podando dinámicamente el árbol de búsqueda
- El corte puede usarse:
  - Para aumentar la eficiencia
    - Se eliminan puntos de *backtracking* que se sabe que no pueden producir ninguna solución
  - Para modificar el comportamiento del programa
    - Se eliminan puntos de *backtracking* que pueden producir soluciones válidas. Se implementa de este modo una forma débil de negación
    - Este tipo de corte debe utilizarse lo menos posible

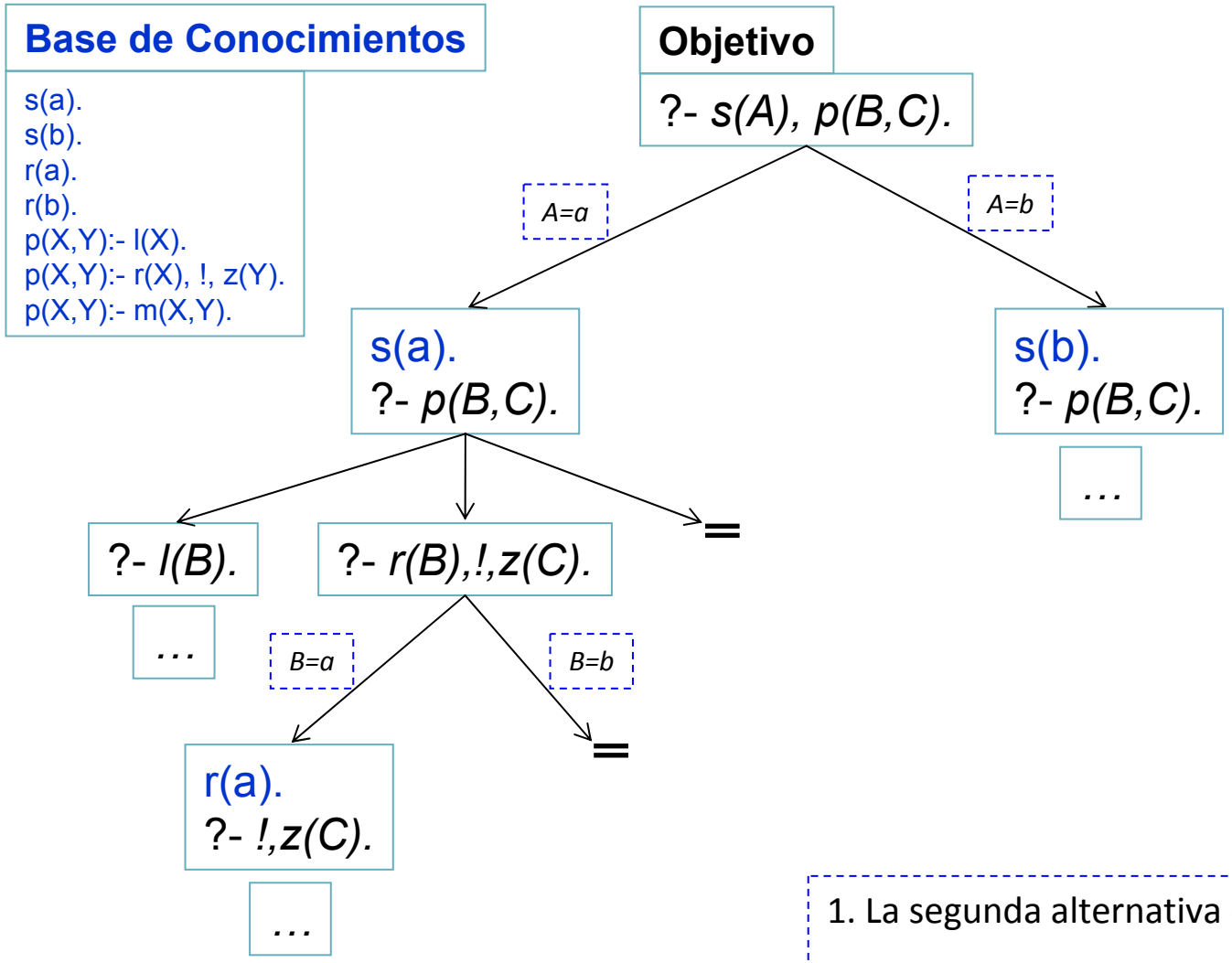
Control con Poda: predicado *cut* (II)

