PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL1: Introducción al paradigma de la Programación Lógica

Grado en Ingeniería Informática

URJC

Ana Pradera

Contenido

1. INTRODUCCIÓN
2. FUNDAMENTOS TEÓRICOS
   - Lógica Matemática
   - Demostración Automática
3. DESCRIPCIÓN Y FUNCIONAMIENTO
   - Programación Lógica
   - Esquema de funcionamiento
   - Ejemplo
4. CARACTERÍSTICAS Y APLICACIONES
   - Características básicas
   - Ejemplo
   - Aplicaciones

INTRODUCCIÓN

Este tema está dedicado a describir brevemente los aspectos fundamentales del paradigma de la Programación Lógica, en particular:

- Sus fundamentos teóricos:

  Lógica Matemática. Demostración Automática.

- Su funcionamiento:

  Esquema general de funcionamiento. Ejemplos.

- Sus características y aplicaciones:

  Propiedades. Diferencias con otros paradigmas. Aplicaciones típicas.

FUNDAMENTOS TEÓRICOS

Programación Lógica

=

uso de la Lógica Matemática y de la Demostración Automática como herramientas de **cómputo**

- La Lógica Matemática se usa para la **representación del conocimiento** (enfoque *declarativo*) y para la **resolución de problemas**.
- Los sistemas de demostración de la Demostración Automática se usan para **realizar cómputos** de forma **eficiente**.

Definición (Lógica Matemática)

La Lógica Matemática (Lógica Formal, Lógica Simbólica) es la ciencia que estudia la *validez formal* de los **razonamientos**.

- *Razonamiento* (deducción, inferencia, argumentación)

  Obtención de nuevo conocimiento (*conclusión*) a partir de una serie de conocimientos previos (*premisas*).

- *Validez formal*

  Un razonamiento es formalmente válido cuando, suponiendo que todas sus premisas son verdaderas, la conclusión es necesariamente verdadera (es válido en virtud de su forma -su estructura-, independientemente del conocimiento concreto del que se trate). Cuando esto ocurre se dice que la conclusión es una *consecuencia lógica* (*se deduce*) de las premisas.

Para conseguir su cometido la Lógica Matemática dispone de:

Lenguajes Lógicos

Permiten formalizar los razonamientos:

- Su **sintaxis** describe los elementos básicos de los lenguajes y las reglas que permiten construir las expresiones admitidas por ellos, denominadas *términos* (que sirven para representar objetos) y *fórmulas* (que permiten expresar hechos relativos a objetos).
- Su **semántica** permite asignar un significado (valor de verdad, cierto o falso) a las fórmulas, y definir qué significa que un razonamiento sea válido.

Sistemas de demostración

Son sistemas formales que permiten averiguar, mediante la aplicación de una serie de *reglas de inferencia*, si un razonamiento es válido.

Ejemplo (La mortalidad de Sócrates)

Dadas las premisas "*Todas las personas son mortales*" y "*Sócrates es una persona*", ¿se deduce lógicamente "*Hay alguien mortal*"?

1. *Formalización del razonamiento* (en Lógica de Predicados -o Lógica de Primer Orden, LPO-, uno de los lenguajes lógicos más habituales):

Premisa 1:	$\forall X(persona(X) \rightarrow mortal(X))$
Premisa 2:	$persona(sócrates)$
Posible conclusión:	$¿\exists Y\ mortal(Y)?$

2. *Estudio de la validez del razonamiento*: en caso de que las dos premisas sean ciertas, ¿lo es necesariamente también la conclusión? La respuesta es "sí", y se obtiene aplicando a las fórmulas anteriores las reglas de inferencia de cualquiera de los sistemas de demostración existentes para la Lógica de Predicados (sistema de Hilbert, deducción natural, resolución, tableaux semánticos, ...).

Muy breve historia de la Lógica Matemática

- La Lógica fue introducida en el marco de la filosofía por el filósofo griego **Aristóteles** (384-322 A.C).
- El matemático alemán **Leibniz** (siglo XVII) fue el primero en plantear la formalización de la lógica como cálculo matemático, pero esto no se completó hasta mediados del siglo XIX con los trabajos, entre otros, de los matemáticos ingleses **Boole** y **De Morgan**, que aplicaron a la Lógica métodos algebraicos.
- El gran desarrollo de la Lógica Formal se produjo a finales del siglo XIX y primera mitad del XX, con las aportaciones, entre otros, de **Frege, Russell y Whitehead, Hilbert, Herbrand, Church, Turing, Gödel, ...**
- A partir de mediados del siglo XX una parte importante de la investigación en Lógica se centra en el estudio de sus aplicaciones en Informática, en particular como herramienta de programación.

Definición (Demostración Automática)

La Demostración Automática estudia la construcción de sistemas de demostración lógicos que sean adecuados para ser aplicados de forma mecánica y que se puedan *ejecutar de forma eficiente en un ordenador*.

- La Demostración Automática (o Demostración Automática de Teoremas) surge alrededor de 1950, con la aparición de los primeros ordenadores.
- Su objetivo es implementar sistemas de demostración lógicos en los ordenadores, de forma que estos sean capaces de decidir cuándo un razonamiento es válido (convirtiendo así a los ordenadores en máquinas capaces de razonar, y por lo tanto ¡inteligentes!).

- La mayoría de los sistemas de demostración lógicos existentes por aquel entonces resultan poco apropiados para ser implementados en un ordenador y tremendamente ineficientes.

- La investigación en Demostración Automática consigue que en 1960 ya se disponga de sistemas de demostración mucho más eficientes que los anteriores (**Gilmore, Davis-Putnam, ...**).

- El cambio cualitativo se produce en 1965 con la aparición del **Sistema de Resolución con unificación** de **Robinson**: sistema muy eficiente, sencillo y de fácil implementación, adoptado pocos años después por el primer lenguaje de Programación Lógica, el lenguaje PROLOG.

DESCRIPCIÓN Y FUNCIONAMIENTO

Definición (Programación Lógica)

La Programación Lógica es un paradigma de programación que se basa en el uso de la Lógica Matemática y la Demostración Automática como **herramientas de programación**.

- Fue introducida a principios de la década de 1970 por, entre otros, **Greene**, **Kowalski** y **Colmerauer**.
- Se enmarca dentro del campo de la Inteligencia Artificial (cuyo objetivo es la construcción de productos informáticos capaces de reproducir comportamientos "inteligentes").
- Supone un paso más allá respecto a la Demostración Automática: surge de la idea de que la Lógica no sólo se puede usar en un ordenador para demostrar la validez de razonamientos (respuesta "sí o no"), sino también para computar.

- Todo entorno de Programación Lógica consta de los dos siguientes componentes:
  1. Un lenguaje lógico que permite representar el conocimiento.
  2. Un sistema de Demostración Automática para resolver los problemas y computar las soluciones.
- Existen por ende distintos sistemas de Programación Lógica, dependiendo tanto del lenguaje utilizado para representar el conocimiento como del mecanismo de Demostración Automática elegido para computar.
- El primer lenguaje de Programación Lógica, PROLOG, se implementó en la Universidad de Marsella en 1972 por un equipo dirigido por **Colmerauer**.

- En los años 80 aparecen otros lenguajes de Programación Lógica, así como los primeros libros y las primeras implementaciones comerciales de PROLOG.
- Desde entonces el paradigma de la Programación Lógica sigue evolucionando, apareciendo distintas variantes y mezclándose a menudo con otros paradigmas de programación. Surgen así, por ejemplo, la programación lógica concurrente, la programación lógico-funcional, la programación lógica con restricciones, la integración con sistemas orientados a objetos, etc.
- En 1995 se define el estándar ISO-PROLOG, basado en la implementación de la Universidad de Edimburgo (el estándar *de facto* hasta ese momento).

Esquema de funcionamiento

La resolución de un problema mediante Programación Lógica requiere los tres siguientes pasos:

PASO 1: escritura de un programa lógico

El conocimiento relativo al problema que se pretende resolver (el **qué**, no el *cómo*) se representa por medio de una serie de fórmulas lógicas, escritas en el lenguaje lógico asociado con el lenguaje de programación, que constituyen lo que se denomina un programa lógico.

programa lógico = conjunto de fórmulas lógicas

Este paso es **responsabilidad de la programadora o programador**.

PASO 2: escritura de una consulta

El problema a resolver se representa mediante una fórmula lógica, denominada consulta, involucrando a los predicados definidos en el programa lógico.

consulta = fórmula lógica

Este paso es **responsabilidad de la programadora o programador**.

PASO 3: resolución del problema

El problema se resuelve aplicando sobre programa y consulta el sistema de *Demostración Automática* asociado con el lenguaje de programación, con el siguiente doble objetivo:

1. Averiguar si la consulta expresada en el PASO 2 es una consecuencia lógica del conjunto de fórmulas que conforman el programa del PASO 1 (cuyas fórmulas actúan como premisas del razonamiento, mientras que la consulta actúa como posible conclusión).
2. Si efectivamente la consulta resulta ser una consecuencia lógica del programa y es además de tipo *existencial* (*¿Existen objetos X,Y, ... tales que ...?*), computar los valores que hacen que la consulta se deduzca del programa.

Este paso es **responsabilidad del lenguaje de programación**.

Ejemplo

El ejemplo de la página 7, desde el punto de vista de la Programación Lógica y usando la Lógica de Predicados como base:

PASO 1. El programa lógico estaría compuesto por las dos fórmulas

$$\forall X(persona(X) \rightarrow mortal(X))$$
$$persona(sócrates)$$

PASO 2. La consulta sería la fórmula existencial

$$\exists Y\ mortal(Y)$$

PASO 3. Resolución del problema:

Se obtendría una respuesta afirmativa (ya que la consulta es una consecuencia lógica de las dos fórmulas que componen el programa), con el cómputo $Y = sócrates$.

CARACTERÍSTICAS Y APLICACIONES

Una de las características fundamentales de la Programación Lógica, que marca además la diferencia con la Programación Imperativa, es la forma en la que implementa la conocida como **Ecuación de Kowalski**:

Ecuación de Kowalski

$$\boxed{\text{ALGORITMO} = \text{LÓGICA} + \text{CONTROL}}$$

Con la ecuación anterior, Kowalski establece que todo algoritmo consta de dos componentes:

1. Una componente lógica, que describe el conocimiento del que se dispone acerca del problema a resolver (**qué**).
2. Una componente de control, que describe la estrategia que se va a adoptar para solucionar el problema (**cómo**).

Ecuación de Kowalski y paradigmas de programación

- La Programación Imperativa no separa las dos componentes.
- La Programación Lógica las separa claramente:
  - De la componente lógica se responsabiliza el/la programador/a (escribiendo el programa lógico y la consulta).
  - De la componente de control se ocupa el lenguaje de programación (aplicando su sistema de Demostración Automática).

Otras características de la Programación Lógica

- La Programación Lógica prescinde de elementos típicos de otros paradigmas como declaraciones de tipos y variables, vectores, registros, subrutinas, bucles o instrucciones de asignación, y hace un uso extenso de la recursión.
- Los programas lógicos están más próximos a la descripción real del problema que pretenden resolver, por lo que en general son más sencillos, más fáciles de entender, mantener y verificar y más fiables.

Ejemplo (Programa “Abuelas/os”, 1 de 4)

Suponga que se tiene información sobre una serie de pares (progenitor/a, hijo/a) y se necesita un programa que averigüe, a partir de esa información, quiénes son los nietos/as de alguien.

Con Programación Imperativa:

Para escribir un programa imperativo que resuelva el problema anterior, se necesita decidir primero cómo representar la información (vector de registros, vector de vectores, ...) y después qué estrategia usar para computar las/os nietas/os de alguien. Hay varias opciones:

1. Buscar los hijos/as de la persona dada y, a continuación, los hijos/as de estos últimos.
2. Para cada persona, comprobar si alguno de sus progenitores es hija o hijo de la persona dada.
3. ....

Ejemplo (Programa “Abuelas/os”, 2 de 4)

Con Programación Lógica (usando Lógica de Predicados):

PASO 1. Escritura de un programa lógico:

La información se representa mediante *predicados* que establecen *propiedades* (si solo tienen un argumento) o *relaciones* entre sus argumentos (si son varios). En este caso serían necesarios los dos siguientes predicados:

- Predicado *progenitor*(X, Y), cierto si X es progenitor (madre o padre) de Y.
- Predicado *abuelo*(X, Y), cierto si X es abuela o abuelo de Y.

El programa lógico *declara* el conocimiento que se tiene, definiendo los predicados anteriores mediante las *fórmulas lógicas* necesarias.

Ejemplo (Programa “Abuelas/os”, 3 de 4)

- El predicado *progenitor*(X, Y) se define estableciendo las relaciones de progenitura que se conozcan mediante las siguientes fórmulas (en este caso fórmulas atómicas):

$$progenitor(pepa, pepito)$$
$$progenitor(pepito, pepita)$$
$$progenitor(pepito, pepón)$$

- El predicado *abuelo*(X, Y) se define declarando qué significa ser abuela/o mediante la fórmula condicional “para cualesquiera X, Y y Z, si X es progenitor de Y y ese Y a su vez es progenitor de Z, entonces X es abuela/o de Z”:

$$\forall X \forall Y \forall Z \Big((progenitor(X, Y) \wedge progenitor(Y, Z)) \rightarrow abuelo(X, Z) \Big)$$

Ejemplo (Programa “Abuelas/os”, 4 de 4)

PASO 2. Escritura de una consulta:

Para averiguar quiénes son los nietos de, por ejemplo, “pepa”, se establece la siguiente fórmula lógica, correspondiente a la pregunta “¿existe algún N tal que se cumpla la relación $abuelo(pepa, N)$?”:

$$\exists N \; abuelo(pepa, N)$$

PASO 3. Resolución del problema:

Ante el programa y la consulta anterior se obtiene una respuesta afirmativa (ya que la consulta es una consecuencia lógica de las fórmulas que componen el programa), con dos posibles cómputos:

Primera solución: $N = pepita$

Segunda solución: $N = pepón$

Programación Lógica versus Programación Funcional

- La Programación Lógica se basa en el uso de predicados, que establecen *propiedades o relaciones* entre objetos y son por ello más generales que las funciones de la Programación Funcional (toda función puede verse como un caso particular de predicado relacionando los argumentos de entrada de la función con su única salida).
- Los sistemas de Demostración Automática incorporan un mecanismo de búsqueda con retroceso que permite computar todas las posibles soluciones a un problema dado.
- En Programación Lógica los argumentos de los predicados pueden usarse a menudo en modos diferentes, dependiendo de si se facilita un valor concreto -modo entrada- o una variable -modo salida-. Por ello, los programas son más versátiles, puesto que sirven para varios propósitos distintos.

Ejemplo (Capacidad de búsqueda y versatilidad, 1 de 2)

El programa “Abuelas/os” del ejemplo anterior, además de para averiguar quiénes son las/os nietas/os de una persona dada, puede usarse, *sin necesidad de introducir ninguna modificación*, con consultas con otros propósitos distintos, como las siguientes:

1. Para averiguar quiénes son los abuelos de alguien, por ejemplo:

$$\exists A\ abuelo(A, pepón) \quad \text{o bien} \quad \exists A\ abuelo(A, pepito)$$

2. Para encontrar todas las parejas <abuela/o, nieta/o>:

$$\exists A \exists N\ abuelo(A, N)$$

3. Para comprobar si dos personas están relacionadas, por ejemplo:

$$abuelo(pepa, pepito), \quad \text{o bien} \quad progenitor(pepa, pepito)$$

Ejemplo (Capacidad de búsqueda y versatilidad, 2 de 2)

El sistema de Demostración Automática comprueba si las consultas dadas son o no consecuencia lógica del programa y, en caso afirmativo, busca *todas* las posibles soluciones (valores de las variables cuantificadas existencialmente, en caso de que las haya).

Así, las respuestas asociadas a las consultas anteriores, dadas las premisas que conforman el programa, serían:

1. $A = pepa$ (*pepón* solo tiene una abuela, *pepa*) y *false* (*pepito* no tiene abuelas/os).
2. Primera solución: $A = pepa, N = pepita$
   Segunda solución: $A = pepa, N = pepón$
3. *false* (pepa no es abuela de pepito) y *true* (pepa sí es progenitora de pepito)

Aplicaciones más comunes de la Programación Lógica

Aunque los lenguajes de Programación Lógica son lenguajes de programación de *propósito general*, resultan más adecuados para la resolución de problemas simbólicos que para la resolución de problemas numéricos. Algunos de los campos más destacados en los que se usa este paradigma son:

- Inteligencia Artificial, en particular problemas de planificación y búsqueda.
- Sistemas expertos, sistemas basados en el conocimiento.
- Reconocimiento y procesamiento de lenguaje natural.
- Sistemas de recomendación y asesoramiento.
- Manejo y mantenimiento de bases de datos.
- Análisis de redes sociales, Web Semántica.
- ...

BIBLIOGRAFÍA

- L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.
- W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fifth edition, 2003.
- I. Bratko.*Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 2001.
- J. Lloyd. *Foundations of Logic Programming*, (Second Edition). Springer-Verlag, 1987.
- R. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
- U. Nilsson and J. Maluszynski. Logic, Programming and Prolog. John Wiley & Sons Ltd, 1996.
- SWI-Prolog, entorno de programación en Prolog de dominio público.
- comp.lang.prolog. Faq

PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL2: El lenguaje PROLOG, aspectos básicos

1. Caraterísticas Generales

Grado en Ingeniería Informática

URJC

Ana Pradera

Contenido

1. EVOLUCIÓN HISTÓRICA
2. COMPONENTES DE PROLOG
3. ENTORNOS DE PROGRAMACIÓN EN PROLOG
4. ESQUEMA GENERAL DE TRABAJO EN SWISH PROLOG

EVOLUCIÓN HISTÓRICA

- PROLOG (del francés, PROgrammation en LOGique), es el primer lenguaje de programación basado en el paradigma de la Programación Lógica.
- Su primer intérprete lo implementa en 1972 un equipo dirigido por **A. Colmerauer** de la Universidad de Marsella (Francia), utilizando resultados teóricos aportados, entre otros, por **R. Kowalski** (Universidad de Edimburgo, Escocia).
- El lenguaje tiene una rápida expansión, sobre todo en Europa y Japón, definiéndose el estándar ISO en 1995.
- PROLOG no es un lenguaje declarativo "puro", ya que por razones de eficiencia incorpora el uso de elementos extra-lógicos con efectos laterales (para aritmética, entrada/salida, control en la búsqueda de soluciones, ...).

COMPONENTES DE PROLOG

Recuerde (Tema PL1) que todo entorno de Programación Lógica consta de dos componentes: un *lenguaje lógico* para la representación del conocimiento y un *sistema de Demostración Automática* para la resolución de problemas y el cómputo de soluciones.

PROLOG utiliza la conocida como Programación Lógica Definida:

- Su lenguaje es un subconjunto de la Lógica de Primer Orden, cuyas fórmulas se conocen como cláusulas de Horn (**A. Horn**, Universidad de California, EEUU) y cuya *sintaxis* se define en el siguiente apartado.
- Su sistema de Demostración, Resolución SLD, es una especialización del Sistema de Resolución de la Lógica de Predicados circunscrito a cláusulas de Horn. Su *semántica operacional* se describe en detalle más adelante.

ENTORNOS DE PROGRAMACIÓN EN PROLOG

- Existen múltiples entornos de programación en PROLOG, algunos comerciales y otros de libre distribución.
- La mayoría de ellos se adaptan al estándar ISO y ofrecen tanto intérprete como compilador.
- Entorno recomendado: SWI-Prolog
  - dominio público
  - multi-plataforma (Linux, MS-Windows, MacOS X)
  - compatible con el estándar ISO
  - incluye intérprete, compilador y múltiples bibliotecas
  - ofrece una herramienta online que no requiere instalación, SWISH. Esta herramienta no incluye todo el lenguaje, pero es suficiente para la realización de todos los ejercicios y prácticas de la asignatura.

ESQUEMA GENERAL DE TRABAJO EN SWISH PROLOG

PASO 1. Escritura de un programa lógico

- Un programa PROLOG es conjunto de fórmulas lógicas adaptadas a la sintaxis específica del lenguaje (que se detalla más adelante).
- Comentarios:
  - Para comentarios en una sola línea basta con anteponer el símbolo %.
  - Los comentarios en varias líneas empiezan con /* y terminan con */.
- En SWISH los programas se crean (botón azul “Create a program”) y editan directamente en la **parte izquierda de la ventana**.

PASO 2. Escritura de una consulta

- Para usar un programa hay que escribir consultas, fórmulas lógicas con una cierta sintaxis (que se detalla más adelante).
- En SWISH las consultas se escriben en la **zona inferior derecha** de la ventana y se activan con el botón “Run!” (esquina inferior derecha) o mediante la combinación de teclas CTRL+ENTER.
- Al recibir una consulta, lo primero que hace el sistema es *interpretar* tanto la consulta como el programa escrito en la parte izquierda.
- Posibles mensajes de error:
  1. Errores sintácticos: corregir errores y volver a activar la consulta.
  2. Avisos (warnings): repasar código en busca de posibles erratas.

PASO 3. Resolución del problema

- Si no hay errores sintácticos ni de ejecución, PROLOG pone en marcha su sistema de Demostración Automática y la respuesta, que aparece en la **parte superior derecha**, puede ser:
  - false si la consulta no es una consecuencia lógica del programa.
  - true si la consulta es una consecuencia lógica del programa pero es una mera comprobación (no hay nada que computar).
  - los valores que hacen cierta la consulta, si esta es una consecuencia lógica del programa y además es de tipo existencial. Si hay varias soluciones posibles, el sistema ofrece la primera y unos botones para obtener más (Next para la siguiente solución, 10/100/1000 para ver las 10/100/1000 siguientes soluciones y STOP para dejar de computar soluciones).
- Para entrar en modo depuración basta con anteponer la palabra `trace,` (seguida de una coma) a la consulta y utilizar los botones que ofrece el sistema para seguir la ejecución paso a paso.

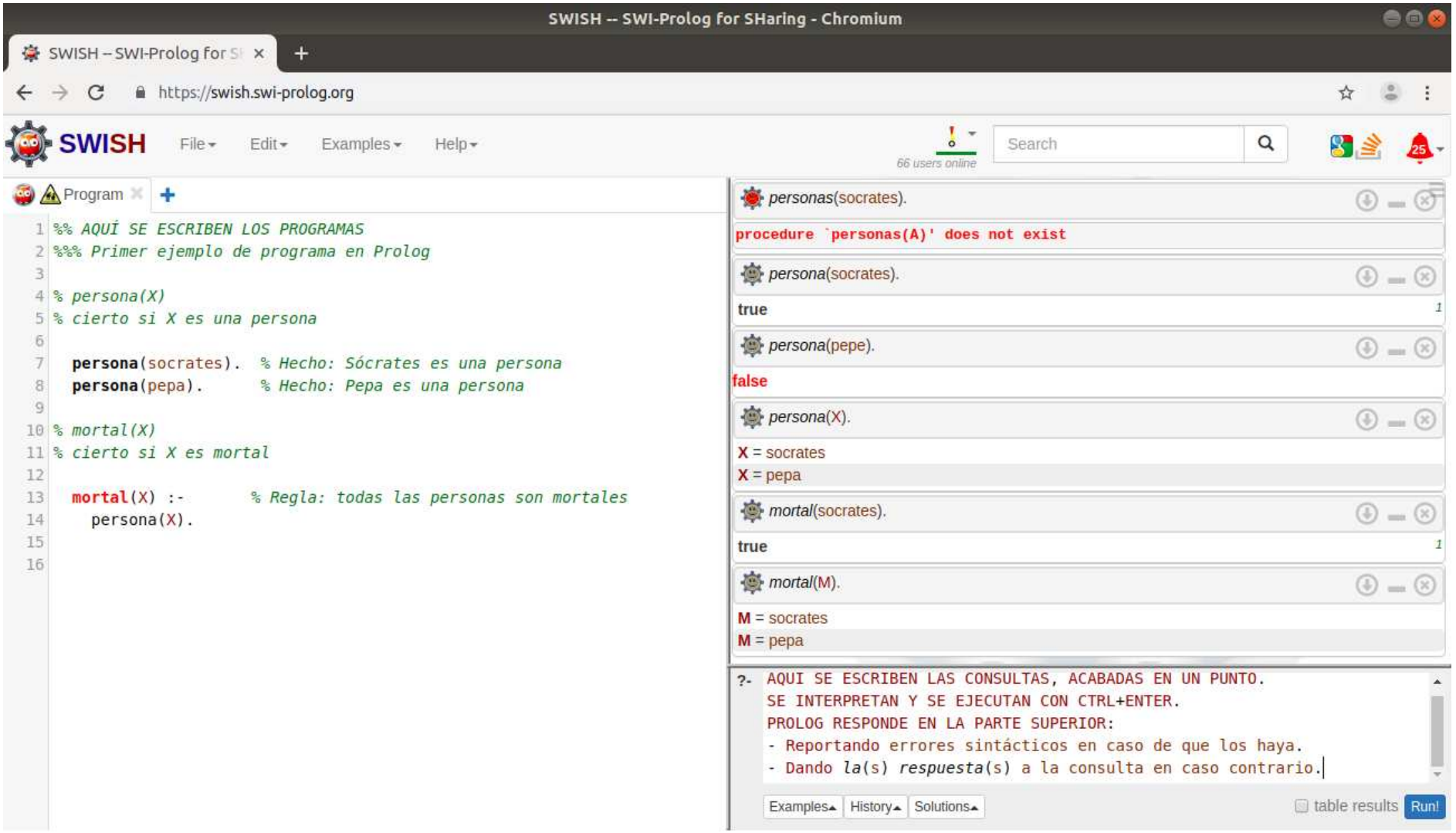


Figura: Captura de una sesión de trabajo en SWISH Prolog

Ejercicios (Primer contacto con SWISH: la mortalidad de Sócrates)

Ponga en práctica lo explicado en este apartado escribiendo en PROLOG *el problema relativo a la mortalidad de Sócrates, discutido en el tema PL1 y recogido en la captura de pantalla anterior:*