- El predicado cut como objetivo se satisface siempre y no puede re-satisfacerse
- Su uso permite podar ramas del árbol de búsqueda de soluciones
- Como consecuencia, un programa que use el corte será generalmente más rápido y ocupará menos espacio en memoria (no tiene que recordar los puntos de *backtracking* para una posible reevaluación)
  - El corte '!' limita el *backtracking*
  - Cuando se ejecuta el corte, se eliminan todos los puntos de *backtracking* anteriores dentro del predicado donde está definido (incluido el propio predicado)

Control con Poda: predicado *cut* (III)

- ¿Cómo funciona?
  - Un corte poda todas las alternativas correspondientes a cláusulas por debajo de él
  - Un corte poda todas las soluciones alternativas de la conjunción de objetivos que aparezcan a su izquierda en la cláusula
    - Es decir, una conjunción de objetivos seguida por un corte producirá como máximo una solución
  - Un corte no afecta a los objetivos que estén a su derecha en la cláusula
    - Estos objetivos pueden producir más de una solución, en caso de *backtracking*
    - Sin embargo, una vez que esta conjunción fracasa, la búsqueda continuará a partir de la última alternativa que había por encima de la elección de la sentencia que contiene el corte

Control con Poda: predicado *cut* (IV). Ejemplo



1. La segunda alternativa de r/1 (*r(b)*) no se considera
2. La tercera cláusula de p/2 no se considera

Control con Poda: predicado *cut* (V)

- Resumiendo, de forma general, el efecto de un corte en una regla C de la forma A :- B1, ..., Bk, !, Bk+2, ..., Bn, es el siguiente:
  - Si el objetivo actual G se unifica con A y los objetivos B1, ..., Bk se satisfacen, entonces el programa fija la elección de esta regla para deducir G
    - Cualquier otra regla alternativa (posterior) para A que pueda unificarse con G se ignora
  - Además, si los Bi con i>k fracasan, la vuelta atrás sólo puede hacerse hasta el corte
    - Las demás elecciones que quedaran para calcular los Bi con i≤ k se han cortado del árbol de búsqueda
  - Si el *backtracking* llega al corte entonces éste fracasa y la búsqueda continúa desde la última elección hecha antes de que G eligiera la regla C

Control con Poda: predicado *cut*. Ejemplo (I)

- mezcla (L1, L2, L), en modo (in,in,out), mezcla dos listas ordenadas de números L1 y L2 en la lista ordenada L

```
1) mezcla ([X| Xs], [Y| Ys], [X| Zs]) :- X<Y, mezcla (Xs, [Y| Ys], Zs).
2) mezcla ([X| Xs], [Y| Ys], [X,Y | Zs]) :- X=Y, mezcla (Xs, Ys, Zs).
3) mezcla ([X| Xs], [Y| Ys], [Y| Zs]) :- X>Y, mezcla ([X| Xs], Ys, Zs).
4) mezcla (Xs, [ ], Xs).
5) mezcla ([ ], Ys, Ys).
```

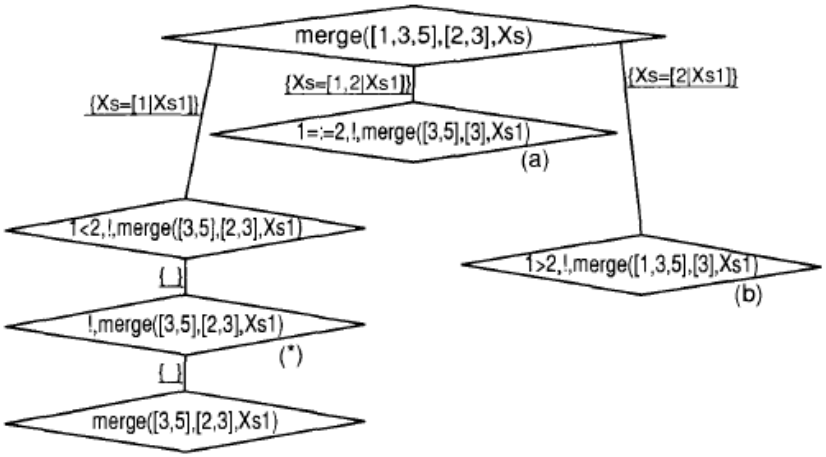
- La mezcla de dos listas ordenadas es una operación determinista
  - Sólo una de las cinco cláusulas se aplica para cada objetivo (no trivial) en una computación dada
  - En concreto, cuando comparamos dos números X e Y, sólo una de las tres comprobaciones X<Y, X=Y ó X>Y es cierta
  - Una vez una comprobación se satisface, no existe posibilidad de que alguna otra comprobación se satisfaga

Control con Poda: predicado *cut*. Ejemplo (II)

- El corte puede usarse para expresar la naturaleza mutuamente exclusiva de las comprobaciones de las tres primeras cláusulas
  1) mezcla ([X | Xs], [Y | Ys], [X | Zs]) :- X<Y, !,mezcla (Xs, [Y | Ys], Zs).
  2) mezcla ([X| Xs], [Y| Ys], [X, Y| Zs]) :- X=Y, !,mezcla (Xs, Ys, Zs).
  3) mezcla ([X| Xs], [Y| Ys], [Y| Zs]) :- X>Y, !, mezcla ([X| Xs], Ys, Zs).
- Por otra parte, los dos casos básicos del programa (cláusulas 4 y 5) son también deterministas
  4) mezcla (Xs, [ ], Xs):- !.
  5) mezcla ([ ], Ys, Ys).
  - La cláusula a utilizar se elige por unificación con la cabeza, por eso el corte aparece como el primer objetivo (en este caso el único) en el cuerpo de la cláusula 4
    - Dicho corte elimina la solución redundante (que se volvería a obtener con la cláusula 5) dado el objetivo "?-mezcla([ ], [ ], X)"

Control con poda: *cut*. Ejemplo (III)

- ?- mezcla([1,3,5], [2,3], X).
  - El objetivo principal se reduce primero (por la regla 1) al objetivo "?- 1<2, !, mezcla([3,5], [2,3], X1)".
  - El objetivo 1<2 se satisface, llegándose a un nodo del árbol cuyo primer objetivo es el corte
  - El efecto de ejecutar el corte (paso marcado con (*)) es podar las ramas (a) y (b)
  - El resto del desarrollo del árbol es similar, con dos cortes más podando ramas



Tipos de Corte

- Blancos: no descartan soluciones
  - No afectan ni a la completitud ni a la corrección
- Verdes: descartan soluciones correctas que no son necesarias
  - No afectan al sentido declarativo del programa
  - Sólo afectan a la eficiencia del programa
  - Afectan a la completitud pero no a la corrección
  - Podan ramas inútiles, redundantes o infinitas
- Rojos: modifican las soluciones del programa (al quitar el corte las soluciones son distintas)
  - Afectan a la semántica declarativa del programa
    - Hacen los programas menos declarativos y deben utilizarse con reservas
  - Modifican el significado lógico del programa
  - Al eliminar el corte se obtiene un programa incorrecto

Cortes Verdes

- No alteran el significado declarativo del programa
- En un programa semánticamente correcto se añade el corte para obtener un programa más eficiente
- Generalmente se usan para expresar determinismo
  - la parte del cuerpo que precede al corte (o a veces el patrón de la cabeza) comprueba un caso que excluye a todos los demás

Ejemplos:

- address(X,Add):- home_address(X,Add), !.

  address(X,Add):- business_address(X,Add).

- membercheck(X,[X|Xs]):- !.

  membercheck(X,[Y|Xs]):- membercheck(X,Xs).

Cortes Verdes. Ejemplo (I)

- Corte verde para evitar soluciones redundantes
  - es_padre(X):- padre(X,Y),!.
  - padre(antonio,juan).
  - padre(antonio,maria).
  - padre(antonio,jose).

ejemploCorte.pl

Cortes Verdes. Ejemplo (II)

- Corte verde para evitar búsquedas inútiles
  - El predicado ordenar(L,R) indica que R es el resultado de ordenar la lista L por medio de intercambios sucesivos

    Este predicado usará ordenada(L) que nos dice si la lista L está ordenada.