- *Abra el entorno de programación SWISH.*
- *Lea la ayuda en el menú "Help". Observe en particular que puede registrarse usando Google o Stackoverflow (esquina superior derecha), lo cual le dará ciertas ventajas para guardar/recuperar sus ficheros y decidir quién puede verlos/editarlos.*
- *Abra una ventana de edición de programas con el botón "Create Program" (arriba a la izquierda) y escriba el programa relativo a la mortalidad de Sócrates (ver captura de pantalla).*
- *Pruebe su programa escribiendo consultas (ver captura de pantalla) en la parte inferior derecha.*

BIBLIOGRAFÍA

- L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.
- W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fifth edition, 2003.
- I. Bratko.*Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 2001.
- J. Lloyd. *Foundations of Logic Programming*, (Second Edition). Springer-Verlag, 1987.
- R. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
- U. Nilsson and J. Maluszynski. Logic, Programming and Prolog. John Wiley & Sons Ltd, 1996.
- SWI-Prolog, entorno de programación en Prolog de dominio público.
- comp.lang.prolog. Faq

PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL2: El lenguaje PROLOG, aspectos básicos

2. Sintaxis

Grado en Ingeniería Informática

URJC

Ana Pradera

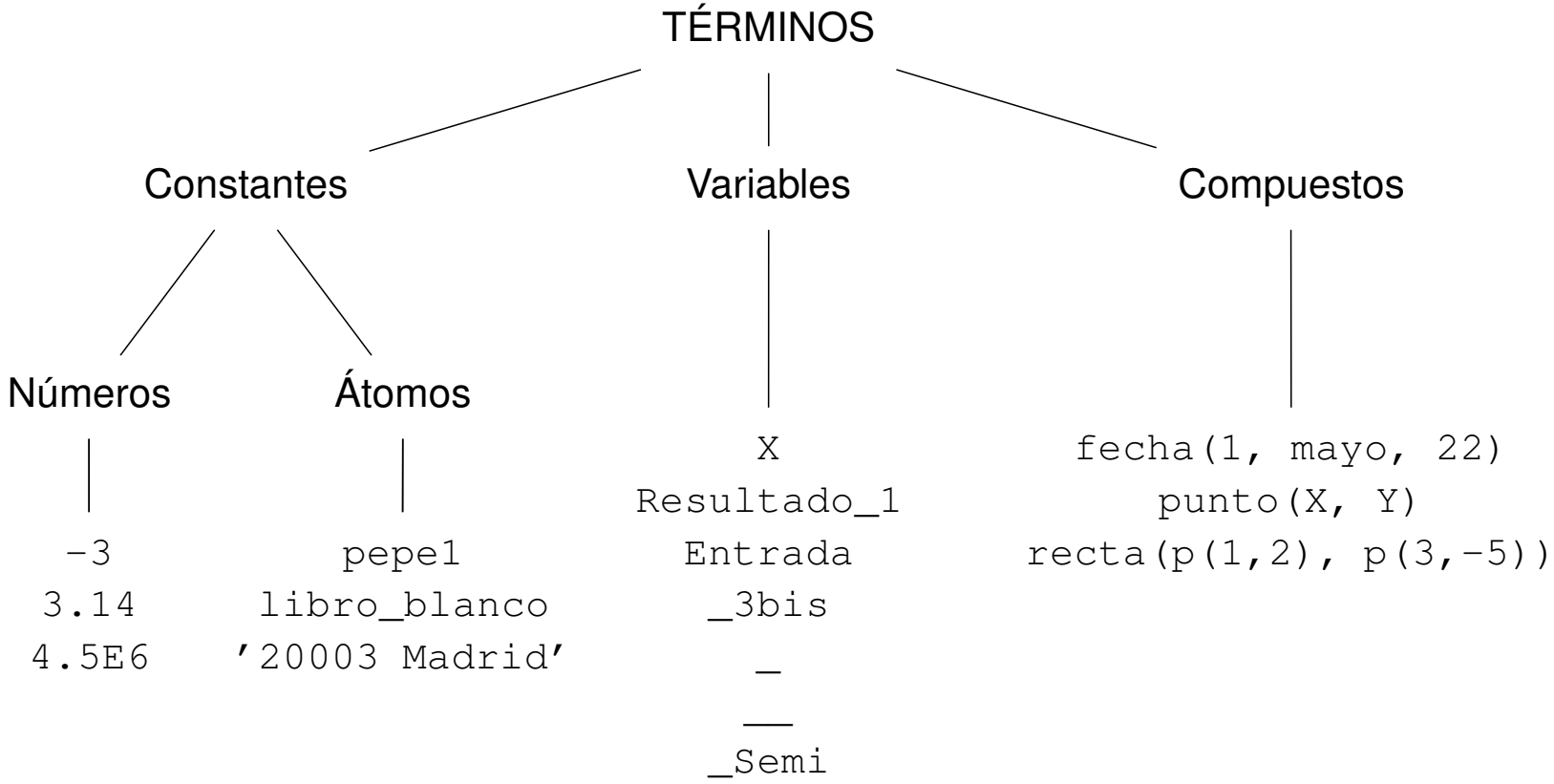
Contenido

1. ELEMENTOS BÁSICOS DE PROLOG
2. TÉRMINOS
3. PREDICADOS
4. PROGRAMAS
   - Ejemplo: programa “Abuelas/os”
   - Ejemplo: programa “Amistades/Enemistades”
   - Ejercicios: programa “Amistades/Enemistades”
   - Ejemplo: programa en lógica “pura” para sumar naturales
   - Ejercicios: programación en lógica “pura”
5. CONSULTAS
   - Ejercicios: consultas al programa “Amistades/Enemistades”
6. RESUMEN
   - Ejercicios: ejercicios 1 y 2 de la Práctica nº 1

ELEMENTOS BÁSICOS DE PROLOG

- Términos: sirven para representar *objetos* y constituyen los argumentos de los predicados.
- Predicados: tienen términos como argumentos y describen *propiedades o relaciones* entre sus argumentos.
- Programas: están compuestos por **cláusulas de Horn positivas**, fórmulas lógicas que involucran predicados y por lo tanto establecen propiedades o relaciones entre objetos. Actúan como premisas en los razonamientos.
- Consultas a programas: son fórmulas lógicas existenciales, denominadas **cláusulas de Horn negativas**, que involucran a uno o más predicados de un programa y que actúan como posibles conclusiones en los razonamientos.

TÉRMINOS



Los términos compuestos no admiten espacios entre su nombre y "(" : por ejemplo "punto (X,Y)" produciría un error sintáctico.

- **Números**: notación usual, formato decimal o exponencial.
- **Átomos**:
  - cadenas de letras, dígitos o subrayado, empezando por letra minúscula.
  - o bien cadenas de caracteres entre comillas simples.
- **Variables**: cadenas de letras, dígitos o subrayado, empezando por letra mayúscula o subrayado. La variable _ se denomina *variable anónima* y la variable ___, junto con todas las que empiezan por subrayado seguido de letra mayúscula, son *variables semianónimas*. Las particularidades de estas variables se discuten más adelante.
- **Términos compuestos (estructuras)**: constan de un nombre, que se denomina functor y se representa mediante un *átomo*, seguido, entre paréntesis, por una serie de argumentos, separados por comas, que son, a su vez, *términos* (constantes, variables o compuestos), por lo que los términos son *recursivos*. La aridad de un término compuesto es su número de argumentos.

PREDICADOS

- Tienen la misma sintaxis que los términos compuestos. Todo predicado tiene:
  1. Un nombre o functor, que se representa mediante un *átomo* y se puede sobrecargar (ver ejemplo a continuación).
  2. Cero o más argumentos, entre paréntesis y separados por comas, representados mediante *términos*.
- Establecen propiedades (si tienen menos de dos argumentos) o relaciones entre sus argumentos (cuando hay al menos dos).

Ejemplos

- `llueve` (predicado sin argumentos).
- `animal(serpiente)` (la serpiente es un animal).
- `abuela(pepa, pepon)` (pepa es abuela de pepon).
- `abuela(pepa)` (pepa es abuela). % sobrecarga de `abuela`

Convenios para describir predicados en libros y manuales

- `nombre_predicado/n`, siendo `n` la aridad (número de argumentos) del predicado.
- `nombre_predicado(<uso>Var_1, ...., <uso>Var_n)`, siendo `<uso>`:
  - `<uso> = +` : el argumento correspondiente debe estar, en el momento de usar el predicado, instanciado con un término *no variable* (parámetro de entrada).
  - `<uso> = -` : el argumento correspondiente *no* debe estar instanciado, es decir, debe ser una variable (parámetro de salida).
  - `<uso> = ?` : el argumento puede estar tanto instanciado como no instanciado (parámetro que se puede usar tanto para entrada como para salida).

PROGRAMAS

- Programa PROLOG = conjunto finito de cláusulas de Horn positivas, que pueden ser de dos clases: hechos o reglas.
- Hechos. Los hechos se escriben en PROLOG mediante

```
B.    % acabado en punto
```

donde `B` es un *predicado*, y se corresponden con fórmulas lógicas de la Lógica de Predicados o Lógica de Primer Orden (LPO) de la forma

$$\forall X_1 \ldots \forall X_p(B)$$

siendo $X_1, \ldots, X_p$, $p \geq 0$, las variables que, en caso de que las haya, aparecen en B.

- Reglas. Las reglas se escriben en PROLOG como

```
B :-
    B1,
    ...,
    Bm.     % acabado en punto
```

siendo `B, B1,...,Bm` *predicados*, y se corresponden con fórmulas lógicas de la LPO de la forma

$$\forall X_1 \ldots \forall X_p \Big((B1 \wedge \ldots \wedge Bm) \rightarrow B \Big)$$

donde $X_1, \ldots, X_p$, $p \geq 0$, son las variables que aparecen en `B, B1,..., Bm`, en caso de que las haya.
`B` es la cabeza de la regla y `B1,..., Bm` constituye su cuerpo.
Si `B` $\in$ {`B1,..., Bm`}, se dice que la regla es recursiva.

Significado de hechos y reglas

- Significado de un hecho

```
B.
```

  Para cualesquiera objetos $X_1, \ldots, X_p$ (variables que aparecen en `B`, si las hay), se cumple `B`.

- Significado de una regla

```
B :-
    B1,
    ...,
    Bm.
```

  Para cualesquiera objetos $X_1, \ldots, X_p$ (variables que aparecen en `B, B1,..., Bm`, si las hay), *si* se cumplen `B1` y ... y `Bm`
  ($, = \wedge =$ *y* $=$ conjunción) *entonces*
  ($\rightarrow =$ si...entonces $=$ implicación) también se cumple `B`.

Ejemplo (Programa “Abuelas/os”, hecho en LPO en PL1, 1/2)

```
% progenitor(?X,?Y)
% cierto si X es progenitor/a (padre o madre) de Y
% progenitor se describe mediante 3 HECHOS

    progenitor(pepa, pepito).
    progenitor(pepito, pepita).
    progenitor(pepito, pepon).

% abuelo(?X,?Z): cierto si X es abuela/o de Z
% abuelo/2 se describe mediante la siguiente REGLA

    abuelo(X,Z) :-
        progenitor(X,Y),
        progenitor(Y,Z).
```

Ejemplo (Programa “Abuelas/os” 2/2)

```
% abuelo(?X): cierto si X es abuela/o
% abuelo/1 se describe mediante la siguiente REGLA
    abuelo(X) :-
        abuelo(X,_).  % o abuelo(X,Y).
```

Observación

- Definición de `abuelo/1`: para cualesquiera `X` e `Y`, si se cumple `abuelo(X,Y)` entonces se cumple `abuelo(X)`.
  En LPO: $\forall X \forall Y(abuelo(X, Y) \rightarrow abuelo(X))$.
- La variable `Y` se puede reemplaza por la *variable anónima* _ puesto que su valor no interesa (esto evita además el aviso del intérprete “singleton variable [Y]”).

Ejemplo (Programa “Amistades/Enemistades”)

```
% enemigo(?X,?Y): cierto si X es enemiga/o de Y
% enemigo se describe mediante 4 HECHOS
    enemigo(abel, cain).
    enemigo(cain, blas).
    enemigo(cain, dolores).
    enemigo(blas, abilio).

% amigo(?X,?Y): cierto si X es amiga/o de Y
% amigo se describe mediante 2 REGLAS
    amigo(abilio, X) :-
        amigo(abel, X).

    amigo(X,Y) :-
        enemigo(X,Z),
        enemigo(Z,Y).
```

Ejercicios (Programa “Amistades/Enemistades”)

1. *Describa en Lógica de Primer Orden (LPO) y en lenguaje natural el conocimiento expresado por cada una de las dos reglas que definen el predicado* `amigo`.
2. *Escriba en LPO y en* PROLOG *el conocimiento expresado por las siguientes frases:*
   - *Los que son enemigos de Caín y de Abel también son enemigos de Abilio.*
   - *Dolores es amiga de cualquiera que sea enemigo de Abilio o de Blas.*
   - *Caín es enemigo de todos.*
3. *Describa en LPO y en lenguaje natural el conocimiento que describe el siguiente hecho:*

   ```
   amigo(abilio, X).
   ```

4. *Defina en* PROLOG *el predicado* `tiene_enemigos(?X)`, *cierto si* `X` *tiene algún enemigo.*

Soluciones propuestas:

1.

- La primera regla de `amigo` se corresponde con la fórmula de la LPO

$$\forall X \Big(amigo(abel, X) \rightarrow amigo(abilio, X) \Big)$$

  es decir, para cualquier persona X, si Abel es amigo de X, entonces Abilio también es amigo de X, o bien, equivalentemente, Abilio es amigo de todos los amigos de Abel.

- La segunda regla de `amigo` se corresponde con la fórmula de la LPO

$$\forall X \forall Y \forall Z \Big((enemigo(X, Z) \wedge enemigo(Z, Y)) \rightarrow amigo(X, Y) \Big)$$

  es decir, para cualesquiera personas X, Y, Z, si X es enemigo de Z y a su vez Z es enemigo de Y, entonces X es amigo de Y, o bien, equivalentemente, todo el mundo es amigo de los enemigos de sus enemigos.

2.

- En LPO:

$$\forall X \Big((enemigo(X, cain) \wedge enemigo(X, abel)) \rightarrow enemigo(X, abilio) \Big)$$

La fórmula anterior da lugar a la siguiente regla en PROLOG:
enemigo(X, abilio) :- enemigo(X, cain), enemigo(X, abel).

- En LPO:

$$\forall X \Big((enemigo(X, abilio) \vee enemigo(X, blas)) \rightarrow amigo(dolores, X) \Big)$$

Lo anterior es lógicamente equivalente a

$$\begin{aligned} & \forall X \Big((enemigo(X, abilio) \rightarrow amigo(dolores, X) \Big) \\ \wedge \quad & \forall X \Big((enemigo(X, blas) \rightarrow amigo(dolores, X) \Big) \end{aligned}$$

que da lugar a las dos siguientes reglas en PROLOG:
amigo(dolores, X) :- enemigo(X, abilio).
amigo(dolores, X) :- enemigo(X, blas).

- En LPO

$$\forall X \; enemigo(cain, X)$$

fórmula a la que le corresponde el hecho en PROLOG
enemigo(cain, X).

3. El hecho *amigo*(*abilio*, *X*) se corresponde con la fórmula lógica $\forall X$ *amigo*(*abilio*, *X*), es decir, Abilio es amigo de todo el mundo. Observe que aunque el hecho *amigo*(*abilio*, *X*) es correcto, la variable *X* solo aparece una vez, por lo que, para evitar avisos del intérprete (*singleton variables: [X]*), la variable *X* se podría reemplazar por la variable anónima _, dando lugar al hecho *amigo*(*abilio*, _).

4. Dada una persona cualquiera *X*, la propiedad *tiene_enemigos*(*X*) se cumplirá si existe al menos otra persona, pongamos *Y*, tal que se cumple la relación *enemigo*(*Y*, *X*). Por lo tanto, para cualesquiera personas *X* e *Y*, si resulta *enemigo*(*Y*, *X*), entonces se puede concluir *tiene_enemigos*(*X*). Lo anterior, escrito en LPO, da lugar a la fórmula

$$\forall X \forall Y \ (\textit{enemigo}(Y, X) \rightarrow \textit{tiene\_enemigos}(X))$$

La fórmula anterior se corresponde con la siguiente regla en PROLOG:

```
tiene_enemigos(X) :- enemigo(Y, X).
```

o bien, para evitar avisos del intérprete (*singleton variables: [Y]*), la *Y* se puede reemplazar por la variable anónima _ puesto que no juega ningún papel:

```
tiene_enemigos(X) :- enemigo(_, X).
```

Ejemplo (Programa en lógica “pura” para sumar naturales)

- Un ejercicio clásico para ilustrar el uso y la potencia de PROLOG es la implementación de un programa para *sumar números naturales* con lógica “pura”, es decir, sin usar las facilidades aritméticas de PROLOG (que se estudian más adelante).
- La suma es una *función* matemática que recibe como entrada dos naturales y devuelve su suma. Para implementar una función en PROLOG, que solo dispone de predicados (relaciones), basta con definir un predicado con los mismos argumentos que la función pero añadiendo un argumento adicional para el resultado (colocado, por convenio, al final). En este caso:

```
suma(?X,?Y,?Z)
cierto si Z es la suma de X e Y
```

- Para trabajar en lógica “pura” se necesita un mecanismo para representar números naturales. Uno sencillo consiste en usar un símbolo, por ejemplo `0`, para denotar el número natural cero, y otro, por ejemplo `s`, para construir el término compuesto `s(X)` representando el sucesor del número `X`, de forma que s(0) representa el número 1, s(s(0)) representa el 2, etc.
- Por otro lado, dado que PROLOG trabaja con recursividad, hay que partir de una definición *recursiva* de la suma:

$$\text{Para cualquier } X \in \mathbb{N}, \quad X + 0 = X \tag{1}$$

$$\text{Para cualesquiera } X, Y \in \mathbb{N}, \quad X + (Y + 1) = (X + Y) + 1 \tag{2}$$

(1) establece que la suma de cualquier número con cero es él mismo, mientras que (2) asegura que si se conoce la suma de dos números, $X + Y$, entonces también se conoce la suma del primero con el sucesor del segundo, $X + (Y + 1)$, que no es otra cosa que el sucesor del valor $X + Y$, es decir, $(X + Y) + 1$.

- Lo anterior, utilizando el predicado y los símbolos definidos previamente, se expresa mediante las siguientes fórmulas:

$$\forall X \; suma(X, 0, X) \tag{1}$$

$$\forall X \, \forall Y \forall Z \Big(suma(X, Y, Z) \rightarrow suma(X, s(Y), s(Z)) \Big) \tag{2}$$

En PROLOG:

```
% suma(?X,?Y,?Z): cierto si Z es la suma de X e Y

% (1) Caso base (expresado mediante un hecho)
      suma(X,0,X).

% (2) Caso recursivo (expresado mediante una regla)
      suma(X,s(Y),s(Z)) :-
          suma(X,Y,Z).
```

Ejercicios (Programación en lógica “pura”)

1. *Suponga que la primera cláusula de* `suma` *fuese* `suma(0,X,X).` *en lugar de la dada. ¿Cómo sería el resto del programa?*
2. *Escriba sendos programas lógicos “puros” capaces de decidir si un número natural es par (impar). Es decir, implemente los predicados* `par(?X)` *e* `impar(?X)`*, ciertos si* `X` *es par (impar) sin usar aritmética y manejando los números naturales mediante la constante cero y la función sucesor* `s(X)` *como se ha hecho para la suma.*
3. *Use el predicado* `suma/3` *para definir mediante lógica “pura” el predicado sobre números naturales* `producto(?X,?Y,?Z)`*, cierto si* `Z` *es el producto de* `X` *por* `Y`*. Parta de la siguiente definición recursiva del producto:*

$$\text{Para cualquier } X \in \mathbb{N}, \quad X \times 0 = 0$$

$$\text{Para cualesquiera } X, Y \in \mathbb{N}, \quad X \times (Y + 1) = (X \times Y) + X$$

Soluciones propuestas:

1. El cambio propuesto hace que la recursión se lleve a cabo en el primer argumento en lugar de en el segundo como se hace en el código original. Este cambio no tiene mayor importancia, dado que la suma es conmutativa, siempre que se adapte la segunda cláusula de la siguiente forma:

```
suma(s(X),Y,s(Z)) :- suma(X,Y,Z).
```

2.

```
par(0). % 0 es par
 % para todo X, si X es par, entonces s(s(X)) también lo es.
par(s(s(X))) :- par(X).

impar(s(0)). % 1 es impar
 % para todo X, si X es impar, entonces s(s(X)) también lo es.
impar(s(s(X))) :- impar(X).
```

Una implementación alternativa para `impar`, basada en el predicado `par`:

```
 % para todo X, si X es par, entonces s(X) es impar.
impar(s(X)) :- par(X).
```

3.

```
 % el producto de cualquier natural por 0 es 0.
producto(_, 0, 0).
 % si se suma X al producto de X por Y, el resultado es X por s(Y)
producto(X, s(Y), Z) :- producto(X, Y, ProdXY), suma(ProdXY, X, Z).
```

CONSULTAS

Consultas para la activación de programas

Las consultas = cláusulas de Horn negativas = cláusulas objetivo = cláusulas meta tienen la siguiente sintaxis:

```
?- A1, ... ,An.
```

---

siendo `A1,...,An` *predicados*, y se corresponden con fórmulas lógicas de la forma

$$\exists X_1 \ldots \exists X_q (A1 \wedge \ldots \wedge An)$$

siendo $X_1, \ldots, X_q$, $q \geq 0$, las variables que, en caso de que las haya, aparecen en `A1,...,An`. Representan la pregunta *¿Es cierto que existen objetos* $X_1, \ldots, X_q$ *que cumplen A1 y ... y An? En caso afirmativo, ¿qué objetos son?* ($, = \wedge = y =$ conjunción)

Ejemplos (Algunas consultas al programa “Abuelas/os”)

- `?- abuelo(pepa, pepito).` *False*
- `?- abuelo(pepa, pepita).` *True*
- `?- abuelo(pepa, N).` 2 soluciones: *N=pepita* y *N=pepon*
- `?- abuelo(pepa, _).` *True (variable anónima, no se computa)*
- `?- abuelo(A, pepon).` *A=pepa*
- `?- abuelo(A, N).` 2 soluciones: [*A=pepa,N=pepita*] y [*A=pepa,N=pepon*]
- `?- abuelo(X).` *X=pepa*
- `?- progenitor(A, P), progenitor(P, pepita).`
  A=pepa, P=pepito
- `?- progenitor(X,pepito), progenitor(X,pepon).` *False*
- `?- progenitor(X, pepon), progenitor(X, pepita).`
  X=pepito
- `?- abuelo(A, pepon), progenitor(P, pepon).`
  A=pepa, P=pepito

Observación (Uso de variables anónimas/semianónimas)

Ojo con el uso de la variable **anónima** (_) y las **semianónimas** (___ o cualquiera que empiece por _ seguido de una mayúscula):

- `?- progenitor(pepa, P), progenitor(P, Z).` (v. normal)

  2 soluciones: [`P=pepito`, `Z=pepita`] y [`P=pepito`, `Z=pepon`]

- `?- progenitor(pepa, _P), progenitor(_P, Z).` (v. semi)

  La variable semianónima `_P` no se reporta pero sí tiene que unificar ("casar") consigo misma:

  2 soluciones: `Z=pepita` y `Z=pepon` (`_P = pepito` pero no se incluye)

- `?- progenitor(pepa, _), progenitor(_, Z).` (v. anónima)

  La variable anónima ni reporta valor ni tiene por qué unificar ("casar") en sus distintas apariciones, por lo que, dado que `progenitor(pepa, _)` se cumple para alguien, `Z` será cualquiera para el que se cumpla `progenitor(_, Z)`:

  3 soluciones: `Z=pepito`, `Z=pepita` y `Z=pepon`

Ejemplos (Algunas consultas al programa “suma”)

- `?- suma(0, s(0), s(0)).` *True*
- `?- suma(s(0), s(0), X).` X = s(s(0))
- `?- suma(s(0), s(0), X), suma(X, s(s(0)), Z).`

  X = s(s(0)), Z = s(s(s(s(0))))

- `?- suma(0,0,Z), suma(0,s(s(0)),Z).` *False*
- `?- suma(s(s(0)), Y, s(0)).` *False*
- `?- suma(s(0), _, s(s(s(0)))).` *True*
- `?- suma(s(0), Y, s(s(s(0)))).` Y = s(s(0))

versatilidad de PROLOG: ¡“suma” también sirve para restar!

- `?- suma(X, Y, s(s(0))).`

  3 soluciones: [X = s(s(0)),Y = 0], [X = s(0),Y = s(0)] y [X = 0, Y = s(s(0))]

versatilidad de PROLOG: ¡“suma” descompone en sumandos!

Ejercicios (Consultas al programa “Amistades/Enemistades”)

1. *Exprese en LPO y en lenguaje natural qué se pretende averiguar con las siguientes consultas en* PROLOG*:*
   - `?- amigo(abel, X).`
   - `?- amigo(abel, _).`
   - `?- enemigo(X, X).`
   - `?- amigo(dolores, X), enemigo(X, dolores).`
   - `?- amigo(X,_Y), amigo(X,_Z), enemigo(_Y,_Z).`
2. *Escriba en Lógica de Primer Orden y en* PROLOG *las siguientes preguntas:*
   - *¿Tienen Abel y Caín algún amigo común? En caso afirmativo, ¿quién o quiénes?*
   - *¿Es Caín amigo de alguno de los enemigos de Abel? En caso afirmativo, ¿quién o quiénes son?*
   - *¿Tiene Abel enemigos?*
   - *¿Qué parejas de personas existen tales que cada una de ellas es amiga de la otra?*

Soluciones propuestas:

1.