    1) ordenar (L, R) :- concatenar(P, [X,Y | S], L), X>Y, !, concatenar(P, [Y,X| S], NL), ordenar(NL, R).
    2) ordenar (L, L) :- ordenada(L).
  - Se sabe que sólo hay una lista ordenada
  - Por tanto, no tiene sentido buscar otras alternativas una vez se ha encontrado la lista ordenada

Cortes Rojos

- Afectan a la semántica declarativa del programa
- Modifican el significado lógico del programa
- Al eliminar el corte se obtiene un programa incorrecto
- Deben emplearse con cuidado

- Ejemplo:
  - max(X,Y,X) :- X>Y, !.

    max(X,Y,Y).
  - ?- max(5,2,2).

Ejercicios (I): Uso del Corte

- Dado el siguiente programa lógico
  - p(X,Y):- q(X),s(Y),t(Y).
  - q(a).
    q(b).
  - s(a).
    s(b).
  - t(b).
- Se piden todas las respuestas a la pregunta "?- p(X,Y).", suponiendo
  - (a) el programa tal cual está (sin cortes)
  - (b) cambiando la primera cláusula por "p(X,Y):- q(X),!,s(Y),t(Y)."
  - (c) cambiando la primera cláusula por "p(X,Y):- q(X),s(Y),!,t(Y)."

Ejercicios (II): Uso del Corte

- Dado el siguiente programa lógico
  - a(1).

    a(2).
  - b(2).

    b(1).
  - c(2).
- Proporcionar todas las respuestas (una por línea) a la pregunta "?- n(X,Y).", siendo la definición de n/2 la indicada en cada caso
  - (a) n(X,Y):- a(X),b(Y),c(Y).
  - (b) n(X,Y):- a(X),!,b(Y),c(Y).
  - (c) n(X,Y):- a(X),b(Y),!,c(Y).

Ejercicios (III): Uso del Corte

- Sea el siguiente programa Prolog
  - p(1).

    p(2) :- !.

    p(3).
- Escribir todas las respuestas a las siguientes consultas
  - (a) ?- p(X).
  - (b) ?- p(X), p(Y).
  - (c) ?- p(X), !, p(Y).

Ejercicios (IV): Uso del Corte

- Definir un predicado para clasificación de notas numéricas, donde
  - Suspenso: nota menor que cinco
  - Aprobado: nota entre cinco y siete
  - Notable: nota entre siete y nueve
  - Sobresaliente: nota mayor que nueve

Ejercicios (V): Uso del Corte

- Definir un predicado funcionF(X,Y), de tal forma que para un X dado, la variable Y se instancie al valor de la siguiente función f(X)

$$f(X) = \begin{cases} 0 & X \leq 3, \\ 2 & 3 < X \leq 6, \\ 4 & X > 6. \end{cases}$$

Ejercicios (VII): Uso del Corte

- Definir un predicado listaMujeres(Xs,Ys) que se verifica si Ys contiene sólo las mujeres de Xs, siendo Xs e Ys listas

Ejercicio: Uso del Corte

- Borrar todas las apariciones de un cierto elemento en una lista dada
  - borrar(X, L1, L2): L2 es la lista obtenida al borrar todas las apariciones de X en la lista L1
    - Modo de uso (in, in, out), es decir, primer y segundo parámetros de entrada y tercer parámetro de salida

Ejercicio: Uso del Corte

- Borrar todas las apariciones de un cierto elemento en una lista dada (borrar(X, L1, L2))
  - borrar(X, [ ], [ ]).

    borrar(X, [X|L], LN) :- !, borrar(X, L, LN).

    borrar(X, [Z|L], [Z|LN]) :- X \= Z, borrar(X, L, LN).

  - borrar2(X, [ ], [ ]).

    borrar2(X, [X|L], LN) :- !, borrar2(X, L, LN).

    borrar2(X, [Z|L], [Z|LN]) :- borrar2(X, L, LN).

Programas *Generate & Test*

- Son básicamente programas que generan soluciones candidatas que se evalúan para comprobar si son o no correctas
  - En algunas ocasiones es más sencillo comprobar si algo es una solución a un problema que crear la solución a dicho problema
- Consisten en dividir la resolución de problemas en dos partes
  - Generar soluciones candidatas
  - Testear que las soluciones sean correctas
- Son programas con la siguiente estructura
  - Una serie de objetivos generan posibles soluciones vía *backtracking*
  - Otros objetivos comprueban si dichas soluciones son las apropiadas

Programas *Generate & Test*: Ejemplo 1

- Identificación de partes en una oración
  - Una sentencia se representa como lista de palabras

  - verb(Sentence,Word):-
    member(Word,Sentence), verb(Word).
  - noun(Sentence,Word):-
    member(Word,Sentence), noun(Word).
  - article(Sentence,Word):-
    member(Word,Sentence), article(Word).