- LPO: $\exists X\ amigo(abel, X)$. ¿Es Abel amigo de alguien? En caso afirmativo, ¿de quién o quiénes?
- LPO: $\exists \_\ amigo(abel, \_)$. ¿Es Abel amigo de alguien? En este caso el uso de la variable anónima hace que se espere solo una respuesta booleana.
- LPO: $\exists X\ enemigo(X, X)$. ¿Hay alguien que sea enemigo de sí mismo? En caso afirmativo, ¿quién o quiénes?
- LPO: $\exists X\ (amigo(dolores, X) \wedge enemigo(X, dolores))$. ¿Es Dolores amiga de alguno de sus enemigos? En caso afirmativo, ¿de quién o quiénes?
- LPO: $\exists X \exists \_Y \exists \_Z\ (amigo(X, \_Y) \wedge amigo(X, \_Z) \wedge enemigo(\_Y, \_Z))$. ¿Hay alguien que tenga al menos dos amigos que son enemigos entre sí? En caso afirmativo, ¿quién o quiénes? (las variables semianónimas no se computarán).

2.

- LPO: $\exists A\ (amigo(abel, A) \wedge amigo(cain, A))$
  PROLOG: `?- amigo(abel, A), amigo(cain,A).`
- LPO: $\exists X\ (amigo(cain, X) \wedge enemigo(X, abel))$
  PROLOG: `?- amigo(cain, X), enemigo(X, abel).`
- LPO: $\exists \_\ enemigo(\_, abel)$
  PROLOG: `?- enemigo(_, abel).`
- LPO: $\exists A \exists B\ (amigo(A, B) \wedge amigo(B, A))$
  PROLOG: `?- amigo(A,B), amigo(B,A).`

RESUMEN

	REGLAS	HECHOS
PROLOG	`B :- B1,...,Bm.`	`B.`
LPO	$\forall X_1 \ldots \forall X_p((B_1 \wedge \ldots \wedge B_m) \rightarrow B)$	$\forall X_1 \ldots \forall X_p(B)$
Uso	en un programa	en un programa

	OBJETIVOS (METAS)
PROLOG	`?- A1,...,An.`
LPO	$\exists X_1 \ldots \exists X_q(A_1 \wedge \ldots \wedge A_n)$
Uso	consulta

- `B`, `Bi` y `Aj` representan *predicados* y X_k *variables*.
- Hechos, reglas y consultas *terminan siempre con un punto*.
- Predicados: *empiezan por minúscula*.
- Variables: *empiezan por mayúscula o subrayado*.

Interpretación procedural de PROLOG

1. Predicados = *procedimientos*, con argumentos = *parámetros*.
2. Reglas = *definiciones de procedimientos*:

   Una regla

   ```
   B :- B1, B2,..., Bm.
   ```

   se puede interpretar de la siguiente forma: computar el procedimiento `B` equivale a computar en primer lugar el procedimiento `B1`, a continuación el procedimiento `B2`, etc hasta `Bm`.
3. Consultas = *llamadas a procedimientos*:

   Una consulta

   ```
   ?- A1, A2,..., An.
   ```

   se puede interpretar como una llamada al procedimiento `A1`, seguida de una llamada al procedimiento `A2`, etc hasta `An`.

Ejercicios (Sintaxis de PROLOG)

- *Ejercicio nº 1 de la Práctica de PROLOG nº 1 (puede consultar las instrucciones generales para la realización de las prácticas aquí).*
- *Ejercicio nº 2 de la Práctica de PROLOG nº 1.*

Las soluciones propuestas para ambos ejercicios, comentadas, están disponibles en Práctica de PROLOG nº 1 con soluciones.

BIBLIOGRAFÍA

- L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.
- W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fifth edition, 2003.
- I. Bratko.*Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 2001.
- J. Lloyd. *Foundations of Logic Programming*, (Second Edition). Springer-Verlag, 1987.
- R. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
- U. Nilsson and J. Maluszynski. Logic, Programming and Prolog. John Wiley & Sons Ltd, 1996.
- SWI-Prolog, entorno de programación en Prolog de dominio público.
- comp.lang.prolog. Faq

Programación Declarativa

3er Curso, Grado en Ingeniería en Informática
Universidad Rey Juan Carlos

Programación Lógica

Construcción detallada de un Árbol de Resolución

Considere el programa Prolog

```
p(X,X).              % cláusula C1 (hecho)
p(X,Z) :-            % cláusula C2 (regla)
    p(Y,Z),
    q(X,Y).
q(a,b).              % cláusula C3 (hecho)
```

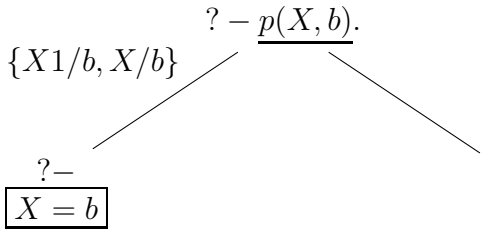
y la consulta `?- p(X,b)`, cuyo objetivo es averiguar, a partir del conocimiento expresado en el programa anterior, si existe algún objeto `X` relacionado con el objeto `b` mediante la relación `p`. En lo que sigue se detalla la construcción, paso a paso, del árbol de Resolución correspondiente a la consulta anterior y se explica cómo, a partir de él, se deduce qué respuesta(s) ofrecería Prolog ante esa consulta, y en qué orden facilitaría dichas respuestas.

Construcción del Árbol de Resolución

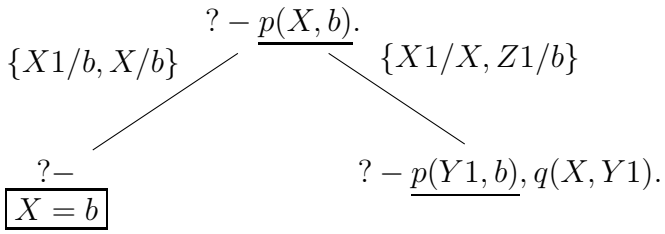
Los árboles de Resolución de Prolog, descritos en el apartado 3.2. del tema PL-2 de los apuntes, se construyen en profundidad por la izquierda y con retroceso. La raíz del árbol no es otra cosa que la consulta, que tendrá tantos hijos como cláusulas haya en el programa con las que se pueda aplicar la Regla de Resolución, es decir, cláusulas cuya cabeza unifique con `p(X,b)` (en general, con el sub-objetivo de más a la izquierda de la consulta). Este objetivo unifica tanto con la cabeza de la cláusula C1 como con la cabeza de la cláusula C2, aunque será necesario renombrar ambas ya que hay conflicto con los nombres de las variables (la variable X aparece tanto en el objetivo como en C1 y en C2). Dado que Prolog desarrolla el árbol en profundidad por la izquierda, en este momento sólo se calcula el nodo correspondiente a la rama de más a la izquierda, la producida por C1 (aunque se deja trazada la otra rama para cuando haya que desarrollarla al retroceder en la construcción del árbol). Aplicando el algoritmo de unificación entre el objetivo a resolver, `p(X,b)`, y la cabeza de C1 renombrada, `p(X1,X1)`, el unificador de máxima generalidad (u.m.g.) resultante es $\{X1/b, X/b\}$, con el que se obtiene la cláusula vacía `?-` como cláusula resolvente. Se ha llegado así a un nodo éxito, por lo que Prolog calcula la solución asociada aplicando a cada una de las variables de la consulta todos los u.m.g.'s de la rama, en orden, empezando desde la raíz. En este caso solo hay una variable, `X`, a la que se aplica el único u.m.g. de su rama:

$$X\{X1/b, X/b\} = b$$

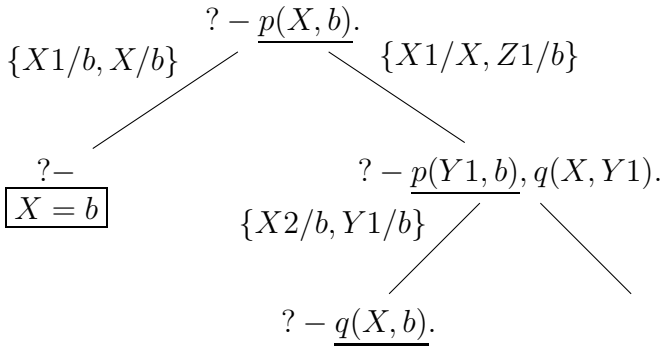
Por lo tanto, el árbol de Resolución es, por el momento, el siguiente:



Dado que se ha terminado de desarrollar la rama de más a la izquierda, Prolog retrocede hasta el siguiente nodo pendiente de cálculo, el segundo hijo de la raíz, que se obtiene aplicando la Regla de Resolución entre el objetivo `?- p(X,b)` y la regla C2 renombrada, `p(X1,Z1) :- p(Y1,Z1), q(X1,Y1)`, mediante el u.m.g. $\{X1/X, Z1/b\}$ (el otro u.m.g. posible sería $\{X/X1, Z1/b\}$, pero da lo mismo elegir uno que otro), que da lugar a la cláusula resolvente `?- p(Y1,b), q(X, Y1)`, completándose así el primer nivel del árbol de Resolución:

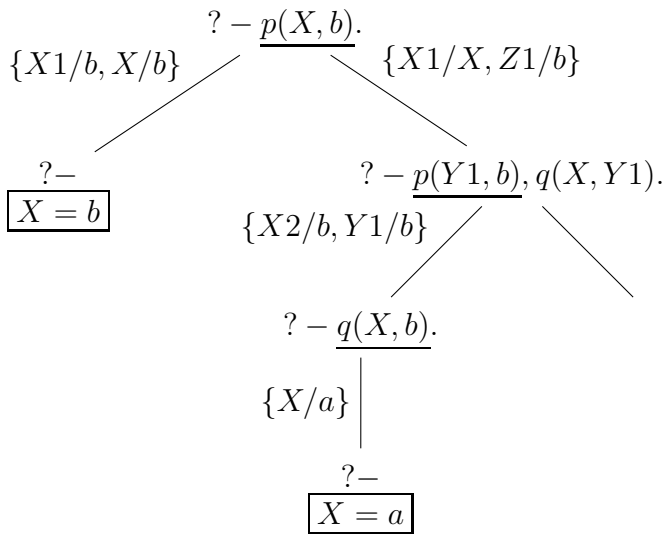


En el siguiente nivel del árbol habrá dos nodos, puesto que el sub-objetivo de más a la izquierda, `p(Y1,b)`, unifica tanto con la cabeza de C1 como con la de C2. Por el momento solo se desarrolla el hijo izquierdo (con C1), que, con u.m.g. $\{X2/b, Y1/b\}$, da lugar a la cláusula resolvente `?- q(X, b)`. Obsérvese que se ha renombrado la cláusula C1 (con subíndice 2 puesto que ahora estamos en el segundo nivel del árbol) debido a que su variable X ya está presente en algunos nodos del árbol.

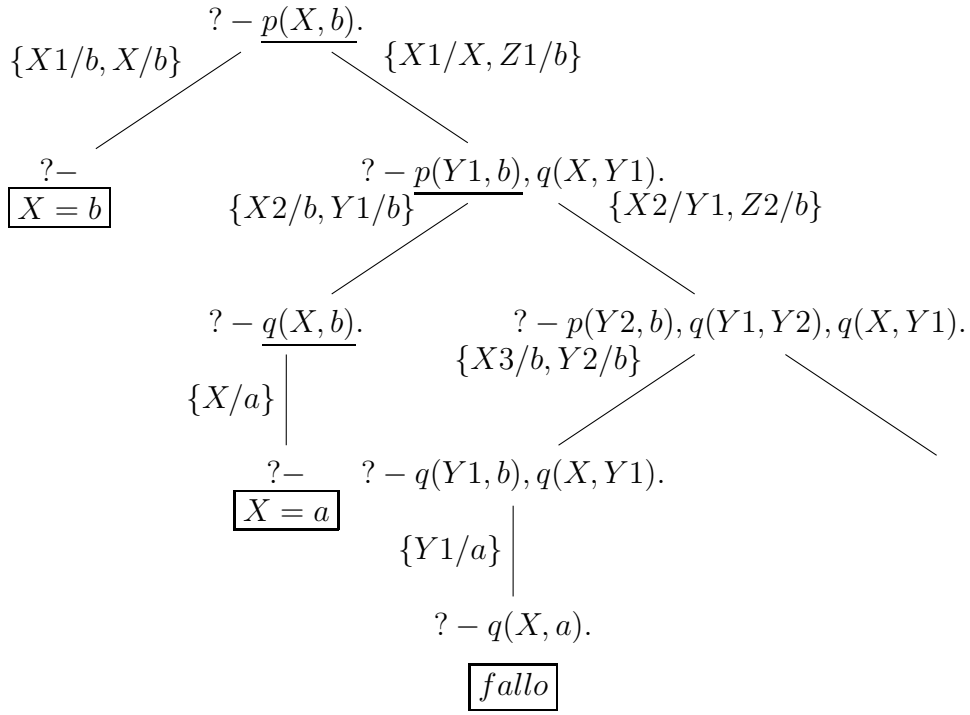


Siguiendo el desarrollo del árbol en profundidad por la izquierda, el nuevo nodo a expandir es `?- q(X,b)`, que tiene un único hijo con C3 mediante el u.m.g. $\{X/a\}$. La cláusula resolvente es la cláusula vacía, por lo que se trata de un nuevo nodo éxito, con solución asociada

$$X\{X1/X, Z1/b\}\ \{X2/b, Y1/b\}\ \{X/a\} = a$$



El siguiente nodo que hay que calcular es el segundo hijo del nodo `?- p(Y1,b), q(X,Y1)`, que, unificando `p(Y1,b)` con la cabeza de la regla C2 (renombrada) mediante el u.m.g. $\{X2/Y1, Z2/b\}$, da lugar a la cláusula resolvente `?- p(Y2,b), q(Y1,Y2), q(X,Y1)`. Esta última, a su vez, tiene dos hijos, obtenidos unificando `p(Y2,b)` con C1 (hijo izquierdo) y con C2 (hijo derecho). Como siempre, Prolog desarrolla primero el hijo izquierdo, obteniendo el resolvente `?- q(Y1,b), q(X,Y1)` mediante el u.m.g. $\{X3/b, Y2/b\}$, y a partir de él el nuevo resolvente `?- q(X,a)` (haciendo la sustitución $\{Y1/a\}$), que falla puesto que no hay ninguna cláusula en el programa que lo resuelva. El árbol es ahora el siguiente:

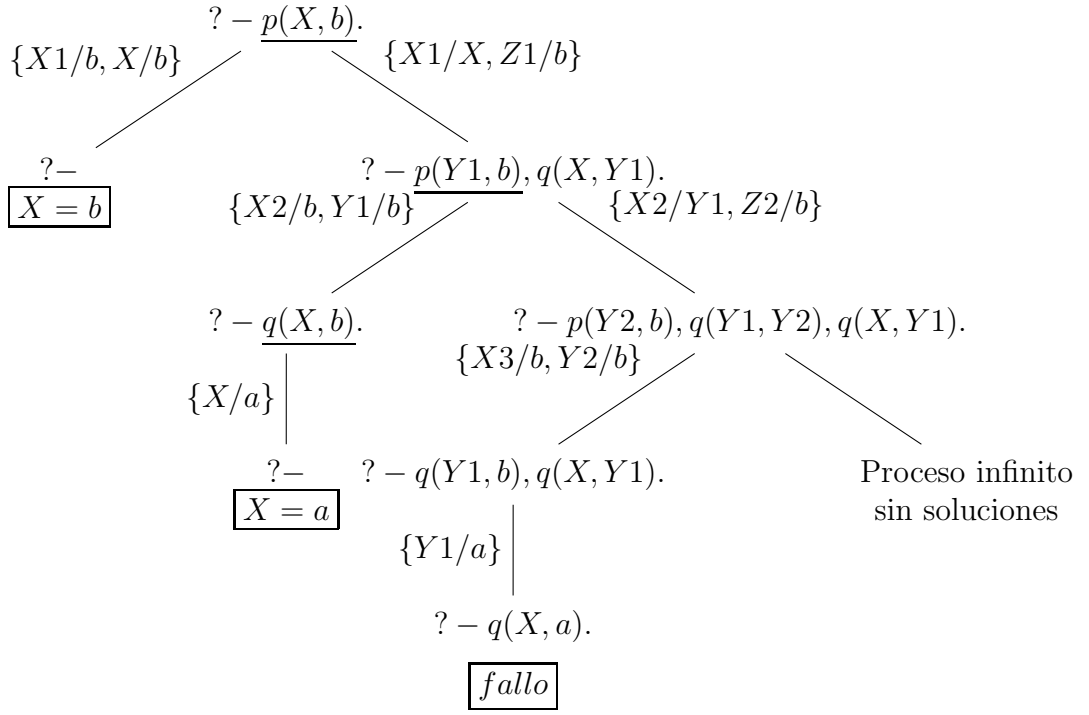


Al encontrarse con un fallo, Prolog retrocede automáticamente hasta el siguiente nodo pendiente de cálculo, que resultará ser `?- p(Y3,b), q(Y2,Y3), q(Y1,Y2), q(X,Y1)`. LLegados a este punto no merece la pena seguir desarrollando el árbol, puesto que se

aprecia que se trata de un proceso infinito. En efecto, el nuevo nodo tendrá dos hijos, de forma que:

- La rama de la izquierda, al cabo de unos pasos, acabará con un nodo fallo. Esto es debido a que los dos últimos sub-objetivos de la consulta, `q(Y1,Y2), q(X,Y1)`, siempre darán lugar a un fallo, ya que `q(Y1,Y2)` solo puede ser cierto mediante la sustitución `Y1/a`, sustitución que necesariamente hace imposible que se cumpla el siguiente sub-objetivo, ya que este pasaría a ser `q(X,a)`.
- La rama de la derecha repetirá el proceso indefinidamente, ya que la regla C2 es recursiva por la izquierda y vuelve a poner en cabeza del objetivo un nuevo predicado `p`.

En definitiva, el árbol de Resolución correspondiente al programa y a la consulta dados es el siguiente:



Respuestas de Prolog

Ante una consulta, Prolog construye el árbol de Resolución en profundidad por la izquierda, con retroceso, y facilita las soluciones según las va encontrando. Por lo tanto, ante la consulta dada, y a la vista del árbol anterior, la respuesta de Prolog será la siguiente:

```
X=b ;                        % primera solución encontrada
X=a ;                        % segunda solución encontrada
ERROR: Out of local stack    % computación infinita
```

PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL2: El lenguaje Prolog, aspectos básicos

3. Semántica

Grado en Ingeniería Informática

URJC

Ana Pradera

Contenido

1. INTRODUCCIÓN
2. UNIFICACIÓN
   - Sustituciones
   - Unificadores
   - El Algoritmo de Unificación
   - Ejercicios de Unificación
   - Unificación en PROLOG
3. LA REGLA DE RESOLUCIÓN
   - Definición
   - Ejemplos y ejercicios
4. EL ÁRBOL DE RESOLUCIÓN
   - Construcción de Árboles de Resolución
   - Mecanismo de cómputo de PROLOG

INTRODUCCIÓN

- En lo que sigue se describe la *semántica operacional* de PROLOG, es decir, su mecanismo de cómputo.
- PROLOG implementa un refinamiento del sistema de Demostración Automática conocido como Sistema de Resolución SLD.
- Para comprender su funcionamiento es necesario conocer:
  - El problema de la Unificación y el algoritmo para resolverlo.
  - La Regla de Resolución, que utiliza el Algoritmo de Unificación.
  - Los Árboles de Resolución, que utilizan la Regla de Resolución.

UNIFICACIÓN

Expresión = término (constante, variable o compuesto) o predicado

El problema de la Unificación

Dadas dos expresiones, el problema de la Unificación consiste en determinar si dichas expresiones pueden resultar sintácticamente idénticas mediante la realización de sustituciones adecuadas en sus variables, en cuyo caso se dice que las expresiones son unificables.

Ejemplos

1. $p(f(X), a)$ y $p(f(g(b)), Y)$ son unificables: basta sustituir X por $g(b)$ e Y por a, obteniéndose la expresión $p(f(g(b)), a)$.
2. $p(f(X), a)$ y $p(f(g(b)), c)$ NO son unificables: nunca podrán resultar iguales, al ser a y c dos constantes distintas.

Sustituciones

- Una sustitución es una función que permite asociar términos a variables.
- Las sustituciones se pueden aplicar a expresiones (términos o predicados), dando lugar a nuevas expresiones denominadas *instancias* de las primeras.
- Las sustituciones se pueden componer entre sí dando lugar a nuevas sustituciones.

Definición (Sustitución)

Una sustitución es un conjunto finito de asociaciones entre variables y términos, denotado $\{X_1/t_1, X_2/t_2, \ldots, X_n/t_n\}$, donde:

- $X_1, \ldots, X_n$ son variables diferentes entre sí.
- $t_1, \ldots, t_n$ son términos y se verifica $t_i \neq X_i$ para todo $i \in \{1, \ldots, n\}$.

Notación

- Cuando $n = 0$ se obtiene la *sustitución vacía*, denotada ϵ o $\{\}$.
- El resto de sustituciones se denotan σ, τ, θ, etc.

Ejemplos

- $\sigma = \{X/a, Y/f(a,b), Z/g(Y)\}$ y $\tau = \{X/f(a,Z), Y/g(f(a,Y))\}$ son sustituciones.
- $\{\mathbf{X}/a, Y/f(a,b), \mathbf{X}/g(Y)\}$ y $\{X/f(a,Z), Y/g(f(a,Y)), \mathbf{Z}/\mathbf{Z}\}$ no son sustituciones.

Definición (Aplicación de una sustitución a una expresión)

Sea φ una expresión y sea $\sigma = \{X_1/t_1, X_2/t_2, \ldots, X_n/t_n\}$ una sustitución. La aplicación de σ a φ se define como la expresión resultante de sustituir en φ, de forma simultánea, cada aparición de la variable X_i, $1 \leq i \leq n$, por su término asociado en la sustitución σ, t_i. La expresión obtenida, denotada $\varphi\sigma$, se denomina una *instancia* de φ.

Ejemplos

- Si $\sigma = \{X/f(X,U), Y/X, Z/h(b,Y), U/a\}$ y $\varphi = g(X,Y,Z)$, entonces $\varphi\sigma = g(f(X,U), X, h(b,Y))$. ($\varphi\sigma \neq g(f(X,a), X, h(b,Y))$ y $\varphi\sigma \neq g(f(X,U), f(X,U), h(b,Y))$ puesto que *no* se encadenan las sustituciones, se hacen de forma simultánea).
- Si $\sigma = \{X/Y, Y/f(a)\}$ y $\varphi = p(X, f(Y), Z)$, entonces $\varphi\sigma = p(Y, f(f(a)), Z)$. ($\varphi\sigma \neq p(f(a), f(f(a)), Z)$ puesto que *no* se encadenan las sustituciones, se hacen de forma simultánea).

Definición (Composición de dos sustituciones)

Sean $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$ y $\tau = \{Y_1/s_1, \ldots, Y_m/s_m\}$ dos sustituciones. La composición de σ con τ, denotada $\sigma\tau$, es la sustitución compuesta por:

1. todos los pares $X_i/(t_i\tau)$, $1 \leq i \leq n$, salvo aquellos tales que $t_i\tau = X_i$ ($t_i\tau$ es la ▸ aplicación de τ a t_i).
2. todos los pares Y_j/s_j, $1 \leq j \leq m$, salvo aquellos tales que $Y_j \in \{X_1, \ldots, X_n\}$.

Ejemplo

Sea $\sigma = \{X/g(U), Y/f(Z), Z/Y\}$ y $\tau = \{U/a, X/b, Y/Z, Z/g(X)\}$.

Entonces:

$$\begin{aligned}\sigma\tau &= \{X/g(a), Y/f(g(X)), \mathbf{Z/Z}, U/a, \mathbf{X/b}, \mathbf{Y/Z}, \mathbf{Z/g(X)}\} \\ &= \{X/g(a), Y/f(g(X)), U/a\}\end{aligned}$$

Ejercicios (Aplicación y composición de sustituciones)

1. *Dadas las sustituciones* $\sigma_1 = \{U/a, X/b, Y/Z, Z/g(X)\}$ *y* $\sigma_2 = \{X/g(U), Y/f(Z), Z/Y, V/U\}$*:*
   - *Calcule la composición* $\sigma_1\sigma_2$*.*
   - *Aplique la sustitución resultante a la expresión* $\varphi = p(X, f(Z), Y, V)$*, es decir, calcule* $\varphi\sigma_1\sigma_2$*.*

2. *Dadas las sustituciones* $\sigma_1 = \{X/Z, Z/g(Y, Y), V/Y, T/X\}$ *y* $\sigma_2 = \{Y/V, Z/g(V, V), T/a\}$*:*
   - *Calcule* $\sigma_1\sigma_2$*.*
   - *Calcule* $\sigma_2\sigma_1$*.*
   - *¿Existe alguna sustitución* θ *tal que* $\sigma_1\theta = \sigma_2$*?*
   - *¿Existe alguna sustitución* θ *tal que* $\sigma_2\theta = \sigma_1$*?*

Soluciones propuestas:

1.

- En el paso 1 de la definición de composición se obtienen los pares $\{U/(a\sigma_2), X/(b\sigma_2), Y/(Z\sigma_2), Z/(g(X)\sigma_2)\} = \{U/a, X/b, Y/Y, Z/g(g(U))\}$, de los que hay que eliminar Y/Y. En el paso 2, de los pares de σ_2 solo se añade V/U, puesto que las demás variables que aparecen a la izquierda en sus pares, X, Y, Z, son variables de σ_1. Se obtiene por lo tanto $\sigma_1\sigma_2 = \{U/a, X/b, Z/g(g(U)), V/U\}$.
- Aplicando esta última sustitución a la expresión φ se obtiene $\varphi\sigma_1\sigma_2 = p(b, f(g(g(U))), Y, U)$.

2.