  - noun(man).
  - noun(woman).
  - verb(loves).
  - article(a).

  - ?- verb([a, man, loves, a ,woman], V).
    - V = loves

(1) **Generación**: member/2

(2) **Prueba**: verb/1; noun/1; article/1

Programas *Generate & Test*: Ejemplo 2

- Ordenación de listas
  - ordenacion(X,Y)
    - Y es la lista resultante de ordenar la lista X de forma ascendente
    - La lista Y contiene, en orden ascendente, los mismos elementos que la lista X
    - La lista Y es una permutación de la lista X con los elementos en orden ascendente
  - ?- ordenacion([2,1,2,3], L).
    - L = [1,2,2,3]
  - ordenacion(X,Y) :-

    permutacion(X,Y),

    ordenada_ascendente(Y).

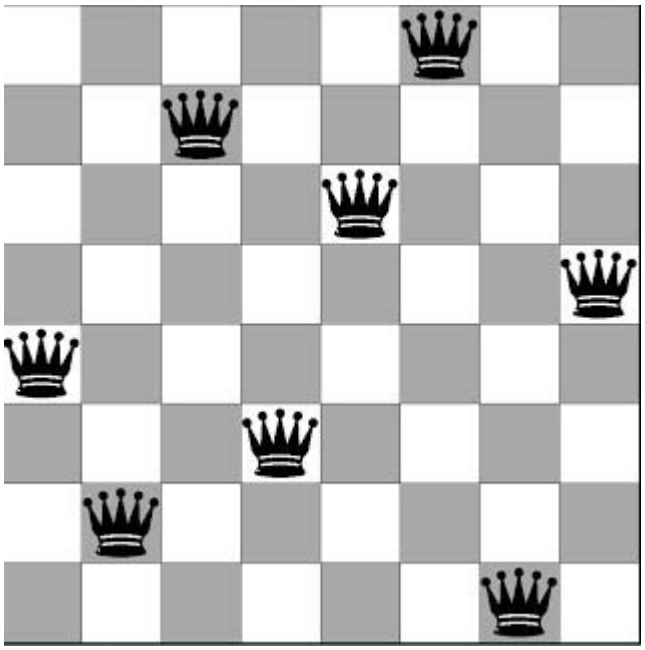
    (1) **Generación**: se obtiene una permutación de X en Y que pasa al objetivo (2) para comprobar si Y está ordenada

    (2) **Prueba**: comprueba si la lista está ordenada. Si no lo está, el *backtracking* se encarga de re-satisfacer el objetivo (1) buscando una nueva permutación

ejemploGenerateTest.pl

Programas *Generate & Test*: Ejemplo 3

- El problema de las N reinas: colocar N reinas en un tablero de ajedrez de manera que las reinas no se ataquen unas a otras
  - No puede haber dos reinas en la misma línea (horizontal, vertical o diagonal)

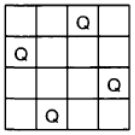


Programas *Generate & Test*: Ejemplo 3

- El problema de las N reinas: colocar N reinas en un tablero de ajedrez de manera que las reinas no se ataquen unas a otras
  - No puede haber dos reinas en la misma línea (horizontal, vertical o diagonal)
  - reinas(N,Tablero)

reinas(N,Tablero) es cierto si Tablero es una solución al problema de las N reinas.

Las soluciones se representan como una permutación de la lista de números entre 1 y N: el primer elemento es la fila en la que se sitúa la reina de la primera columna, el segundo la fila de la reina de la segunda columna, y así sucesivamente.



[2,4,1,3]

Programas *Generate & Test*: Ejemplo 3

- reinas(N,Tablero) :-

  range(1,N,L), Crea la lista de números entre 1 y N

  permutation(L,Tablero), Crea una permutación de la lista

  safe(Tablero). Comprueba si la permutación es solución al problema

- safe([]).

  safe([Q|Qs]) :- safe(Qs), not (attack(Q,Qs)).

- attack(X,Xs) :- attack(X,1,Xs).