- $\sigma_1\sigma_2 = \{X/g(V, V), Z/g(V, V), T/X, Y/V\}$.
- $\sigma_2\sigma_1 = \{Z/g(Y, Y), T/a, X/Z, V/Y\}$.
- Sí: $\theta = \{Z/X, Y/V, X/a\}$.
  - El par Z/X es necesario para que el par X/Z de σ_1 desaparezca de $\sigma_1\theta$.
  - El par Y/V es necesario para transformar el par $Z/g(Y, Y)$ de σ_1 en el par $Z/g(V, V)$ de σ_2.
  - El par X/a es necesario para transformar el par T/X de σ_1 en el par T/a de σ_2.
- No, puesto que en el paso 1 de la composición de σ_2 con θ se deberá incluir siempre, sea quien sea θ, el par T/a (puesto que a es una constante y por lo tanto $a\theta = a$ para cualquier θ), y resulta que σ_1 no incluye dicho par.

Unificadores

Definición (Unificador)

Sean φ_1 y φ_2 dos expresiones. Si existe una sustitución σ tal que $\varphi_1\sigma = \varphi_2\sigma$, se dice que φ_1 y φ_2 son unificables y que la sustitución σ es un unificador de φ_1, φ_2.

Expresiones	¿Unifican?
a , b	No
a, $f(X)$	No
X , b	Sí
X , $g(a, Y)$	Sí

Expresiones	¿Unifican?
X , $f(X, a)$	No
$p(a)$, $p(f(Y))$	No
$g(a, b)$, $g(U, U)$	No
$p(X, f(Y), Z)$, $p(g(a), f(b), U)$	Sí

Observación

Cuando dos expresiones resultan ser unificables, es posible que la unificación se pueda conseguir mediante varios unificadores distintos.

Ejemplo

Por ejemplo, las expresiones $\varphi_1 = p(f(X), Y)$ y $\varphi_2 = p(f(a), Z)$ son unificables mediante los siguientes unificadores:

- Con $\sigma = \{X/a, Y/a, Z/a\}$, $\varphi_1\sigma = \varphi_2\sigma = p(f(a), a)$.
- Con $\sigma = \{X/a, Y/b, Z/b\}$, $\varphi_1\sigma = \varphi_2\sigma = p(f(a), b)$.
- Con $\sigma = \{X/a, Y/Z\}$, $\varphi_1\sigma = \varphi_2\sigma = p(f(a), Z)$.
- Con $\sigma = \{X/a, Z/Y\}$, $\varphi_1\sigma = \varphi_2\sigma = p(f(a), Y)$.
- ...

¿Cuál de ellos elegir? Uno que sea de máxima generalidad (si hay varios, cualquiera de ellos).

Definición (Unificador de máxima generalidad, u.m.g.)

Sean φ_1, φ_2 dos expresiones unificables y sea σ un unificador de $\{\varphi_1, \varphi_2\}$. Se dice que σ es un unificador de máxima generalidad (u.m.g.) si es *más general* que cualquier otro unificador de $\{\varphi_1, \varphi_2\}$, es decir, si para cualquier otro unificador τ resulta que existe una sustitución θ tal que $\tau = \sigma\theta$.

Ejemplo

Considere el ejemplo anterior:

- Tanto $\{X/a, Y/Z\}$ como $\{X/a, Z/Y\}$ resultan ser unificadores de máxima generalidad (intuitivamente: los más sencillos).
- Los unificadores $\{X/a, Y/a, Z/a\}$ y $\{X/a, Y/b, Z/b\}$ no puede ser u.m.g.'s puesto que no son más generales, por ejemplo, que $\{X/a, Y/Z\}$.

El Algoritmo de Unificación

- El problema de la unificación tiene por tanto un doble objetivo:
  1. Dado un par de expresiones, decidir si son o no unificables.
  2. En caso afirmativo, encontrar un unificador de máxima generalidad, u.m.g. (si hay varios, cualquiera de ellos).
- Existen distintos algoritmos capaces de resolver este problema. A continuación se presenta uno de los más comunes, basado en las ideas originales de Robinson de 1965.
- Se trata de un proceso iterativo que localiza e intenta resolver, mediante la aplicación de sustituciones adecuadas, las discordancias existentes entre las dos expresiones de entrada.
- El funcionamiento del algoritmo se ilustra a continuación con varios ejemplos, utilizando para ello una *tabla* en la que se resume la información relevante en cada uno de los pasos del algoritmo.

ALGORITMO DE UNIFICACIÓN

Entrada: Dos expresiones φ_1 y φ_2.

Salida:

1. Un valor booleano que indica si las expresiones son unificables.
2. En caso afirmativo, un u.m.g. para φ_1, φ_2.

Descripción:

PASO 1: *unificables* := cierto ; $\sigma := \{\}$

PASO 2: MIENTRAS *unificables* = cierto y $\varphi_1\sigma \neq \varphi_2\sigma$ HACER:

2.1. Encontrar el símbolo más a la izqda de $\varphi_1\sigma$ tal que el símbolo correspondiente de $\varphi_2\sigma$ sea distinto (en rojo en los ejemplos).

2.2. Sean s_1 y s_2 los subtérminos de $\varphi_1\sigma, \varphi_2\sigma$ que empiezan por los símbolos elegidos en el paso anterior (subrayados en los ejemplos):

(a) Si ninguno de los dos es una variable o uno es una variable que aparece en el otro (esto último se denomina test de ocurrencia):

unificables := falso.

(b) En otro caso:

- si s_1 es una variable: $\sigma := \sigma\{s_1/s_2\}$
- si s_1 no es una variable: $\sigma := \sigma\{s_2/s_1\}$
- actualizar las expresiones $\varphi_1\sigma$ y $\varphi_2\sigma$, para lo que basta aplicar $\{s_1/s_2\}$ (o $\{s_2/s_1\}$) a las expresiones de la iteración anterior.

(Nota: $\sigma\{s_1/s_2\}$ y $\sigma\{s_2/s_1\}$ son ▸ composiciones de sustituciones)

PASO 3:

DEVOLVER *unificables*.

SI *unificables* = cierto, σ es un u.m.g. de φ_1, φ_2.

Ejemplo

$\varphi_1\sigma$	$\varphi_2\sigma$	s_1	s_2	*unif.*	σ
$p(\underline{X}, X)$	$p(\underline{f(U)}, f(V))$	X	$f(U)$	cierto	$\{X/f(U)\}$
$p(f(U), f(\underline{U}))$	$p(f(U), f(\underline{V}))$	U	V	cierto	$\{X/f(V), U/V\}$
$p(f(V), f(V))$	$p(f(V), f(V))$	FIN			

El algoritmo termina en ese punto porque se llega a $\varphi_1\sigma = \varphi_2\sigma$, concluyendo que las dos expresiones de entrada son unificables con un u.m.g. dado por $\sigma = \{X/f(V), U/V\}$. Observe cómo para obtener el u.m.g. $\{X/f(V), U/V\}$ se ha realizado la *composición* de $\{X/f(U)\}$ con $\{U/V\}$. Otro posible u.m.g. sería $\{X/f(U), V/U\}$ si en la segunda línea se hubiese elegido $\{V/U\}$ en lugar de $\{U/V\}$.

Ejemplo

$\varphi_1\sigma$	$\varphi_2\sigma$	s_1	s_2	*unificables*	σ
$p(\underline{X}, f(Y))$	$p(\underline{Z}, a)$	X	Z	cierto	$\{X/Z\}$
$p(Z, \underline{f(Y)})$	$p(Z, \underline{a})$	$f(Y)$	a	falso	FIN

El Paso 2 termina en el punto **2.2. (a)**, puesto que ninguno de los dos subtérminos s_1 y s_2 es una variable, y concluye que las expresiones de entrada no son unificables.

Ejemplo

$\varphi_1\sigma$	$\varphi_2\sigma$	s_1	s_2	*unif.*	σ
$p(\underline{X}, f(X), X)$	$p(\underline{U}, W, W)$	X	U	cierto	$\{X/U\}$
$p(U, \underline{f(U)}, U)$	$p(U, \underline{W}, W)$	$f(U)$	W	cierto	$\{X/U, W/f(U)\}$
$p(U, f(U), \underline{U})$	$p(U, f(U), \underline{f(U)})$	U	$f(U)$	falso	FIN

El Paso 2 termina en el punto **2.2. (a)**, puesto que aunque el subtérmino s_1 es una variable, s_2 contiene a esa variable, por lo que falla el test de ocurrencia y se concluye que las expresiones de entrada no son unificables.

Ejercicios (Aplicación del Algoritmo de Unificación)

Para cada uno de los siguientes pares de expresiones, aplique el Algoritmo de Unificación para averiguar si las expresiones son unificables y, en caso afirmativo, facilitar un unificador de máxima generalidad (u.m.g.).

1. $p(W, X, f(g(Y)))$ *y* $p(Z, f(Z), f(U))$.
2. $p(a, X, f(g(Y)))$ *y* $p(Z, h(Z, U), f(U))$.
3. $q(h(X, a), X)$ *y* $q(Y, f(a, c))$.
4. $p(a, W, X, f(f(X)))$ *y* $p(Z, g(Y), g(Z), f(Y))$.
5. $q(X, g(Y), a)$ *y* $q(c, Z, Z)$.
6. $p(f(g(Y)), Y, Z)$ *y* $p(f(U), a, U)$.
7. $q(f(g(Z, Y)), Y, Z)$ *y* $q(f(U), b, U)$.

(los dos últimos ejercicios están resueltos en el vídeo Ejemplos Algoritmo de Unificación).

Soluciones propuestas:

1. Las expresiones $p(W, X, f(g(Y)))$ y $p(Z, f(Z), f(U))$ son unificables, como muestra la tabla con la aplicación del Algoritmo de Unificación resumida que se incluye a continuación. Recuerde que en el paso de una fila de la tabla a la siguiente la sustitución s_1/s_2 (o s_2/s_1) se aplica a las dos expresiones, φ_1 y φ_2, *completas*, es decir, reemplazando *todas* las posibles apariciones de s_1 por s_2, tanto en φ_1 como en φ_2. Es el caso en el paso de la fila 1 a la fila 2 de la tabla siguiente, donde se reemplazan *todas* las apariciones de Z por W. El u.m.g. obtenido es $\{Z/W, X/f(W), U/g(Y)\}$, aunque otro u.m.g. posible sería $\{W/Z, X/f(Z), U/g(Y)\}$, dado que la primera discordancia, que involucra a dos variables, se podría resolver tanto con Z/W como con W/Z.

$\varphi_1\sigma$	$\varphi_2\sigma$	s_1	s_2	*unif.*	σ
$p(\underline{W}, X, f(g(Y)))$	$p(\underline{Z}, f(Z), f(U))$	W	Z	cierto	$\{Z/W\}$
$p(W, \underline{X}, f(g(Y)))$	$p(W, \underline{f(W)}, f(U))$	X	$f(W)$	cierto	$\{Z/W, X/f(W)\}$
$p(W, f(W), f(\underline{g(Y)}))$	$p(W, f(W), f(\underline{U}))$	$g(Y)$	U	cierto	$\{Z/W, X/f(W), U/g(Y)\}$
$p(W, f(W), f(g(Y)))$	$p(W, f(W), f(g(Y)))$	FIN			

2. Las expresiones $p(a, X, f(g(Y)))$ y $p(Z, h(Z, U), f(U))$ son unificables con u.m.g. $\{Z/a, X/h(a, g(Y)), U/g(Y)\}$. La tabla resumiendo los pasos del algoritmo en este caso se incluye a continuación. Recuerde que las sustituciones de la última columna de la tabla se obtienen *componiendo* la sustitución de la fila anterior con el nuevo par $\{s_1/s_2\}$ (o $\{s_2/s_1\}$), y que dicha composición no siempre es igual a una unión. En particular, el u.m.g. de la tercera fila es la *composición* de $\{Z/a, X/h(a, U)\}$ con $\{U/g(Y)\}$, que es igual a $\{Z/a, X/h(a, g(Y)), U/g(Y)\}$, conjunto distinto a la unión.

$\varphi_1\sigma$	$\varphi_2\sigma$	s_1	s_2	*unif.*	σ
$p(\underline{a}, X, f(g(Y)))$	$p(\underline{Z}, h(Z, U), f(U))$	a	Z	cierto	$\{Z/a\}$
$p(a, \underline{X}, f(g(Y)))$	$p(a, \underline{h(a, U)}, f(U))$	X	$h(a, U)$	cierto	$\{Z/a, X/h(a, U)\}$
$p(a, h(a, U), f(\underline{g(Y)}))$	$p(a, h(a, U), f(\underline{U}))$	$g(Y)$	U	cierto	$\{Z/a, X/h(a, g(Y)), U/g(Y)\}$
$p(a, h(a, g(Y)), f(g(Y)))$	$p(a, h(a, g(Y)), f(g(Y)))$	FIN			

3. Las expresiones $q(h(X, a), X)$ y $q(Y, f(a, c))$ son unificables con u.m.g. $\{Y/h(f(a, c), a), X/f(a, c)\}$ (observe la *composición* en el segundo paso):

$\varphi_1\sigma$	$\varphi_2\sigma$	s_1	s_2	*unif.*	σ
$q(\underline{h(X, a)}, X)$	$q(\underline{Y}, f(a, c))$	$h(X, a)$	Y	cierto	$\{Y/h(X, a)\}$
$q(h(X, a), \underline{X})$	$q(h(X, a), \underline{f(a, c)})$	X	$f(a, c)$	cierto	$\{Y/h(f(a, c), a), X/f(a, c)\}$
$q(h(f(a, c), a), f(a, c))$	$q(h(f(a, c), a), f(a, c))$	FIN			

4. Las expresiones $p(a, W, X, f(f(X)))$ y $p(Z, g(Y), g(Z), f(Y))$ son unificables con u.m.g. $\{Z/a, W/g(f(g(a))), X/g(a), Y/f(g(a))\}$. Note cómo en el paso de la fila 1 a la fila 2 de la tabla correspondiente a este ejemplo que se incluye a continuación, se reemplazan *todas* las apariciones de Z por a, no solo la ocurrencia donde se encontró la discordancia.

$\varphi_1\sigma$	$\varphi_2\sigma$	s_1	s_2	*unif.*	σ
$p(\underline{a}, W, X, f(f(X)))$	$p(\underline{Z}, g(Y), g(Z), f(Y))$	a	Z	cierto	$\{Z/a\}$
$p(a, \underline{W}, X, f(f(X)))$	$p(a, \underline{g(Y)}, g(a), f(Y))$	W	$g(Y)$	cierto	$\{Z/a, W/g(Y)\}$
$p(a, g(Y), \underline{X}, f(f(X)))$	$p(a, g(Y), \underline{g(a)}, f(Y))$	X	$g(a)$	cierto	$\{Z/a, W/g(Y), X/g(a)\}$
$p(a, g(Y), g(a), f(\underline{f(g(a))}))$	$p(a, g(Y), g(a), f(\underline{Y}))$	$f(g(a))$	Y	cierto	$\{Z/a, W/g(f(g(a))),$ $X/g(a), Y/f(g(a))\}$
$p(a, g(Y), g(a), f(f(g(a))))$	$p(a, g(Y), g(a), f(f(g(a))))$	FIN			

5. A simple vista se puede comprobar que las expresiones $q(X, g(Y), a)$ y $q(c, Z, Z)$ no son unificables, puesto que la variable Z no puede unificar a la vez con dos términos no unificables entre sí como son $g(Y)$ y a. La tabla correspondiente sería la siguiente:

$\varphi_1\sigma$	$\varphi_2\sigma$	s_1	s_2	*unif.*	σ
$q(\underline{X}, g(Y), a)$	$q(\underline{c}, Z, Z)$	X	c	cierto	$\{X/c\}$
$q(c, \underline{g(Y)}, a)$	$q(c, \underline{Z}, Z)$	$g(Y)$	Z	cierto	$\{X/c, Z/g(Y)\}$
$q(c, g(Y), \underline{a})$	$q(c, g(Y), \underline{g(Y)})$	a	$g(Y)$	falso	FIN

Unificación en PROLOG

PROLOG implementa el Algoritmo de Unificación anterior omitiendo el test de ocurrencia (occur-check) por razones de eficiencia. El lenguaje usa internamente este algoritmo (para aplicar la Regla de Resolución, ver siguiente apartado) pero también lo ofrece para uso del programador:

- El predicado predefinido = *(predicado de unificación)*, se usa en notación infija (`arg1 = arg2`) y:
  - Si las dos expresiones que se le pasan resultan ser, *omitiendo el test de ocurrencia*, unificables, devuelve cierto y realiza las sustituciones de variables indicadas por el u.m.g. calculado por el Algoritmo de Unificación.
  - En caso contrario falla (devuelve *false*).
- El predicado predefinido `\=` *(predicado de no unificación)* también se usa en notación infija (`arg1 \= arg2`) y devuelve cierto (*true*) si el predicado = falla, y falla (devuelve *false*) en caso contrario.

Ejemplo (Unificación en PROLOG)

```
?- f(X, g(X,c)) = f(h(U), Z).
        X = h(U),     Z = g(h(U), c).
?- f(X, g(X,c)) \= f(h(U), Z).
        false
?- p(f(X), g(Y)) = p(U, f(U)).
        false
?- p(f(X), g(Y)) \= p(U, f(U)).
        true
?- X = f(X).
        X = f(X)
% el predicado = no realiza el test de ocurrencia
?- unify_with_occurs_check(X, f(X)).
        false
% este predicado sí realiza el test de ocurrencia
```

Observación (Unificación de variables (semi)anónimas)

La variable anónima _ y las semianónimas (___ más todas las que empiezan por subrayado seguido de letra mayúscula) presentan las siguientes peculiaridades respecto a las variables normales:

	Normales	Semianónimas	Anónima
¿Tienen que unificar?	sí: $p(U, U) = p(a, b)$ da *false*	sí: $p(\_U, \_U) = p(a, b)$ da *false*	no: $p(\_, \_) = p(a, b)$ da *true*
¿Reportan valor?	sí: $p(U, b) = p(a, b)$ da $U = a$	no: $p(\_U, b) = p(a, b)$ da *true*	no: $p(\_, b) = p(a, b)$ da *true*

Ejercicios (El Algoritmo de Unificación de PROLOG)

- *Ejercicio nº 3 de la Práctica de PROLOG nº 1 (soluciones comentadas en Práctica de PROLOG nº 1 con soluciones).*

LA REGLA DE RESOLUCIÓN

Definición (Regla de Resolución (1/3))

La Regla de Resolución de PROLOG recibe como entrada:

- Una cláusula objetivo (consulta) $? - A_1, A_2, \ldots, A_n$, $n \geq 1$.
- Una cláusula de programa, que podrá ser un hecho $B.$ o bien una regla $B :- B_1, \ldots, B_m.$ con $m \geq 1$.

1. Si las cláusulas de entrada tienen variables en común, se deben hacer los renombramientos de variables oportunos (por convenio, en la cláusula de programa) para evitar esta colisión de variables.
2. Si el ▸ Algoritmo de Unificación determina que el *primer* subobjetivo de la cláusula objetivo, A_1, es unificable, mediante un cierto u.m.g. σ, con la cabeza B de la cláusula de programa, entonces la Regla de Resolución se puede aplicar y se llama cláusula resolvente a la cláusula objetivo obtenida como sigue:

Definición (Regla de Resolución (2/3))

- Si la cláusula de programa es un hecho $B.$, la cláusula resolvente es

$$? - A_2\sigma, \ldots, A_n\sigma$$

El subobjetivo A_1 desaparece porque se ha unificado con el hecho B mediante σ. Gráficamente:

$$? - \underline{A_1}, A_2, \ldots, A_n.$$

$$\underline{B}. \qquad \sigma = \{\ldots, \ldots\}$$

$$? - A_2\sigma, \ldots, A_n\sigma.$$

Definición (Regla de Resolución (3/3))

- Si la cláusula de programa es una regla $B :- B_1, \ldots, B_m.$, la cláusula resolvente es

$$? - \underbrace{B_1\sigma, \ldots, B_m\sigma}_{\text{cuerpo de la regla}}, \quad \underbrace{A_2\sigma, \ldots, A_n\sigma}_{\text{resto de subobjetivos}}$$

El subobjetivo A_1 se reemplaza por $B_1\sigma, \ldots, B_m\sigma$ puesto que se ha unificado con B mediante σ. Gráficamente:

$$? - \underline{A_1}, A_2, \ldots, A_n.$$

$$\underline{B} :- B_1, \ldots, B_m. \qquad \sigma = \{\ldots, \ldots\}$$

$$? - B_1\sigma, \ldots, B_m\sigma, A_2\sigma, \ldots, A_n\sigma.$$

Observación

En las cláusulas resolventes:

- $A_i\sigma/B_i\sigma$ son el resultado de ▸ aplicar la sustitución σ a los predicados A_i/B_i.
- Se debe respetar el orden en el que aparecen los predicados en la cláusula resolvente (primero los $B_i\sigma$, en el orden en el que aparecen en la regla, y luego los $A_i\sigma$, en el orden en el que estaban en la cláusula objetivo original).
- Al aplicar la Regla de Resolución entre un objetivo *unitario* "$? - A_1$." y un hecho "B." unificable con A_1, la cláusula resolvente obtenida es $?-$, que se conoce como la cláusula vacía.

Ejemplo (Regla de Resolución)

Dada la cláusula objetivo `?- amigo(abel,X), enemigo(X,Y).`

- La Regla de Resolución no es aplicable con el hecho `enemigo(abel, cain)` puesto que `amigo(abel, X)` y `enemigo(abel, cain)` no son unificables (¡ni siquiera tienen el mismo nombre de predicado!).
- Tampoco con la regla `amigo(abilio,X1):-amigo(abel,X1)`, puesto que `amigo(abel, X)` y `amigo(abilio, X1)` no son unificables (tienen dos constantes distintas como 1er argumento).
- Sí lo sería con `amigo(X1,Y1) :- enemigo(X1,Z1), enemigo(Z1,Y1)` (con renombramiento de variables debido a la coincidencia de variables entre las dos cláusulas): `amigo(abel, X)` y `amigo(X1, Y1)` sí son unificables, con u.m.g. $\sigma = \{X1/abel, X/Y1\}$, y la cláusula resolvente sería `?-enemigo(abel,Z1),enemigo(Z1,Y1),enemigo(Y1,Y).`

Ejemplo (Regla de Resolución)

¿Es posible aplicar la Regla de Resolución entre el objetivo `?-q(X,a),p(X),q(b,Y)` y la regla `q(f(X),Y) :- r(X),r(Y)`?

1. Como las dos cláusulas dadas tienen variables en común, se renombran las variables de la regla, que pasa a ser `q(f(X1), Y1) :- r(X1),r(Y1).`
2. la Regla de Resolución sí es aplicable entre las cláusulas dadas puesto que el Algoritmo de Unificación muestra que el primer predicado del objetivo, `q(X,a)`, es unificable con la cabeza de la regla, `q(f(X1), Y1)`, mediante u.m.g. $\sigma = \{X/f(X1), Y1/a\}$. La cláusula resolvente correspondiente es entonces:

$$\begin{aligned} & ?- \; r(X1)\sigma, r(Y1)\sigma, p(X)\sigma, q(b, Y)\sigma \\ = \; & ?- \; r(X1), r(a), p(f(X1)), q(b, Y). \end{aligned}$$

Ejercicios (Aplicación de la Regla de Resolución)

Para cada uno de los siguientes pares de cláusulas, razone si es posible aplicar la Regla de Resolución, y, en caso afirmativo, calcule la cláusula resolvente asociada:

1. *Objetivo* $?- r(Z,a), q(a,f(a))$. *y regla* $q(X,X) :- p(X), r(X,a)$.
2. *Objetivo* $?- q(a,f(a)), r(Z,a)$. *y regla* $q(X,X) :- p(X), r(X,a)$.
3. *Objetivo* $?- q(X,a), p(X), q(b,Y)$. *y hecho* $q(X,X)$.
4. *Objetivo* $?- r(f(X),X), q(X,Z), r(f(Z),f(f(a)))$. *y regla* $r(X,f(Z)) :- p(X,Z), q(Z,X)$.
5. *Objetivo* $?- p(X,b)$. *y hecho* $p(h(a,X),X)$.

Soluciones propuestas:

1. No es posible aplicar la Regla de Resolución, ya que los predicados `r(Z,a)` y `q(X,X)` no son unificables (ni siquiera tienen mismo nombre de predicado).
2. No es posible aplicar la Regla de Resolución, ya que los predicados `q(a,f(a))` y `q(X,X)` no son unificables (la variable `X` no puede unificar al mismo tiempo con dos términos, `a` y `f(a)`, que no son unificables entre sí).
3. Sí es posible aplicar la Regla de Resolución: renombrando el hecho a `q(X1,X1).`, mediante u.m.g. $\{X1/a, X/a\}$ se obtiene la cláusula resolvente

   ```
   ?- p(a), q(b,Y).
   ```
4. Sí es posible aplicar la Regla de Resolución, pero es necesario renombrar la regla a

   ```
   r(X1,f(Z1)) :- p(X1,Z1), q(Z1,X1).
   ```

   porque en caso contrario el test de ocurrencia del Algoritmo de Unificación impediría la unificación. El u.m.g. es $\{X1/f(f(Z1)), X/f(Z1)\}$ (ojo a la composición entre $\{X1/f(X)\}$ y $\{X/f(Z1)\}$) y se obtiene la cláusula resolvente

   ```
   ?- p(f(f(Z1)),Z1), q(Z1,f(f(Z1))), q(f(Z1),Z),
   r(f(Z),f(f(a))).
   ```
5. Sí es posible aplicar la Regla de Resolución: renombrando el hecho a `p(h(a,X1),X1)` (si no, el test de ocurrencia impediría la unificación) y mediante u.m.g. $\{X/h(a,b), X1/b\}$ (ojo a la composición) se obtiene como cláusula resolvente la cláusula vacía

   ```
   ?-
   ```

EL ÁRBOL DE RESOLUCIÓN

Dados:

- Un *programa* (conjunto de fórmulas -cláusulas de Horn positivas- que pueden ser hechos o reglas),
- Y una *consulta* (fórmula denominada cláusula de Horn negativa),

el Árbol de Resolución asociado a ese programa y esa consulta

- Es una herramienta para *explorar y organizar todas las posibles formas en las que es posible resolver la consulta*, es decir, los posibles caminos (si los hay) para demostrar que la consulta es una consecuencia lógica del programa y, en su caso, *computar* las soluciones asociadas.
- Se construye aplicando reiteradamente la *Regla de Resolución* como se describe a continuación.

Construcción de un Árbol de Resolución (1/2)

El Árbol de Resolución correspondiente a un programa y una consulta es un árbol etiquetado y con raíz:

- Raíz del árbol: la consulta.
- Nodos: son cláusulas objetivo de la forma "`?- O1,..., Ok.`", con $k \geq 0$, que se expanden de la siguiente forma:
  - Si $k = 0$ (cláusula vacía), el nodo no se puede expandir, es una hoja del árbol denominada nodo éxito.
  - Si $k \neq 0$ pero no existe ninguna cláusula en el programa con la que se pueda aplicar la ▸ Regla de Resolución, el nodo tampoco se puede expandir, es una hoja del árbol denominada nodo fallo.
  - En cualquier otro caso, el nodo tendrá un nodo hijo por cada cláusula del programa con la que se pueda aplicar la ▸ Regla de Resolución, de forma que los hijos contienen las *cláusulas resolventes* obtenidas al aplicar la Regla de Resolución.

Construcción de un Árbol de Resolución (2/2)

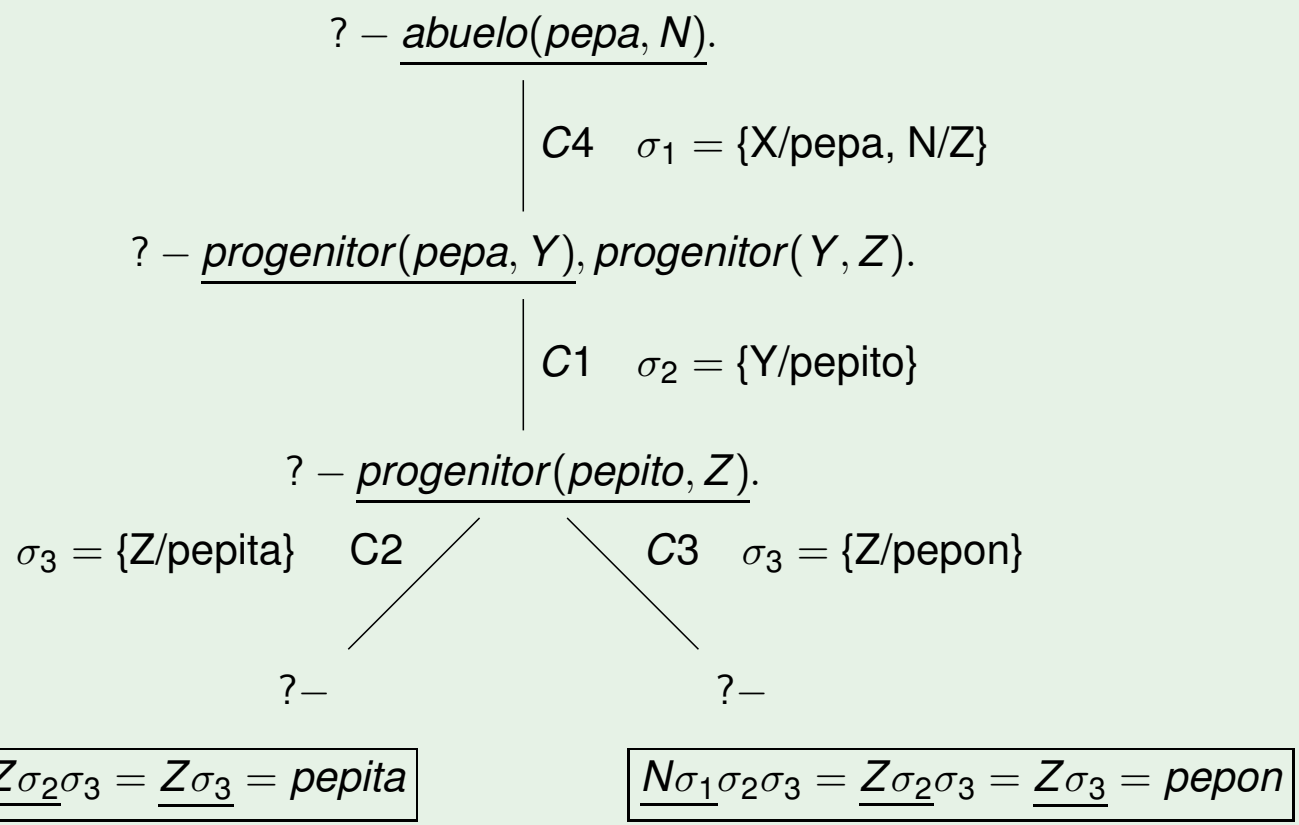
- Los hijos de los nodos que los tengan se colocan de izquierda a derecha eligiendo las cláusulas (hechos o reglas) del programa *de acuerdo con el orden en el que aparecen en él* (el hijo de más a la izquierda será la cláusula resolvente obtenida con la primera cláusula del programa con la que se pueda aplicar la Regla de Resolución, etc).
- Las aristas se etiquetan con la siguiente información:
  - El u.m.g. σ_j utilizado al aplicar la Regla de Resolución. Convenio: los u.m.g.'s se numeran en función del nivel del árbol (σ_1 en el primer nivel, etc).
  - Opcionalmente, la cláusula C_i (C_i^{ren} si se ha renombrado) del programa con la que se aplica la Regla de Resolución. Convenios: (1) la primera cláusula del programa es C_1, la segunda C_2, etc. (2) en caso necesario, las variables se renombran añadiéndoles un subíndice que indica el nivel del árbol (X_1 en el primer nivel, etc).

Interpretación de las hojas de un Árbol de Resolución

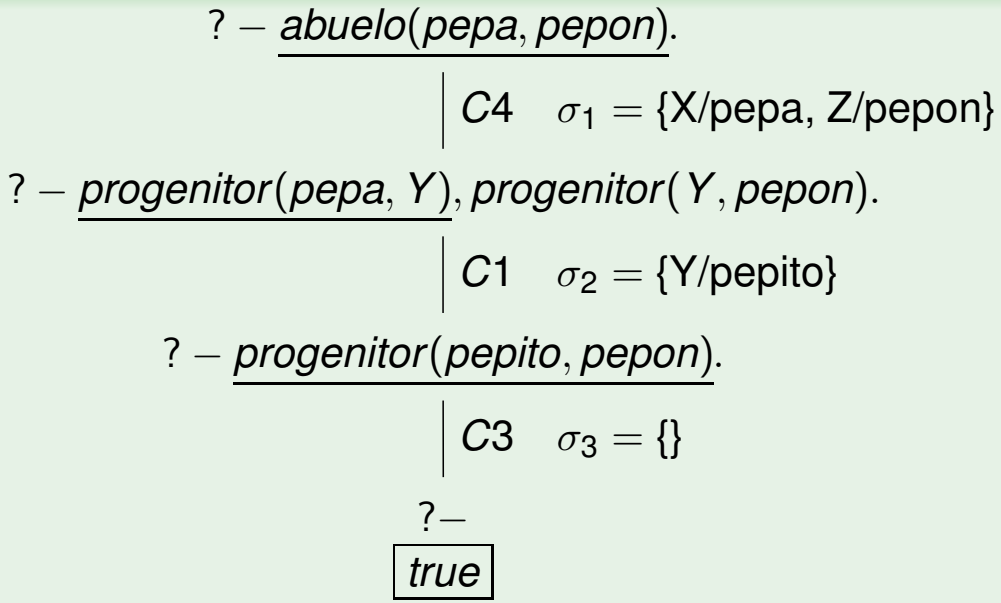
- Nodos éxito
  - Prueban que la consulta se ha podido realizar con éxito (se ha probado que es una consecuencia lógica del programa).
  - Cada nodo éxito proporciona una solución a la consulta:
    - Si la consulta no tiene variables: true.
    - En otro caso: la solución está compuesta por el valor $X\sigma_1 \ldots \sigma_r$ para cada variable X (salvo anónimas o semianónimas) que aparezca en la consulta, donde $\sigma_1, \ldots, \sigma_r$ son los u.m.g.'s asociados con cada una de las aristas de la rama, en orden, empezando desde la raíz del árbol. La forma más fácil de computar $X\sigma_1 \ldots \sigma_r$ es ▸ aplicar σ_1 a X, a continuación ▸ aplicar σ_2 a la expresión resultante de la aplicación anterior, etc.
- Nodos fallo
  - Son fracasos a la hora de intentar probar la consulta.
  - Su aparición no significa que la consulta no se pueda realizar con éxito, simplemente muestra que la rama elegida para intentarlo no lo permite.

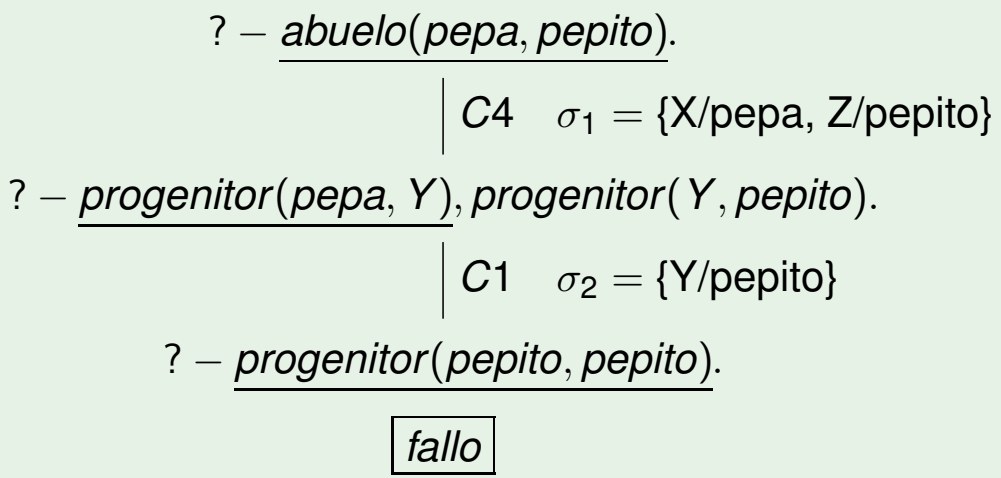
Ejemplo (Árboles de Resolución programa "Abuelas/os" 1/2)

```
progenitor(pepa, pepito).                          % C1
progenitor(pepito, pepita).                        % C2
progenitor(pepito, pepon).                         % C3
abuelo(X,Z) :- progenitor(X,Y), progenitor(Y,Z).   % C4
```



Ejemplo (Árboles de Resolución programa “Abuelas/os” 2/2)



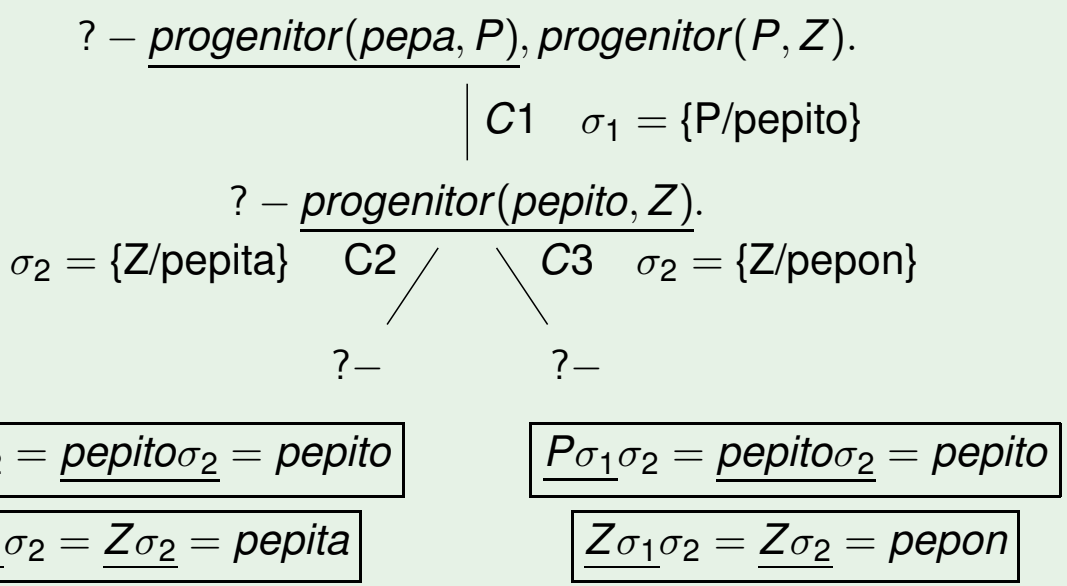


Observación (Árboles con variables anónimas/semianónimas)

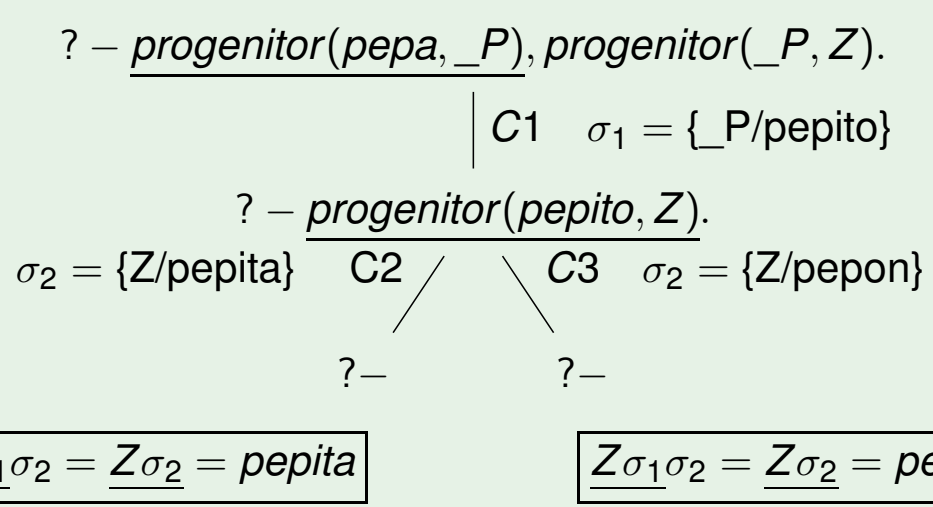
Al construir Árboles de Resolución en los que aparezcan variables anónimas o semianónimas se deben tener en cuenta las particularidades de este tipo de variables en la unificación. Compare los tres árboles a continuación.

Ejemplo (Árboles con variables anónimas/semianónimas (1/2))

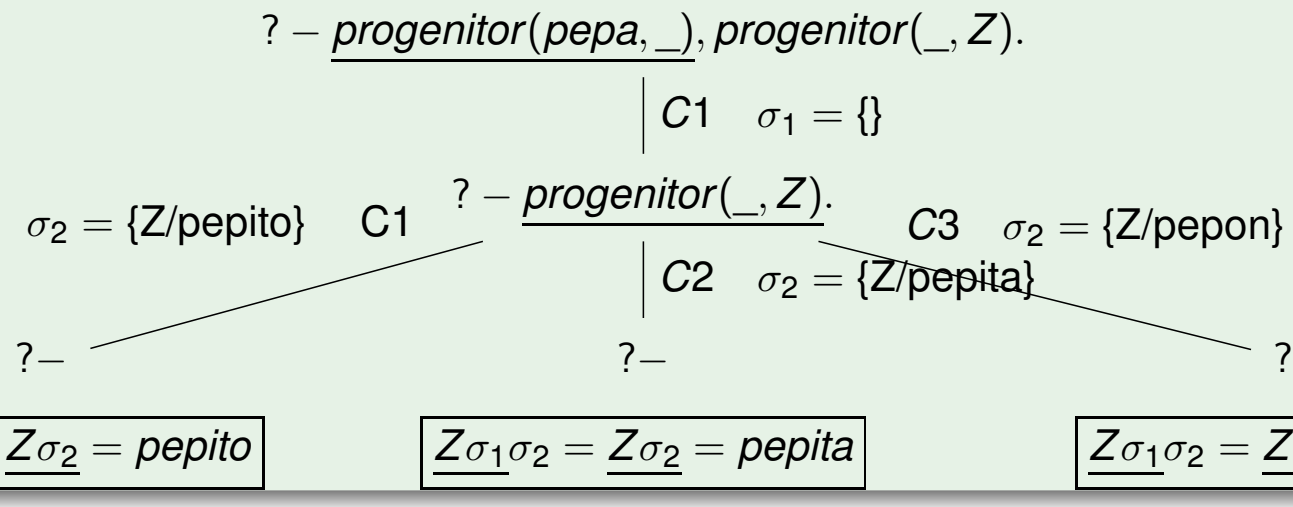
P = variable normal



Ejemplo (Árboles con variables anónimas/semianónimas (2/2))



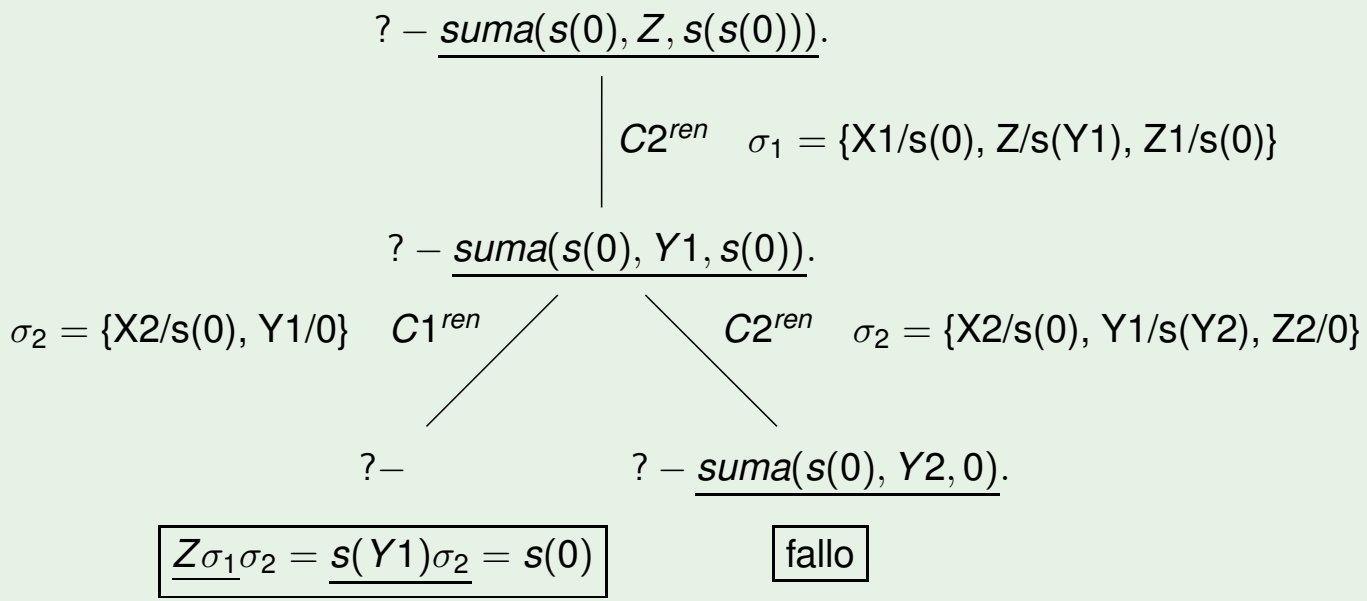
_P = v. semianónima



_ = v. anónima

Ejemplo (Árbol de Resolución programa "suma")

```
suma(X,0,X).                          % C1
suma(X,s(Y),s(Z)) :-  suma(X,Y,Z).    % C2
```



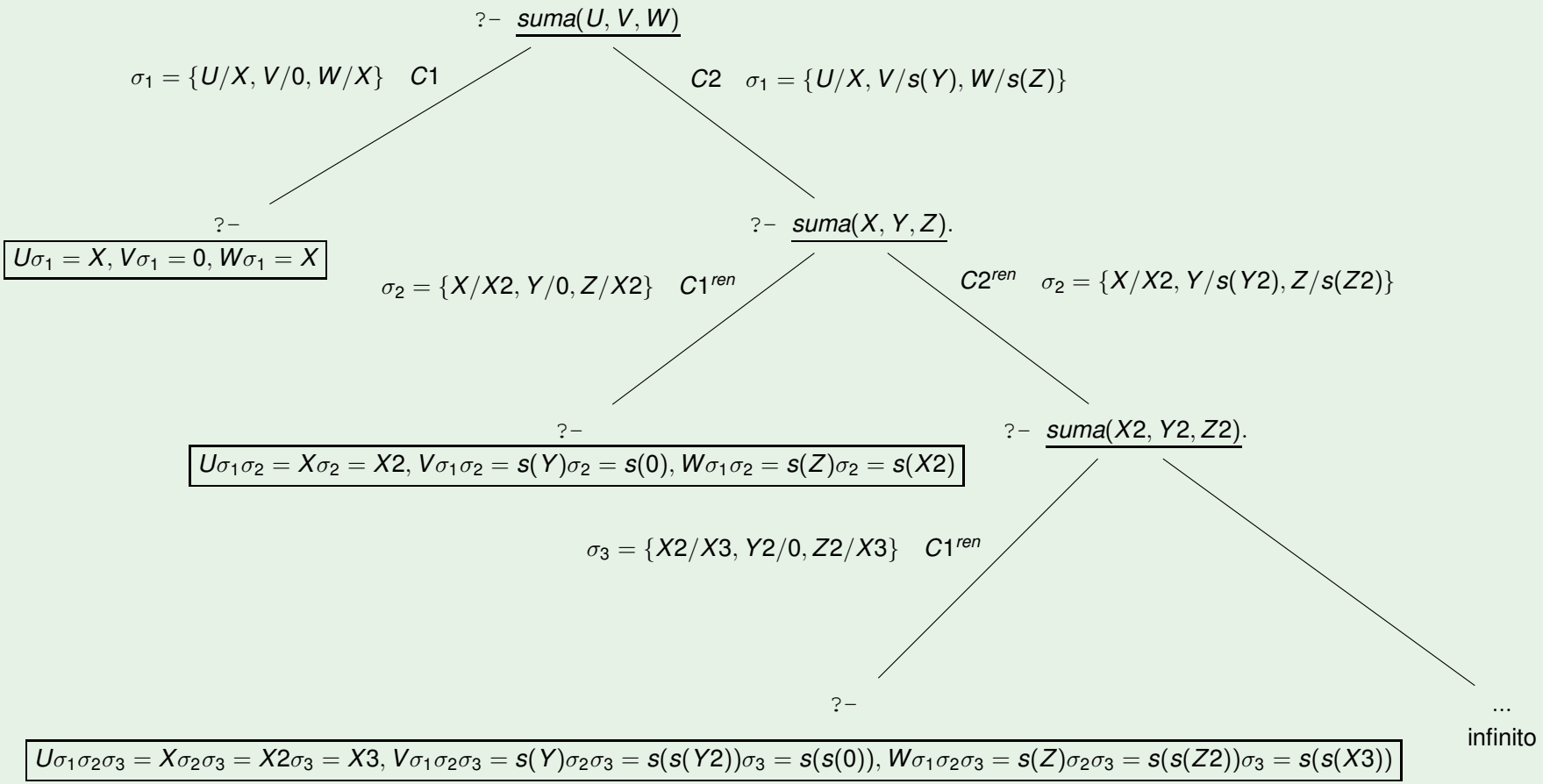
Observación (Árboles con ramas infinitas)

Los Árboles de Resolución también pueden tener ramas infinitas, que pueden ser de dos clases:

- Ramas con un número infinito de soluciones que van apareciendo sucesivamente (ver ejemplo a continuación con el programa de la suma), o bien
- Ramas sin soluciones que desarrollan hijos indefinidamente, hasta que acaban desbordando la memoria (ver ejemplo más adelante con la tercera versión del programa “ancestro”, `ancestro3`, rama de más a la derecha).