- attack(X,N,[Y|Ys]) :- X is Y+N; X is Y-N.

  attack(X,N,[Y|Ys]) :- N1 is N+1, attack(X,N1,Ys).

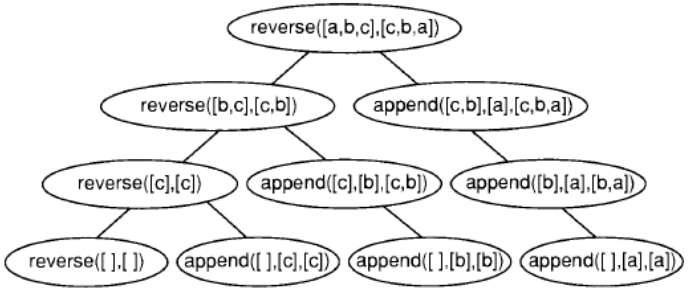
- Esta solución es ineficiente:
  - se generan muchas permutaciones que no pueden ser solución
- Una solución más eficiente consiste en aplicar la **técnica de los acumuladores**

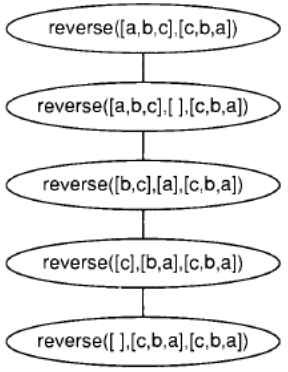
Técnica de los Acumuladores (I)

- Ejemplo: Inversa de una lista
  - reverse(Xs,Ys): Ys es la lista que se obtiene al invertir los elementos de la lista Xs
    - reverse([],[]).
      reverse([X|Xs],Ys) :- reverse(Xs,Zs), append(Zs,[X],Ys).

      reverse/2 es ineficiente
    - reverse(Xs,Ys) :- reverse(Xs,[],Ys).
    - reverse([],Ys,Ys).
      reverse([X|Xs],Acc,Ys) :- reverse(Xs,[X|Acc],Ys).

      Otra opción (reverse/2) usando **acumuladores**





Técnica de los Acumuladores (II)

- **Acumuladores**:
  - son argumentos adicionales usados en los predicados para almacenar resultados intermedios
    - se usan para simular algoritmos iterativos
  - son variables lógicas y su valor se pasa entre iteraciones
- Ejemplo 1: Factorial
  - factorial(N,F) :- factorial(0,N,1,F).
  - factorial(N,N,F,F).

    factorial(I,N,T,F) :-

    I < N,

    I1 is I+1,

    T1 is T*I1,

    factorial(I1,N,T1,F).

Este programa lógico simula el comportamiento de un programa iterativo con bucle *while* (de 0 a N)

El **primer argumento** en factorial/4 es el contador del bucle

El **tercer argumento** en factorial/4 es el acumulador de los productos calculados

Técnica de los Acumuladores (III)

- Ejemplo 2: Otra versión de Factorial
  - factorial(N,F) :- factorial(N,1,F).
  - factorial(0,F,F).

    factorial(N,T,F) :-

    N > 0,

    T1 is T*N,

    N1 is N-1,

    factorial(N1,T1,F).

Versión iterativa de factorial desde N hasta 0 (bucle *while*)

El **segundo argumento** en factorial/3 actúa como acumulador de los productos calculados

- La versión 2 es más eficiente que la versión 1
  - Normalmente, cuantos menos argumentos tiene un predicado, más rápido y más entendible es

Ejercicio: *Generate & Test*

- Definir el predicado numeroParMenor/2 que es verdadero cuando X es un numero par menor que N y mayor que cero
  - ?- numeroParMenor(X,5).
    - X=0; X=2; X=4
  - ?- numeroParMenor(2,4).
    - Yes
  - ?- numeroParMenor(3,5).
    - No
  - ?- numeroParMenor(10,7).
    - No

Ejercicio: *Generate & Test*

- Estrategia: generar todos los números entre 0 y N y comprobar si son pares
  - numParMenor(X,N) :-
    entre(0,N,X),
    par(X).
  - par(X) :- 0 is X mod 2.
  - **entre** es el encargado de generar todos los números menores que N
  - **par** actúa como filtro

generateTest-numParMenor.pl

Programación Declarativa:
Lógica y Restricciones

Conceptos Básicos de la Programación en Prolog

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