Ejemplo (Árbol de Resolución con infinitas soluciones)

```
suma(X,0,X).                              % C1
suma(X,s(Y),s(Z)) :-  suma(X,Y,Z).        % C2
```



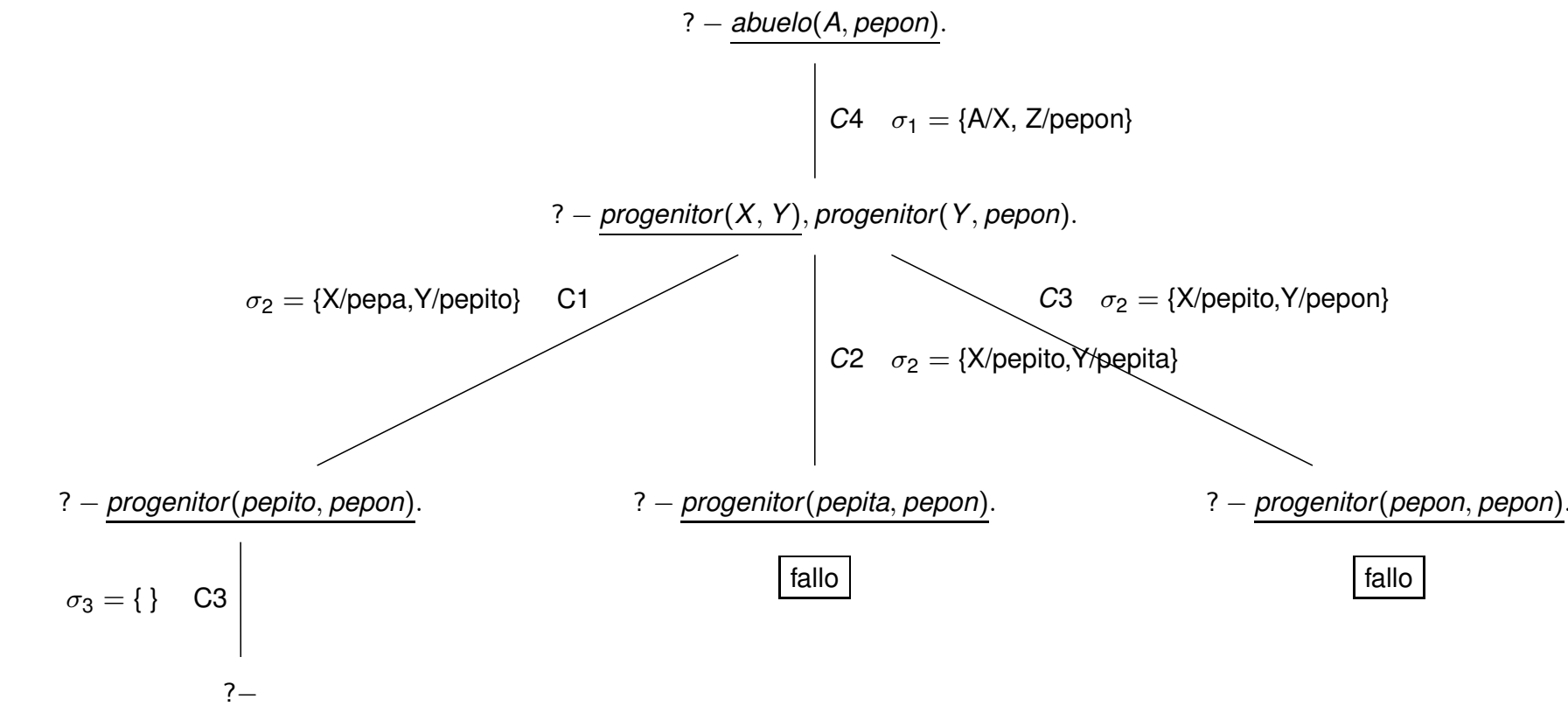
Ejercicios (Construcción de Árboles de Resolución)

Construya los Árboles de Resolución asociados con los programas y consultas dados a continuación:

1. ▸ Programa Abuelos *y consulta (escríbala en LPO y en* PROLOG*) adecuada para averiguar quiénes son abuelas/os de Pepón.*
2. ▸ Programa Abuelos *y consultas (escríbalas en LPO y en* PROLOG*) adecuadas para averiguar si Pepito y Pepón son hermanos (tienen algún progenitor en común) y, en caso afirmativo, saber quiénes son esos progenitores comunes. ¿Y Pepita y Pepón?*
3. *Programa Amistades/Enemistades y consulta (escríbala en LPO y en* PROLOG*) adecuada para averiguar si Abel es amigo de alguien que, a su vez, sea enemigo de alguien, y, en caso afirmativo, saber quiénes serían.*
4. ▸ Programa Suma *y consulta* `?- suma(X,Y,s(s(0))).` *(escríbala en LPO y describa su objetivo en lenguaje natural).*

Soluciones propuestas:

1.



$\underline{A\sigma_1}\sigma_2\sigma_3 = \underline{X\sigma_2}\sigma_3 = \underline{pepa\sigma_3} = pepa$

2a)

$$? - \underline{progenitor(X, pepito)}, progenitor(X, pepon).$$

$$\Big| \quad C1 \quad \sigma_1 = \{X/pepa\}$$

$$? - \underline{progenitor(pepa, pepon)}.$$

$$\boxed{\text{fallo}}$$

2b)

$$? - \underline{progenitor(X, pepita)}, progenitor(X, pepon).$$

$$\Big| \quad C2 \quad \sigma_1 = \{X/pepito\}$$

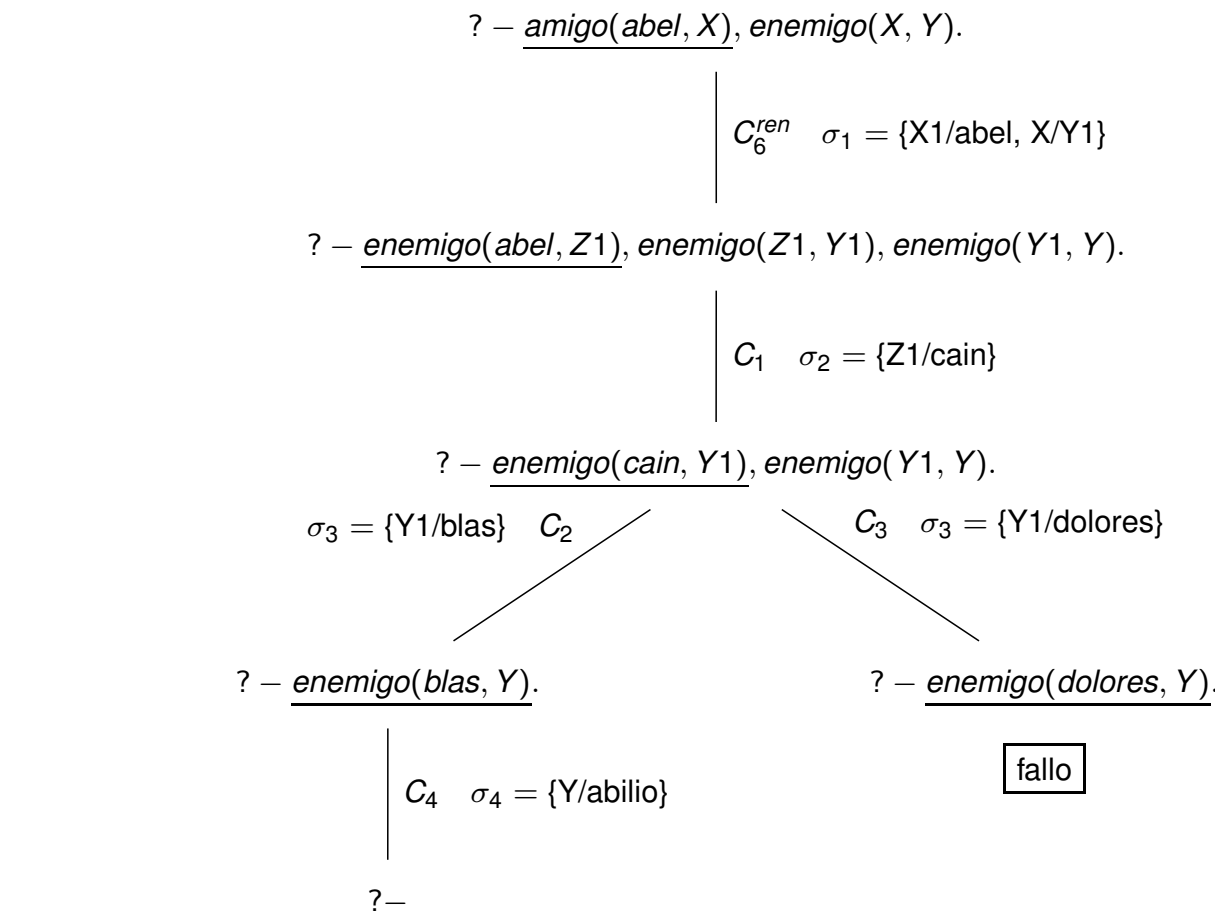
$$? - \underline{progenitor(pepito, pepon)}.$$

$$\Big| \quad C3 \quad \sigma_2 = \{a\}$$

$$?-$$

$$\boxed{\underline{X\sigma_1\sigma_2} = \underline{pepito\sigma_2} = pepito}$$

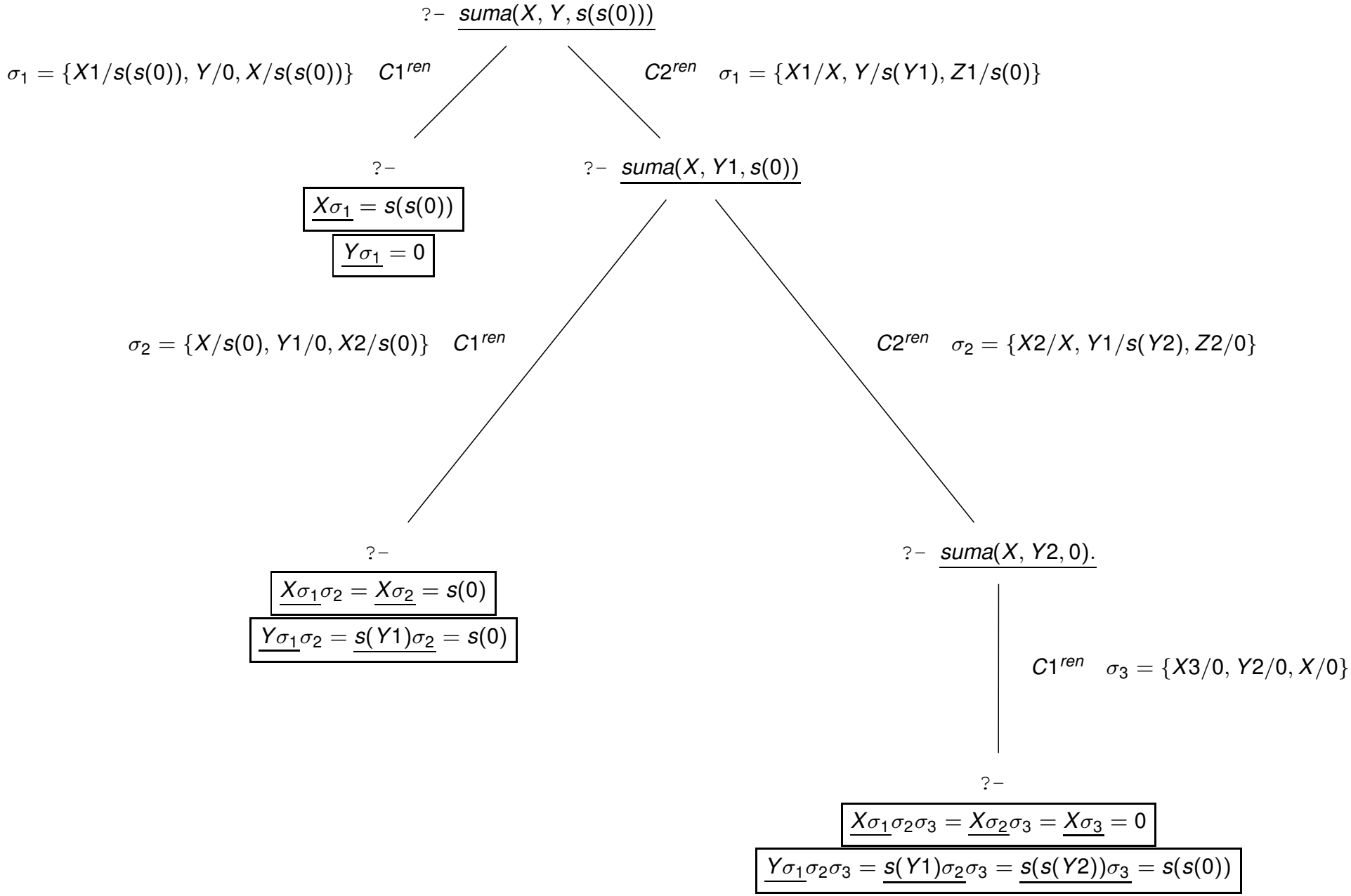
3.



$$\boxed{X\sigma_1\sigma_2\sigma_3\sigma_4 = Y1\sigma_2\sigma_3\sigma_4 = Y1\sigma_3\sigma_4 = blas\,\sigma_4 = blas}$$

$$\boxed{Y\sigma_1\sigma_2\sigma_3\sigma_4 = Y\sigma_2\sigma_3\sigma_4 = Y\sigma_3\sigma_4 = Y\sigma_4 = abilio}$$

4. LPO $\exists X \exists Y suma(X, Y, s(s(0)))$. ¿Qué formas hay de descomponer el natural 2 en dos sumandos?

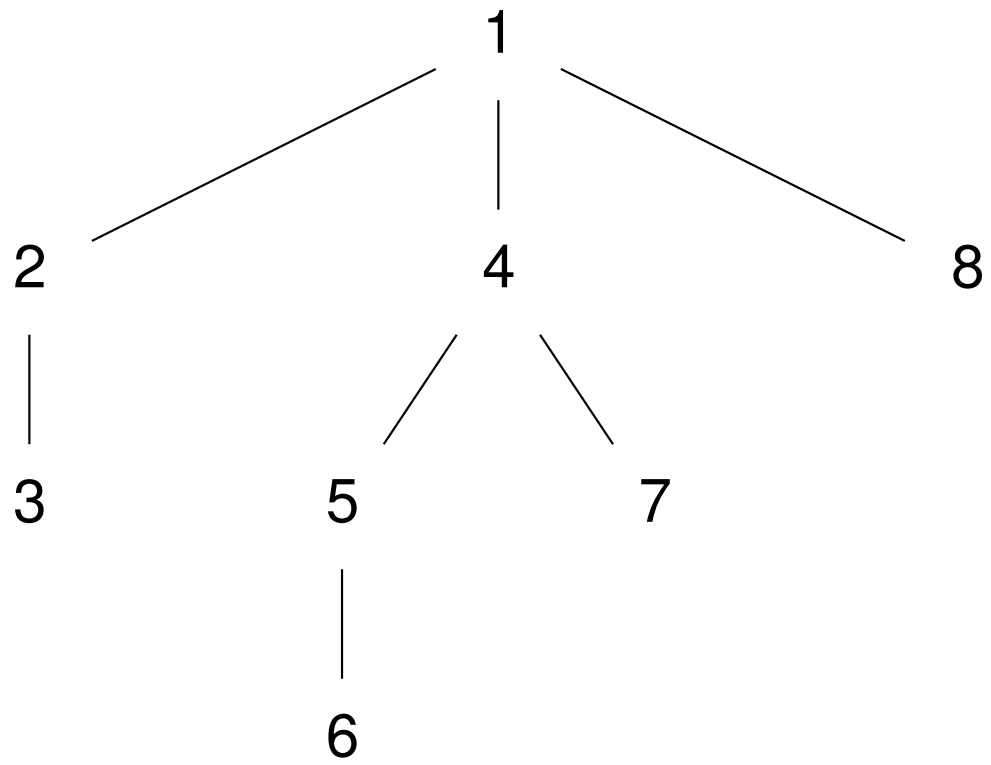


Mecanismo de cómputo de PROLOG

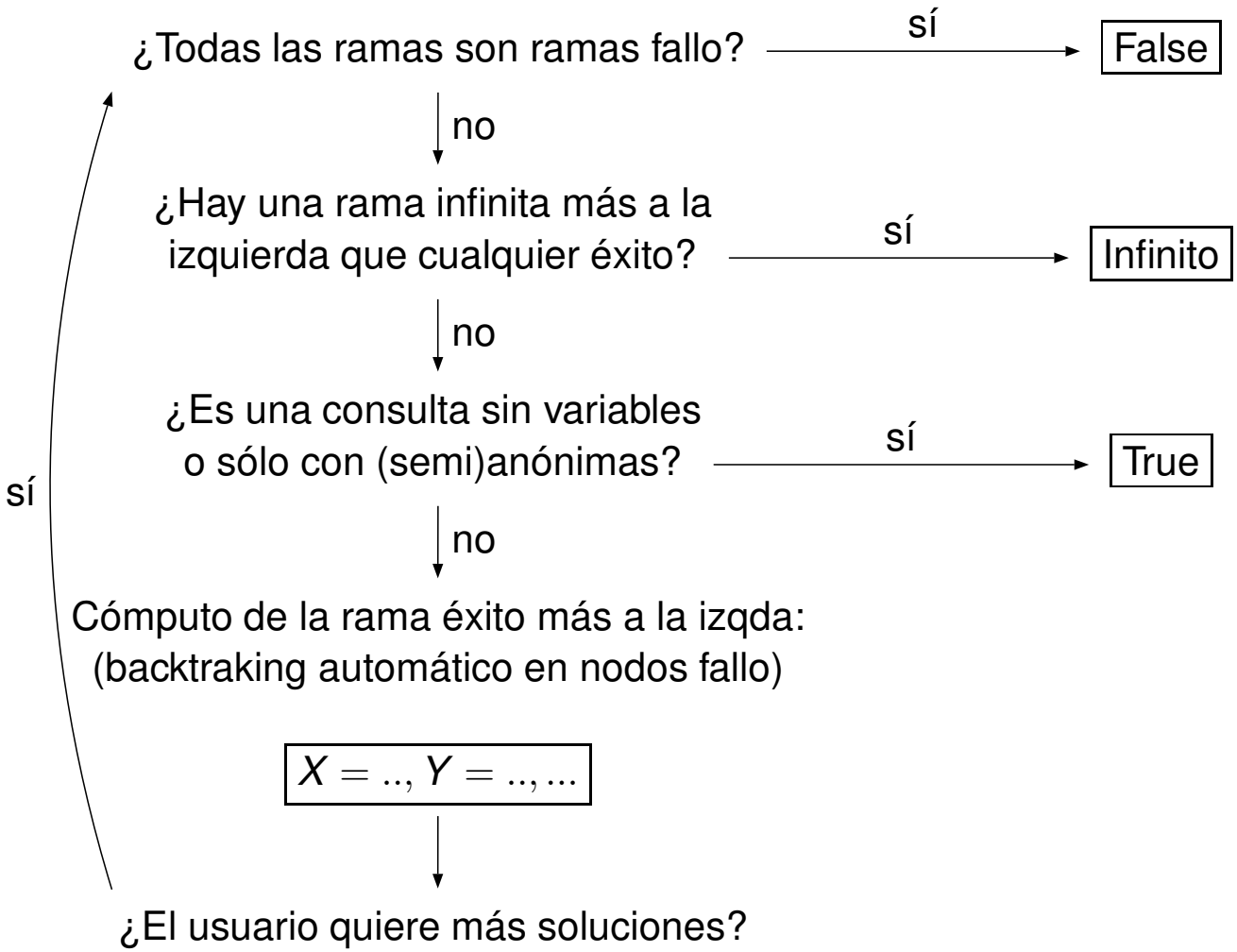
Cuando recibe una consulta relativa a un programa, PROLOG construye el Árbol de Resolución correspondiente de la siguiente forma:

- El árbol se construye en profundidad por la izquierda (ver dibujo en la página siguiente).
- El árbol se construye utilizando retroceso (backtracking) automático al llegar a un nodo fallo: cuando se encuentra con uno de estos nodos, PROLOG *no da cuenta del fallo*, simplemente retrocede para continuar la búsqueda (en profundidad) de posibles éxitos.
- El árbol no se construye necesariamente entero: en cada nodo *éxito* el sistema pregunta al usuario si quiere seguir o terminar la ejecución de la consulta.

Ejemplo de orden (en profundidad) con el que PROLOG construye los Árboles de Resolución:



Actitud de PROLOG al construir (en profundidad) un Árbol de Resolución



Ejemplo (Respuestas ofrecidas por PROLOG)

¿Qué respuestas ofrecería PROLOG, y en qué orden, al construir los Árboles de Resolución, completos, de los últimos ejemplos?

- En los ejemplos ▸ Árboles de Resolución programa Abuelas/os:
  - En el primer árbol: primera solución: `N = pepita`, segunda solución: `N = pepon` y a continuación `false` (este último indicando que no hay más soluciones).
  - Las respuestas para los otros dos árboles serían `true` y `false`.
- En el ejemplo ▸ Árbol de Resolución programa Suma:
  - Primera solución: `Z = s(0)`
  - `false` (indicando que no hay más soluciones -el nodo fallo no se reporta-)
- En el ejemplo ▸ Árbol de Resolución con infinitas soluciones:
  - Primera solución: `U = X, V = 0, W = X`
  - Segunda solución: `U = X2, V = s(0), W = s(X2)`
  - Tercera solución: `U = X3, V = s(s(0)), W = s(s(X3))`
  - Cuarta solución: ... (seguiría dando todas las posibles soluciones)

Ejercicios (Respuestas ofrecidas por PROLOG)

Para cada uno de los Árboles construidos en los ejercicios ▸ *Construcción de Árboles de Resolución*, *indique qué respuestas ofrecería* PROLOG, *y en qué orden lo haría.*

Soluciones propuestas:

1. “Abuelas/os”, `?- abuelo(A, pepon).`
   - Primera solución: `A = pepa`
   - False (indicando que no hay más soluciones, los nodos fallo no se reportan).

2a. “Abuelas/os”, `?- progenitor(X, pepito), progenitor(X,pepon).`
   - False (indicando que no hay soluciones)

2b. “Abuelas/os”, `?- progenitor(X, pepita), progenitor(X,pepon).`
   - Primera solución: `X = pepito`
   - False (indicando que no hay más soluciones)

3. “Amistades/Enemistades”, `?- amigo(abel,X), enemigo(X,Y).`
   - Primera solución: `X = blas, Y = abilio`
   - False (indicando que no hay más soluciones, los nodos fallo no se reportan)

4. “Suma”, `?- suma(X, Y, s(s(0))).`
   - Primera solución: `X = s(s(0)), Y = 0`
   - Segunda solución: `X = s(0), Y = s(0)`
   - Tercera solución: `X = 0, Y = s(s(0))`
   - False (indicando que no hay más soluciones)

Observación (Influencia del orden en programas y reglas)

Tanto:

1. El orden de las cláusulas en el programa, como
2. El orden de los predicados en el cuerpo de una regla.

pueden influir en:

1. Qué soluciones se encuentran y en qué orden aparecen.
2. La terminación o no de las consultas que se realizan.

En particular, la búsqueda de soluciones no es completa: no asegura encontrar una solución a la consulta aunque la haya (esto ocurrirá siempre que el Árbol de Resolución tenga alguna rama infinita más a la izquierda que la primera rama éxito).

El predicado `ancestro`, cuya implementación se discute a continuación, se usa a menudo para ilustrar lo anterior.

Ejemplo (El predicado `ancestro`)

Dado `progenitor(X,Y)`, cierto si X es progenitor/a de Y, se define el predicado `ancestro(X,Y)`, cierto si X es un ancestro de Y, es decir, si X es progenitor/abuela-o/bisabuela-o/etc de Y.

- Caso base. Los progenitores son ancestros de sus hijas/os: $\forall X \forall Y(progenitor(X, Y) \rightarrow ancestro(X, Y))$. En Prolog:

```
ancestro(X, Y) :-
  progrenitor(X, Y).
```

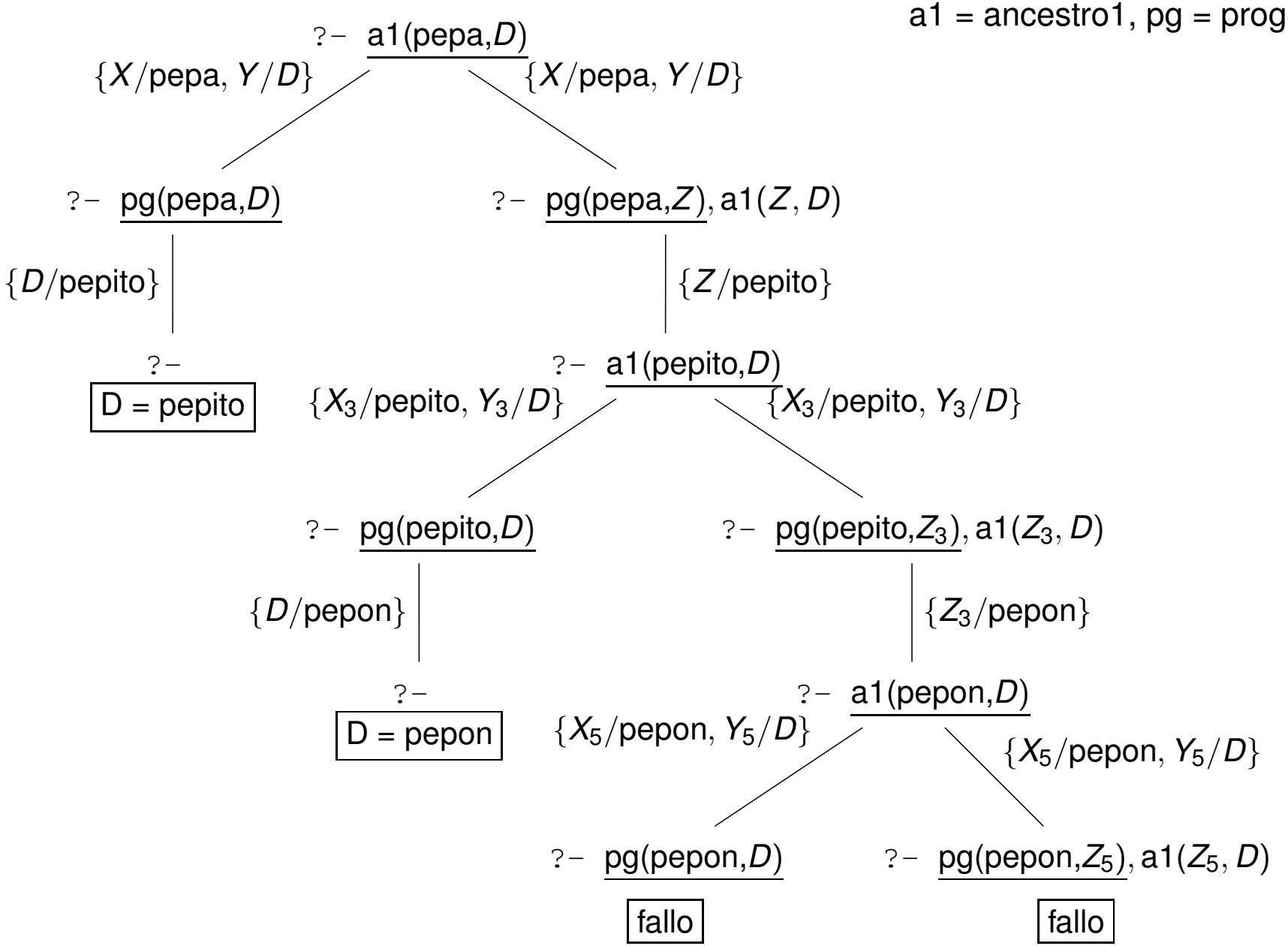
- Caso recursivo. Los progenitores son ancestros de todos aquellos de los que sus hijas/os lo son. $\forall X \forall Y \forall Z((progenitor(X, Z) \wedge ancestro(Z, Y)) \rightarrow ancestro(X, Y))$. En Prolog:

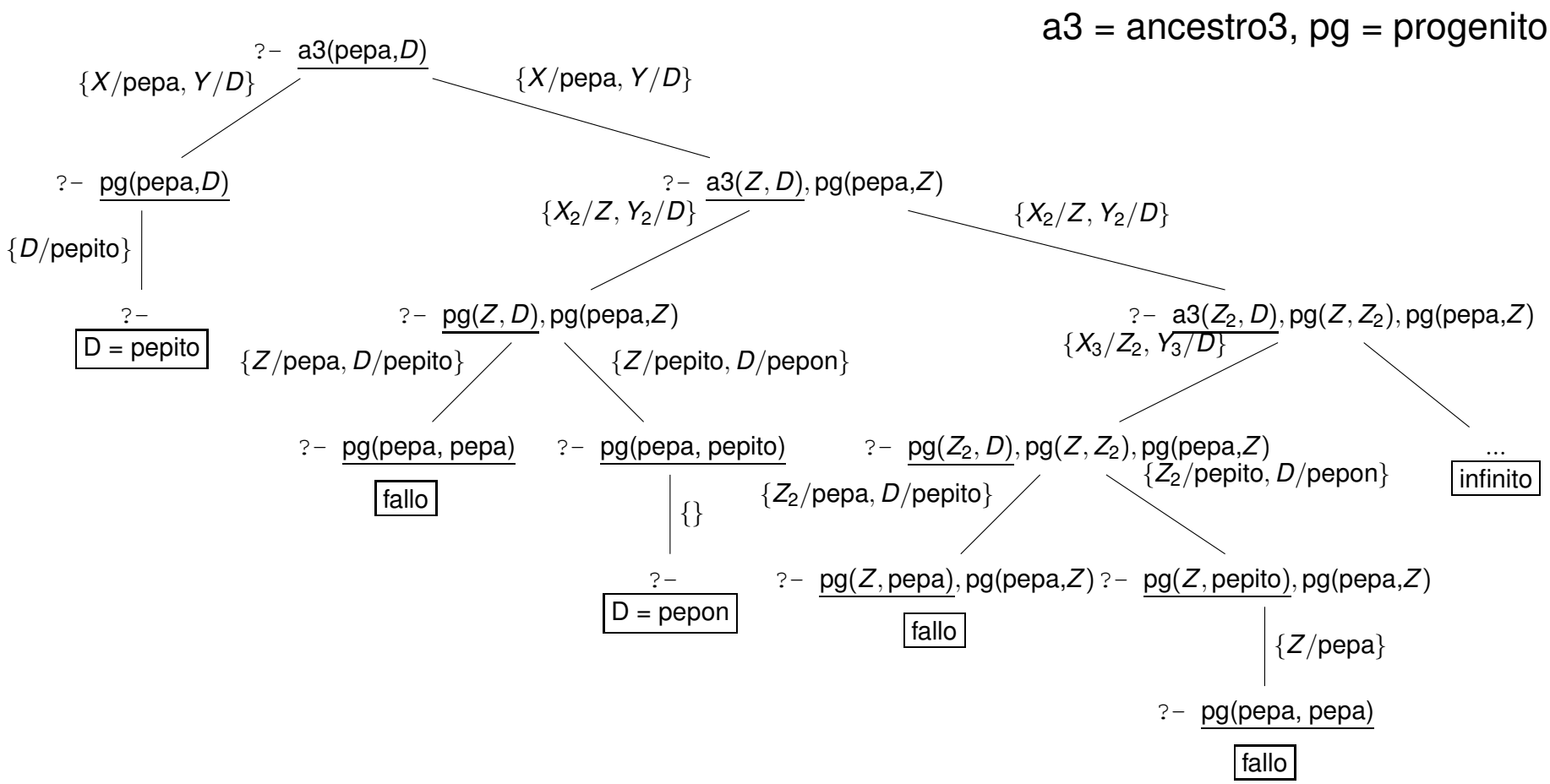
```
ancestro(X, Y) :-
  progrenitor(X, Z),
  ancestro(Z, Y).
```

Ejemplo (Influencia del orden en programas y reglas)

```
progenitor(pepa, pepito).
progenitor(pepito, pepon).
% CUATRO POSIBLES ORDENACIONES PARA "ancestro"
(todas equivalentes en Lógica, pero no en Prolog)

ancestro1(X,Y) :- progenitor(X,Y).
ancestro1(X,Y) :- progenitor(X,Z), ancestro1(Z,Y).

ancestro2(X,Y) :- progenitor(X,Z), ancestro2(Z,Y).
ancestro2(X,Y) :- progenitor(X,Y).

ancestro3(X,Y) :- progenitor(X,Y).
ancestro3(X,Y) :- ancestro3(Z,Y), progenitor(X,Z).

ancestro4(X,Y) :- ancestro4(Z,Y), progenitor(X,Z).
ancestro4(X,Y) :- progenitor(X,Y).
```

a1 = ancestro1, pg = progenitor





Los árboles de `ancestro1` y `ancestro3` son muy distintos entre sí. Los de `ancestro2` y `ancestro4` no son otra cosa que las *imágenes especulares* de los árboles de `ancestro1` y `ancestro3`, respectivamente.

Respuestas ofrecidas: PROLOG construye los Árboles de Resolución correspondientes en profundidad, ofreciendo las soluciones de las ramas éxito según las va encontrando y sin reportar las ramas fallo (donde hace backtracking de forma automática):

```
?- ancestro1(pepa, D).
D = pepito ; D = pepon

?- ancestro2(pepa, D).
D = pepon ; D = pepito. % Soluciones intercambiadas

?- ancestro3(pepa, D).
D = pepito ; D = pepon ; % Encuentra las soluciones
ERROR: Out of local stack % rama infinita

?- ancestro4(pepa, D). % ¡NO encuentra soluciones!
ERROR: Out of local stack  % rama infinita
```

- No existe ninguna regla general que establezca el orden óptimo de las cláusulas en el programa ni el orden óptimo de los predicados en el cuerpo de las reglas (depende de cada caso).
- En general, sí es recomendable empezar por "lo más sencillo":
  1. Colocar primero los casos base (cláusulas que expresan las condiciones de parada de la recursividad).
  2. Evitar las reglas con recursión a la izquierda (reglas tales que el predicado del primer subobjetivo de su cuerpo coincide con el predicado de su cabeza), primando siempre que sea posible la recursión de cola o recursión final (recursión en último lugar).

Ejemplo

De las cuatro versiones del predicado `ancestro`, la única que cumple las dos recomendaciones anteriores es la primera, `ancestro1`.

Ejercicios (El orden importa + uso del depurador)

Ejercicio nº 4 de la Práctica de PROLOG nº 1.

Ejercicios (Mecanismo de cómputo de PROLOG)

1. *Considere el programa*

```
p(X,X).                          % C1
p(X,Z) :- q(X,Y), p(Y,Z).        % C2
q(a,b).                          % C3
```

   1. *Exprese en lenguaje natural y en LPO el significado de cada una de las cláusulas del programa.*
   2. *Construya el Árbol de Resolución correspondiente a la consulta* `?- p(X,b).` *¿Qué se pretende averiguar? ¿Qué respuestas ofrecería* PROLOG*, y en qué orden?*
   3. *¿Qué consecuencias tendría el intercambio de las dos primeras cláusulas del programa?*
   4. *¿Qué consecuencias tendría el intercambio de los dos predicados del cuerpo de la regla?*
   5. *¿Qué significado tiene la consulta* `?- p(X,b), q(X,b)` *? Construya el Árbol de Resolución necesario e indique qué respondería* PROLOG *ante esta consulta.*

Soluciones propuestas:

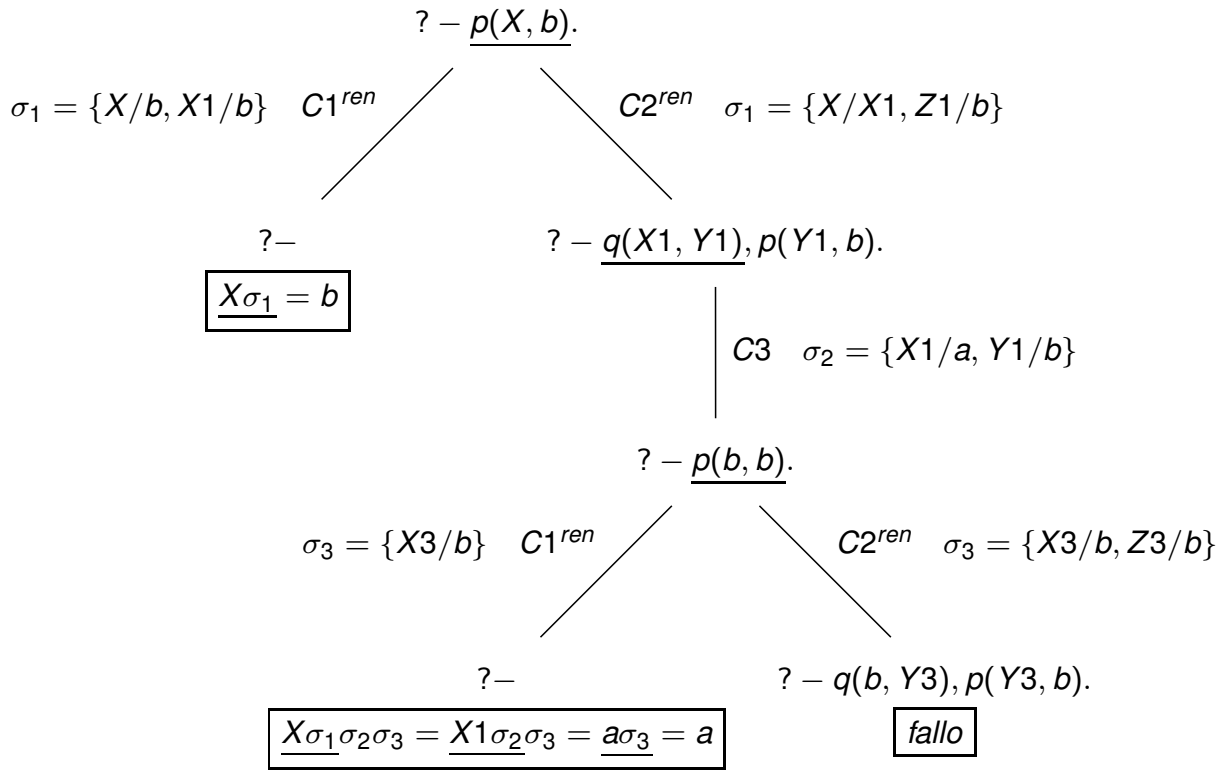
1. El programa define dos relaciones binarias, `p/2` y `q/2`, de la siguiente forma:
   - C1: todo objeto está relacionado consigo mismo mediante la relación `p/2` (es decir, la relación `p` es reflexiva). LPO: $\forall X\ p(X, X)$.
   - C2: para cualesquiera objetos X,Y,Z, si X está relacionado mediante `q` con Y y ese Y está relacionado con Z mediante `p`, entonces X y Z están relacionados mediante `p`. LPO: $\forall X \forall Y \forall Z((q(X, Y) \wedge p(Y, Z)) \rightarrow p(X, Z))$.
   - El objeto `a` está relacionado con el objeto `b` mediante `q`. LPO: $q(a, b)$.

2. La consulta `?- p(X,b)` pretende averiguar si existe algún objeto relacionado con `b` mediante `p`, y, en caso afirmativo, qué objeto u objetos cumplen lo anterior. En LPO: $\exists X\ p(X, b)$. Ver Árbol y soluciones en página siguiente.

3. Si se intercambian las dos primeras cláusulas, el Árbol de Resolución sería la imagen especular (imagen generada al reflejarse sobre un espejo) del árbol del apartado anterior, por lo que las soluciones aparecerían intercambiadas: la primera solución sería `X=a` y la segunda `X=b`.

4. Si se intercambian los predicados del cuerpo de la Regla se pasa de una recursión de cola o final (a la derecha) a una recursión a la izquierda, y la estructura del Árbol cambia. Este Árbol está dibujado más adelante y también puede encontrarse una descripción detallada de su construcción aquí.

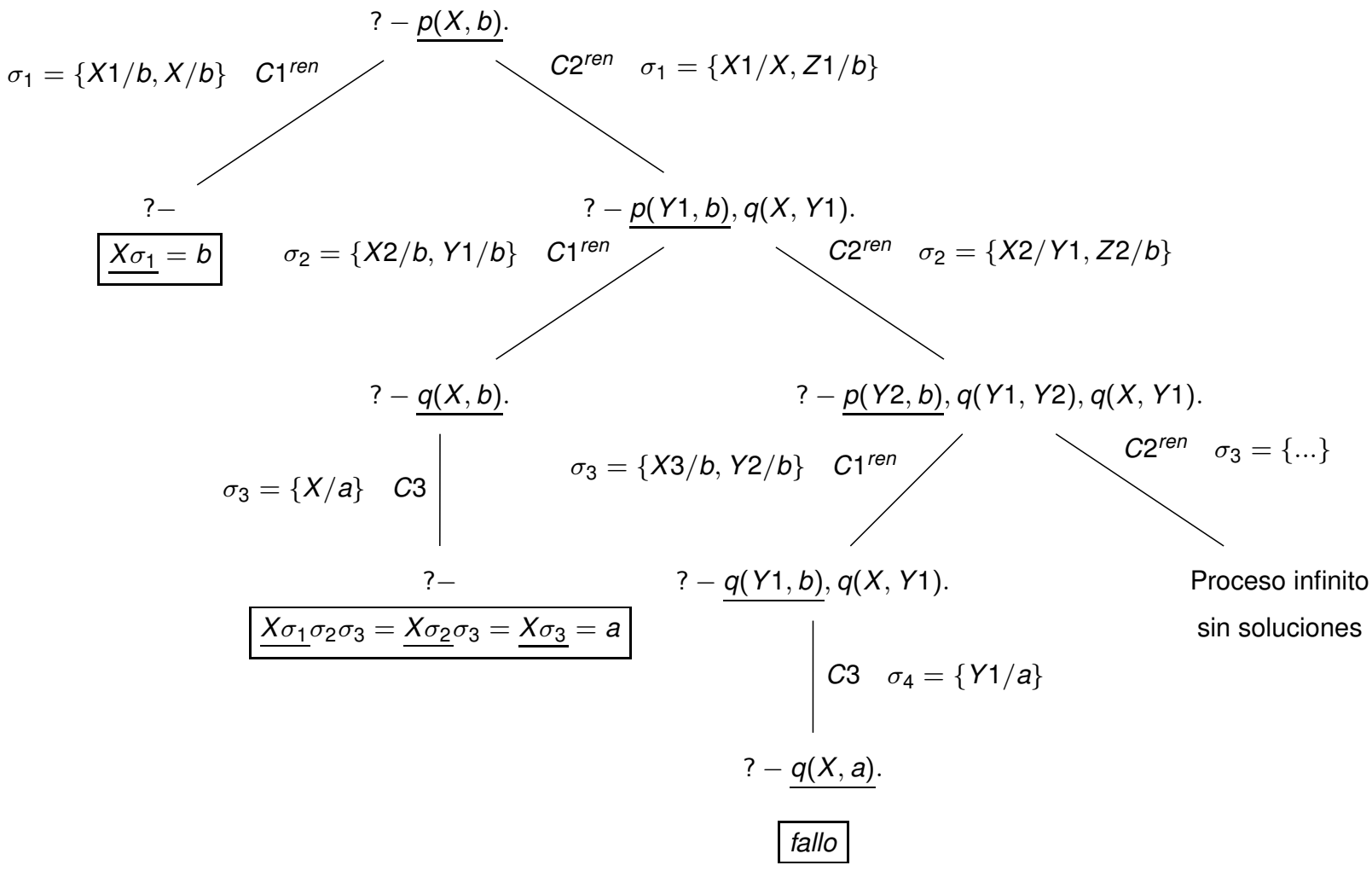
2.



PROLOG construye el Árbol de Resolución anterior en profundidad, haciendo backtracking automático en las ramas fallo (sin reportar estos fallos), por lo que las soluciones que ofrece ante la consulta dada son:

- Primera solución: $X = b$
- Segunda solución: $X = a$
- False (indicando que no hay más soluciones)

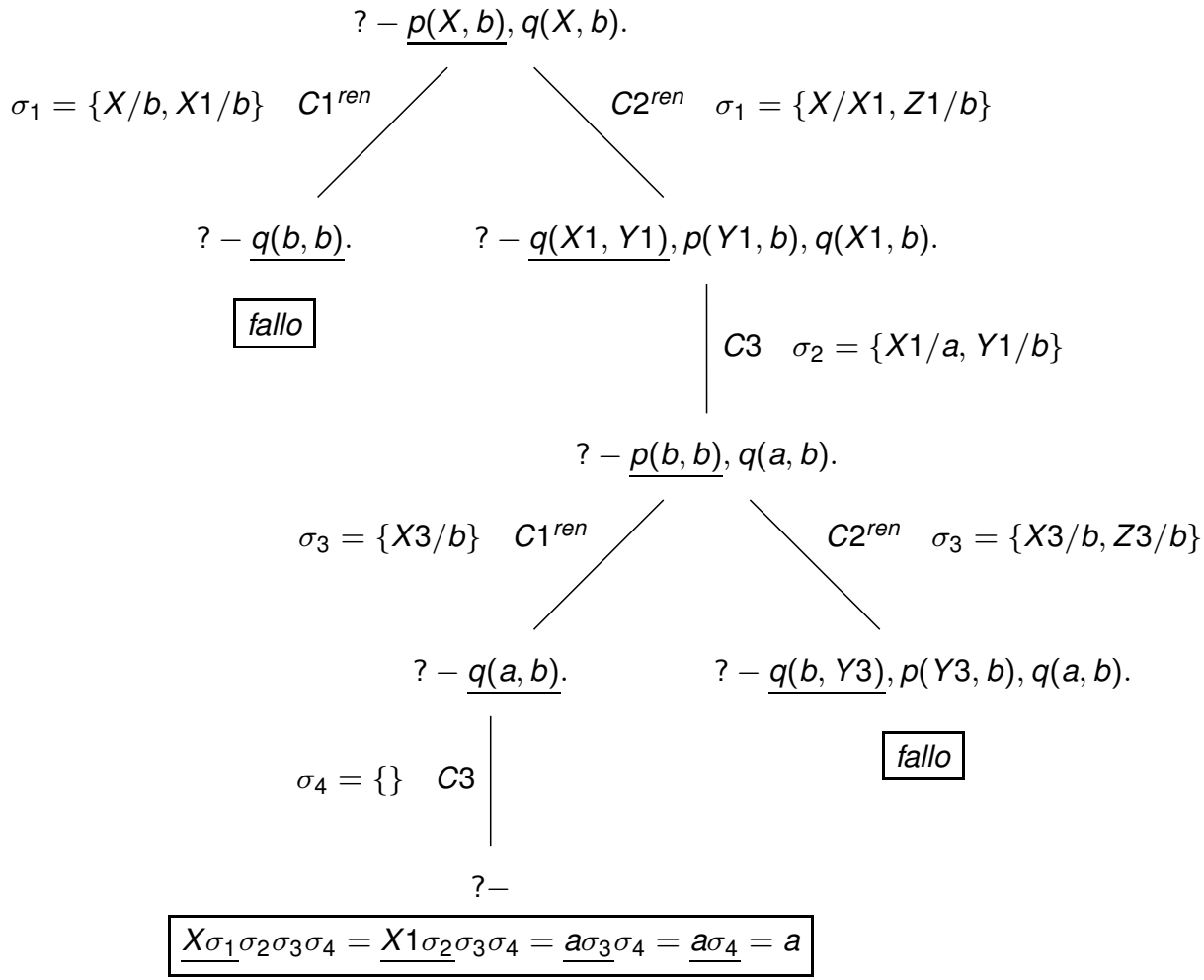
4.



A la vista del árbol anterior, la respuesta de PROLOG será la siguiente:

- Primera solución: $X = b$
- Segunda solución: $X = a$
- ERROR: Out of local stack (por computación infinita)

5. ¿Qué objeto(s) existen(n) que estén relacionado con `b` tanto mediante `p` como mediante `q`?



PROLOG construye el Árbol de Resolución anterior en profundidad, haciendo backtracking automático en las ramas fallo (sin reportar estos fallos):

- Primera solución: $X = a$
- False (indicando que no hay más soluciones)

BIBLIOGRAFÍA

- L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.
- W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fifth edition, 2003.
- I. Bratko.*Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 2001.
- J. Lloyd. *Foundations of Logic Programming*, (Second Edition). Springer-Verlag, 1987.
- R. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
- U. Nilsson and J. Maluszynski. Logic, Programming and Prolog. John Wiley & Sons Ltd, 1996.
- SWI-Prolog, entorno de programación en Prolog de dominio público.
- comp.lang.prolog. Faq

PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL2: El lenguaje PROLOG, aspectos básicos

4. Clasificación y Comparación de Términos

Grado en Ingeniería Informática

URJC

Ana Pradera

Contenido

1. INTRODUCCIÓN
2. CLASIFICACIÓN DE TÉRMINOS
3. COMPARACIÓN DE TÉRMINOS

INTRODUCCIÓN

- Recuerde que la sintaxis de PROLOG clasifica los términos, que sirven para representar objetos, en tres grandes categorías:
  - **Constantes**, que pueden a su vez ser números -enteros o reales- o átomos (estos últimos construidos mediante cadenas de letras, dígitos y subrayado empezando por minúscula o cadenas "...").
  - **Variables** (cadenas de letras, dígitos y subrayado empezando por mayúscula o subrayado).
  - **Términos compuestos o estructuras** (un nombre seguido, entre paréntesis, por una serie de términos separados por comas).
- En ocasiones es conveniente poder tanto clasificar (¿qué tipo de término es?) como comparar (¿son iguales? ¿cuál va antes/después?) términos entre sí, por lo que PROLOG ofrece predicados predefinidos para estos cometidos.

CLASIFICACIÓN DE TÉRMINOS

- Constantes:
  `atomic(+T)`, cierto si `T` es una constante (número o átomo).
  - Números:
    `number(+T)`, cierto si `T` es un número.
    - `integer(+T)`, cierto si `T` es un número entero.
    - `float(+T)`, cierto si `T` es un número real.
  - Átomos:
    `atom(+T)`, cierto si `T` es un átomo.
- Variables:
  `var(+T)`, cierto si `T` es una variable;
  `nonvar(+T)`, cierto si `T` no es una variable.
- Términos compuestos:
  `compound(+T)`, cierto si `T` es un término compuesto.

Ejemplos (Clasificación de términos)

```
?- number(12).
true
?- integer(X).
false  %% OJO, solo acepta argumentos de entrada:
       %% NO sirve para generar enteros
?- atomic(X).
false.
?- X = 5.2, atomic(X).
X=5.2
?- var(X), X = 4, integer(X).
X=4
?- X = 4, var(X).
false
?- compound(progenitor(X,Y)).
true
```

COMPARACIÓN DE TÉRMINOS

- `T1 == T2`
  Cierto si los términos `T1` y `T2` son literalmente idénticos, es decir, son la misma constante, la misma variable -con el mismo nombre- o dos términos compuestos literalmente idénticos -mismo nombre, misma aridad y argumentos idénticos-.
- `T1 \== T2`
  Cierto si los términos no son literalmente idénticos.
- `T1 @< T2`, `T1 @=< T2`, `T1 @> T2`, `T1 @>= T2`
  Cierto si `T1` es menor, menor o igual, mayor, mayor o igual que `T2` atendiendo al orden estándar entre términos:
  Variables < Números < Cadenas < Átomos < Compuestos
  y donde las variables se ordenan por dirección, los números por su valor, las cadenas ("...") y los átomos alfabéticamente, y los términos compuestos por su aridad, functor y argumentos.

Ejemplos

```
?- X \== a.
true
?- X == Y.
false
?- X = Y, X == Y.
true (la unificación hace idénticas a las variables)
?-progenitor(pepe,X) == progenitor(pepe,Y).
false.
?- adios @=< hola.
true (orden alfabético)
?- ancestro(pepa,pepita) @< hola(mundo).
false (ordenado por aridad: 2 no es menor que 1)
?- ancestro(pepa) @< hola(mundo).
true (misma aridad: orden alfabético functor)
?- abuelo(pepon, pepito) @> abuelo(pepa, pepito).
true (misma aridad y functor: orden argumentos)
```

BIBLIOGRAFÍA

- L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.
- W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fifth edition, 2003.
- I. Bratko.*Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 2001.
- J. Lloyd. *Foundations of Logic Programming*, (Second Edition). Springer-Verlag, 1987.
- R. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
- U. Nilsson and J. Maluszynski. Logic, Programming and Prolog. John Wiley & Sons Ltd, 1996.
- SWI-Prolog, entorno de programación en Prolog de dominio público.
- comp.lang.prolog. Faq

PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL2: El lenguaje PROLOG, aspectos básicos

5. Aritmética

Grado en Ingeniería Informática

URJC

Ana Pradera

Contenido

1. INTRODUCCIÓN
2. CONSTANTES Y OPERADORES ARITMÉTICOS
3. PREDICADOS ARITMÉTICOS
4. ALGUNOS PROGRAMAS ARITMÉTICOS

INTRODUCCIÓN

- PROLOG incluye herramientas extra-lógicas que permiten realizar de forma eficiente todas las operaciones aritméticas habituales en cualquier lenguaje de programación.

- Estas herramientas son de dos tipos:
  - Constantes y operadores aritméticos, que sirven para **construir** expresiones aritméticas.
  - Predicados aritméticos, que sirven para **evaluar** las expresiones aritméticas construidas mediante los operadores aritméticos anteriores.

- En lo que sigue se introducen tanto los operadores como los predicados aritméticos, y se ilustra su funcionamiento mediante algunos ejemplos de programas aritméticos.

CONSTANTES Y OPERADORES ARITMÉTICOS

Constantes numéricas

Las habituales: enteros y reales en notación decimal o científica.

Operadores aritméticos

`X+Y`	suma de X e Y
`X-Y`	X menos Y
`X*Y`	producto de X por Y
`X/Y`	cociente real de la división de X por Y
`X//Y`	cociente entero de la división de X por Y
`X mod Y`	resto de la división entera de X por Y
`abs(X)`	valor absoluto de X
`sqrt(X)`	raíz cuadrada de X
`log(X)`	logaritmo neperiano de X
.....	

Observación

Los operadores aritméticos permiten simplemente construir expresiones aritméticas, pero estas no son más que *términos compuestos* (algunos, como los seis primeros de más arriba, en notación infija) que no representan ningún valor.

Ejemplo

La expresión `3+5` no es más que el término compuesto `+(3,5)` escrito en notación infija.

- " `?- 3+5.`": ERROR, puesto que "+" no es un predicado.
- La consulta " `?- 3+5 = 8.`" devuelve `false`, dado que el término compuesto `3+5`, equivalente a `+(3,5)`, no es unificable con el término constante `8`.

Para poder evaluar expresiones aritméticas en PROLOG hay que utilizar los predicados aritméticos que se describen a continuación.

PREDICADOS ARITMÉTICOS

`X is Y` Si Y es una expresión aritmética, esta se evalúa y el resultado se intenta *unificar* con X (término cualquiera).

A la hora de usar este predicado hay que tener en cuenta las siguientes consideraciones:

1. Su uso puede dar lugar a un error en los dos siguientes casos:
   1. cuando la parte derecha no es una expresión aritmética:
      ```
      ?- X is a+1. % a es una constante no numérica
      { ... ERROR: ...}
      ```
   2. cuando la parte derecha es una expresión aritmética pero no se puede evaluar (error de instanciación):
      ```
      ?- X is 4*Z. % Z es una variable sin valor concreto
      {... ERROR: ...}
      ```
2. Salvo en los casos anteriores, el resultado del predicado dependerá de si la parte izquierda (tal cual aparece) unifica o no con el resultado obtenido al evaluar la parte derecha.

Ejemplos (Uso del predicado aritmético `is`)

`?- X is sqrt(4).` resultado: X = 2.0

% sqrt(4) da 2, que se puede unficar (y se unifica) con X mediante u.m.g. $\{X = 2\}$

`?- 5 is 2+3.` resultado: true

% 2+3 da 5, que se puede unificar con 5 con u.m.g. vacío

`?- 3+5 is 3+5.` resultado: false

% el 3+5 de la dcha da 8, que NO se puede unificar con el 3+5=+(3,5) de la izqda

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

% PROLOG construye el Árbol de Resolución siguiente:

$$?- \underline{X \text{ is } 5}, Y \text{ is } X + 1.$$
$$\Big| \sigma_1 = \{X/5\}$$
$$?- \underline{Y \text{ is } 5 + 1}.$$
$$\Big| \sigma_2 = \{Y/6\}$$
$$?-$$

$$\boxed{\underline{X\sigma_1\sigma_2} = \underline{5\sigma_2} = 5, \qquad \underline{Y\sigma_1\sigma_2} = \underline{Y\sigma_2} = 6}$$

Otros predicados aritméticos, en los que, a diferencia de `is`, *ambos* argumentos deben ser expresiones aritméticas evaluables (se producirá un error si esto no es así). Una vez evaluados X e Y:

`X=:=Y`	cierto si los valores numéricos de X e Y son iguales
`X=\=Y`	cierto si los valores numéricos de X e Y son distintos
`X<Y`	cierto si el valor numérico de X es menor que el de Y
`X=<Y`	cierto si el valor numérico de X es menor o igual que el de Y
`X>Y`	cierto si el valor numérico de X es mayor que el de Y
`X>=Y`	cierto si el valor numérico de X es mayor o igual que el de Y

Ejemplos (Uso de predicados aritméticos)

```
?- X+3 < sqrt(4).
 .. ERROR: ..
```

```
?- 3+5 =< 8.
 true (OJO, es =< y no <=)
```

```
?- 1+5 > abs(-8).
 false
```

```
?- 3 =\= 3*a.
 .. ERROR: ..
```

Observación

Conviene resaltar las diferencias entre los siguientes predicados predefinidos:

`X = Y` (`X \= Y`)	cierto si `X` e `Y` son términos que son (no son) unificables
`X =:= Y` (`X =\= Y`)	cierto si `X` e `Y` son expresiones aritméticas, se pueden evaluar, y sus valores numéricos son (no son) iguales.
`X == Y` (`X \== Y`)	cierto si `X` e `Y` son términos que son (no son) literalmente idénticos.

Ejercicios (Uso básico de operadores y predicados aritméticos)

Ejercicio nº 1 de la Práctica de PROLOG *nº 2.*

ALGUNOS PROGRAMAS ARITMÉTICOS

- En el apartado relativo a la sintaxis de PROLOG se ha discutido la implementación de algunos predicados aritméticos (suma, producto, par, impar) con lógica "pura".
- Son programas interesantes por su *versatilidad* (por ejemplo el predicado `suma` también sirve para restar o para descomponer un natural en sumandos) pero *incómodos* -uso de `s(X)` para representar a los naturales- e *ineficientes* -cálculo recursivo-.
- En la práctica, para realizar programas aritméticos se deben utilizar los operadores y predicados aritméticos de PROLOG.
- Lo anterior supone ganar en comodidad y eficiencia pero perder la versatilidad: observe por ejemplo cómo el programa para sumar que se incluye a continuación, debido al uso de `is`, requiere que sus dos primeros argumentos sean de *entrada*, por lo que ya no sirve ni para restar ni para descomponer.

Ejemplo (Programas aritméticos “suma, producto” y “par”)

```
% suma(+X,+Y,?Z)
% cierto si Z es la suma de X e Y
    suma(X,Y,Z) :-
        Z is X+Y.

% producto(+X,+Y,?Z)
% cierto si Z es el producto de X e Y
    producto(X,Y,Z) :-
        Z is X*Y.

% par(+X)
% cierto si X es par
    par(X) :-
        X mod 2 =:= 0.
```

Ejemplo (Algunas consultas a “suma” y “par”)

- `?- suma(0, 1, 1).` *True*
- `?- suma(1, 1, X).` X = 2
- `?- suma(1, 1, X), suma(X, 2, Z).` X = 2, Z = 4
- `?- suma(0,0,Z), suma(0,2,Z).` *False*
- `?- suma(1, Y, 3).` ERROR de instanciación

Esta versión de “suma” NO sirve para restar.

- `?- suma(X, Y, 2).` ERROR de instanciación
- `?- suma(X, Y, Z).` ERROR de instanciación

Esta versión de “suma” NO sirve para descomponer.

- `?- par(4).` *True*
- `?- par(X).` ERROR de instanciación

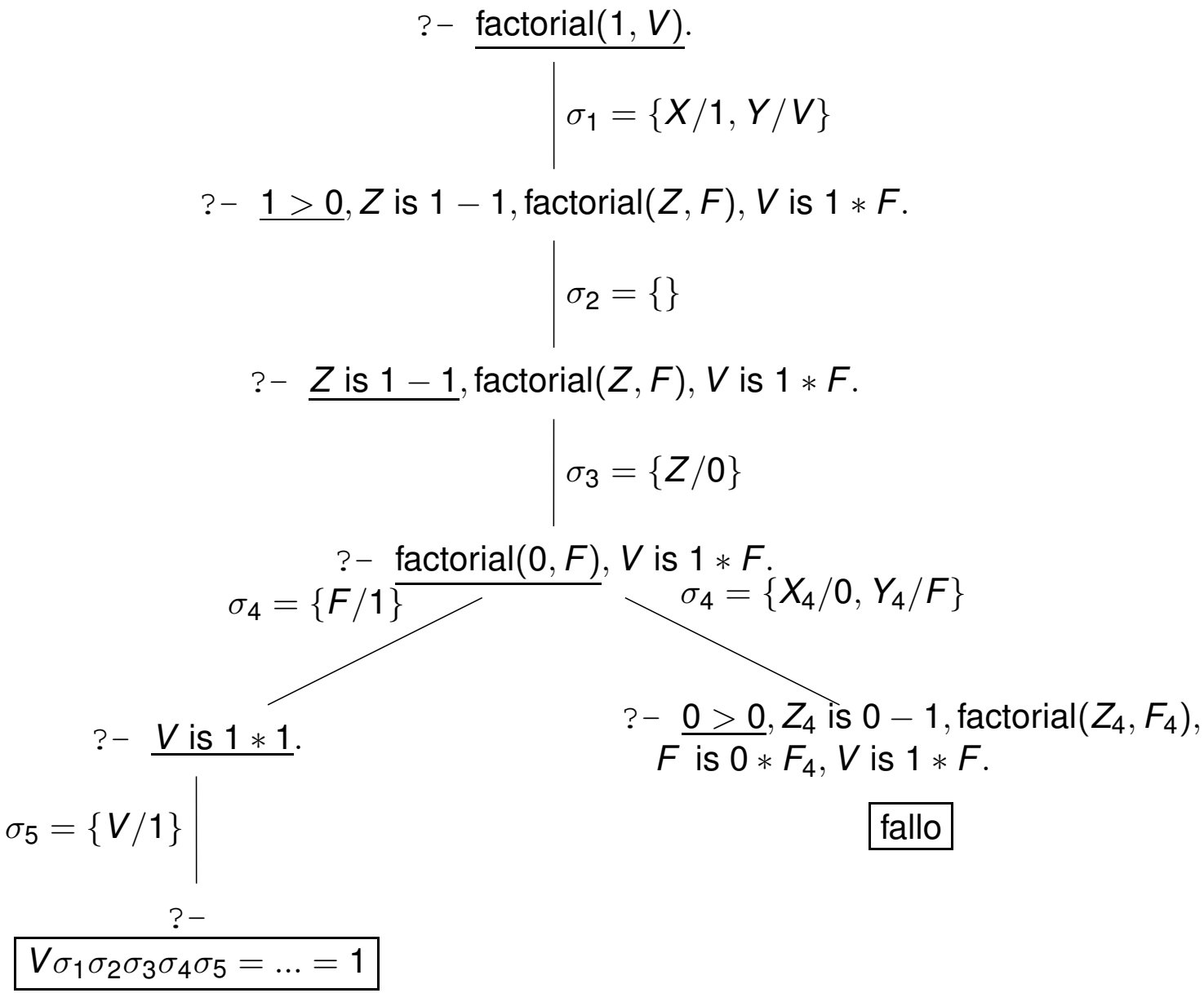
Esta versión de “par” NO sirve para generar números pares.

Ejemplo (Programa aritmético para calcular factoriales)

```
% factorial(+X, ?Y)
% cierto si Y es el factorial de X

    factorial(0, 1).

    factorial(X, Y) :-
        X > 0,
        Z is X-1,
        factorial(Z, F),
        Y is X*F.
```

Ejemplo (Algunas consultas a “factorial”)

- `?- factorial(1, 1).` *True*
- `?- factorial(4, X).` X = 24
- `?- factorial(X,24).` ERROR de instanciación



Respuesta PROLOG: `V=1` y `false` (no hay más soluciones)

Ejercicios (Cálculo de factoriales)

Ejercicio nº 2 de la Práctica de PROLOG *nº 2.*

Observación (Recursión final o de cola)

- La recursión de la implementación anterior de `factorial` es *recursión no final*: se realizan computaciones (en este caso una operación `is`) con posterioridad a la llamada recursiva.
- Por razones de eficiencia conviene utilizar siempre que se pueda recursión final (o recursión de cola), aquella en la que la llamada recursiva es lo último que se computa.
- Una técnica habitual para transformar una recursión no final en final es añadir uno o más parámetros adicionales, denominados parámetros de acumulación, en los que se irán acumulando los valores parciales necesarios en cada llamada recursiva.

Ejemplo (Factorial con recursión de cola, 1/2)

```
% factorial_rc(+X, ?Y)
% cierto si Y es el factorial de X.
% Implementación con recursión de cola.

% sobrecarga del predicado añadiendo -en medio-
% un parámetro de acumulación cuyo valor inicial
% es el del caso base (el factorial de 0, 1).

  factorial_rc(X, Y) :-
      factorial(X, 1, Y).

 % caso base
 % la salida es el valor acumulado

  factorial(0, Ac, Ac).
```

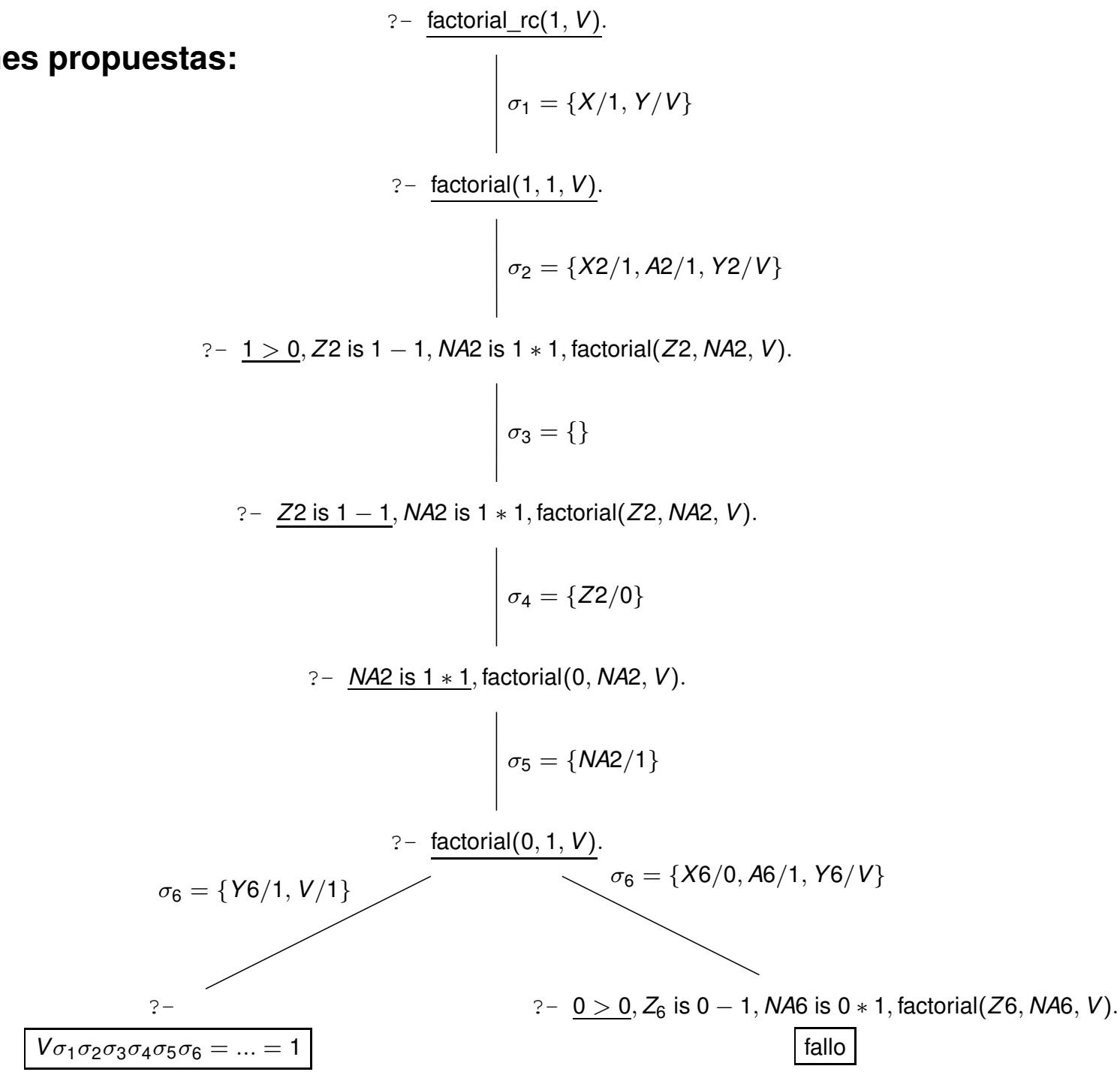
Ejemplo (Factorial con recursión de cola, 2/2)

```
% caso recursivo
% actualización del parámetro de acumulación
% previa a la llamada recursiva
  factorial(X, Ac, Y) :-
        X > 0,
        Z is X-1,
        NAc is X*Ac,    % actualización parámetro
        factorial(Z, NAc, Y).
```

Ejercicios (Programas aritméticos)

1. *Construya el Árbol de Resolución para la consulta* `?- factorial_rc(1,F)` *(con la implementación del predicado* `factorial` *mediante recursión de cola).*
2. *Ejercicios nº 3, 4 y 5 de la Práctica de* PROLOG *nº2.*

Soluciones propuestas:



BIBLIOGRAFÍA

- L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.
- W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fifth edition, 2003.
- I. Bratko.*Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 2001.
- J. Lloyd. *Foundations of Logic Programming*, (Second Edition). Springer-Verlag, 1987.
- R. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
- U. Nilsson and J. Maluszynski. Logic, Programming and Prolog. John Wiley & Sons Ltd, 1996.
- SWI-Prolog, entorno de programación en Prolog de dominio público.
- comp.lang.prolog. Faq

PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL2: El lenguaje PROLOG, aspectos básicos

6. Entrada/salida

Grado en Ingeniería Informática

URJC

Ana Pradera

Contenido

1. INTRODUCCIÓN
2. MANEJO DE FICHEROS
3. LECTURA/ESCRITURA

INTRODUCCIÓN

- PROLOG ofrece toda una serie de predicados predefinidos, extra-lógicos, para la realización de operaciones de entrada/salida.

- A continuación se describen algunos de los predicados de entrada/salida básicos de SWI-Prolog, muy similares a los de otros lenguajes de programación.

- Por razones de seguridad, no es posible manejar ficheros en SWISH (la versión online de SWI-Prolog), por lo que para trabajar con ellos es necesario descargar e instalar la versión completa del intérprete en el ordenador.

MANEJO DE FICHEROS

- `open(+NombreFichero, +Modo, -Fichero)`
  Si `NombreFichero` es un nombre de fichero válido, abre el fichero correspondiente de acuerdo con el modo especificado por `Modo`, y unifica con `Fichero` el identificador del fichero abierto.
  Los valores para el argumento `Modo` pueden ser:
  - `read` para abrir el fichero en modo lectura.
  - `write` para abrir el fichero en modo escritura (si el fichero no existe, lo crea; si ya existe, su contenido se perderá).
  - `append` para abrir el fichero en modo escritura (si el fichero no existe, lo crea; si ya existe, las operaciones de escritura se realizarán al final del fichero).
- `close(+Fichero)`
  Cierra el fichero asociado con el identificador `Fichero`.
- `set_input(+Fich)` `set_output(+Fich)`
  Convierte al fichero con identificador `Fich` en el fichero de lectura (escritura) actual.

- `current_input(?Fich)` `current_output(?Fich)`
  `Fich` es el fichero de lectura (escritura) actual.

LECTURA/ESCRITURA

El fichero por defecto, tanto para lectura como para escritura, es la pantalla, cuyo identificador es user.

- `read(?Termino)` `read(+Fich, ?Termino)`
  Lee del fichero de lectura actual o de `Fich` (abierto en modo lectura) y unifica el resultado con `Termino`. En ficheros, el término debe acabar con un punto y un retorno de carro.
- `write(?Termino)` `write(+Fich, ?Termino)`
  Escribe el término `Termino` en el fichero de escritura actual o en `Fich` (previamente abierto en modo escritura).
- `nl` `nl(+Fich)`
  Escribe un retorno de carro en el fichero de escritura actual o en `Fich` (previamente abierto en modo escritura).

Ejemplo (Entrada/salida por pantalla)

```
% pide_numero(-X)

pide_numero(X) :-
    write('Introduzca un número: '),
    nl,
    read(X).

% escribe_cuadrado(+X)

escribe_cuadrado(X) :-
    X2 is X*X,
    write('El cuadrado de '),
    write(X),
    write(' es '),
    write(X2).
```

Ejemplo (Entrada/salida por pantalla/fichero)

```
% pide un número y lo escribe, junto con su cuadrado,
%  en un fichero que se crea a tal efecto

cuadrado :-
    pide_numero(X),
    open('c:/prolog/prueba.txt', write, Prueba),
    set_output(Prueba),
    escribe_cuadrado(X),
    close(Prueba),
    set_output(user),
    % user es el identificador de la pantalla
    write('el cuadrado se ha escrito en prueba.txt').
```

Ejercicios (Entrada/salida)

Ejercicio nº 6 de la Práctica de PROLOG nº2.

BIBLIOGRAFÍA

- L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.
- W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fifth edition, 2003.
- I. Bratko.*Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 2001.
- J. Lloyd. *Foundations of Logic Programming*, (Second Edition). Springer-Verlag, 1987.
- R. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
- U. Nilsson and J. Maluszynski. Logic, Programming and Prolog. John Wiley & Sons Ltd, 1996.
- SWI-Prolog, entorno de programación en Prolog de dominio público.
- comp.lang.prolog. Faq

PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL2: El lenguaje PROLOG, aspectos básicos

7. Manejo de listas

Grado en Ingeniería Informática

URJC

Ana Pradera

Contenido

1. LAS LISTAS EN PROLOG
   - Representación
   - Patrones
2. ALGUNOS PREDICADOS BÁSICOS SOBRE LISTAS
   - Pertenece (member en SWI-Prolog)
   - Longitud (length en SWI-Prolog)
   - Concatena (append en SWI-Prolog)
   - Último (last en SWI-Prolog)
   - Inversa (reverse en SWI-Prolog)
3. EJEMPLOS ADICIONALES
   - Escritura de una lista
   - Suma de los elementos de una lista
   - Implementación de autómatas finitos

LAS LISTAS EN PROLOG

Representación de listas en PROLOG

- Las listas de PROLOG son términos compuestos (estructuras) y constituyen la estructura de datos más utilizada.
- Sus elementos son términos (constantes, variables o términos compuestos), por lo que pueden ser, en particular, listas.
- Las listas no tienen por qué ser homogéneas: una misma lista puede contener términos de clases distintas.
- Toda lista es:
  - O bien vacía, y se representa por `[]`.
  - O bien no vacía, se representa como `[a1,a2,...,an]` y tiene:
    - una cabeza, que se corresponde con el primer *término* de la lista: `a1`.
    - un resto, que es la *lista* formada por todos sus términos salvo el primero: `[a2,...,an]` si `n>1` o `[]` en el caso `n=1`.

Patrones para listas

- El patrón `[C|R]` unifica con cualquier lista *no vacía* siempre que `C` unifique con la cabeza y `R` unifique con el resto de la lista.
- El patrón `[C1,C2|R]` unifica con cualquier lista que tenga *al menos dos elementos* siempre que `C1` unifique con el primer elemento de la lista, `C2` con el segundo, y `R` unifique con el resto de la lista.
- El patrón `[C1,C2,C3|R]` unifica con cualquier lista que tenga *al menos tres elementos* y tal que `C1,C2,C3` unifiquen con los tres primeros elementos de la lista y `R` unifique con el resto de la lista.
- El patrón `[C1,C2,C3,C4|R]` ...

Observe que lo que está a la derecha de la barra vertical `|` tiene que ser en todos los casos una *lista*.

Ejemplos (Listas y patrones, 1/2)

- Los términos `a`, `X`, `progenitor(a,X)`, `[a|b]` o `[a|X]` *no* son listas (los dos últimos debido a que lo que tienen a la derecha de la barra `|` no son listas).
- Los términos `[a,X]`, `[a,X|[]]`, `[a|[X]]` y `[a|[X|[]]]` son todos ellos listas representando una misma lista: la lista compuesta por los elementos `a` y `X`.
- La lista `[X1]` (`[X1,X2]`, ...) solo unificará con listas que contengan exactamente un (dos, ...) elemento.
- El patrón `[a|X]` solo unificará con listas con cabeza `a`, por lo que *no* es unificable por ejemplo con `a`, `[]`, `[b]` o `[b,a]`.
- El patrón `[a|X]` es unificable con cualquiera de las siguientes listas: `[a]` (mediante la sustitución `X/[]`), `[a,b,c]` (mediante la sustitución `X/[b,c]`), `[a,[b]]` (mediante la sustitución `X/[[b]]`), `[a|[b]]` (mediante la sustitución `X/[b]`), etc.

Ejemplos (Listas y patrones, 2/2)

```
?- [X,b] = [a|[B]].
X=a, B=b.
%[B] tiene que unificar con [b], posible con B=b.

?- [[a,b],c] = [X|Y].
X=[a,b], Y=[c].

?- [[X,f(b)],[c]] = [C|[c]].
false.
% la derecha de "|", [c], no unifica con el resto
%  de la lista de la izquierda, [[c]].

?- [[X,a],[c]] = [C|[[D]]].
C=[X,a], D=c.
%[[D]] tiene que unificar con [[c]], posible con D=c.
```

Ejercicios (Uso básico de listas)

1. *Ejercicio nº 1 de la Práctica de PROLOG nº 3.*
2. *Describa en lenguaje natural (con sus propias palabras) el resultado del siguiente predicado (cierto si ...):*

```
misterio1(E1, E2, [E1,E2|_]).

misterio1(E1, E2, [_|R]) :-
    misterio1(E1, E2, R).
```

3. *Describa en lenguaje natural (con sus propias palabras) el resultado del siguiente predicado (cierto si ...):*

```
misterio2(L1, L2) :-
    L1 = [C | _],
    L2 = [C, C | _].
```

Soluciones propuestas:

1. Ver soluciones comentadas en Práctica de PROLOG nº 3 con soluciones.

2. `misterio1(E1, E2, L)` es cierto si E1 y E2 son elementos consecutivos de la lista L que constituye el tercer argumento. Por ejemplo, `?- misterio1(b,c,[a,b,c]).` daía cierto, mientras que `?- misterio1(a,c,[a,b,c]).` daría falso.

3. `misterio2(L1, L2)` es cierto si L1 es una lista no vacía, L2 es una lista con al menos dos elementos y el primer elemento de L1 es unificable con los dos primeros elementos de L2. Por ejemplo, `?- misterio2([a,b], [a]).` daría falso, mientras que `?- misterio2([a,b], [a,a,c]).` daría cierto.

ALGUNOS PREDICADOS BÁSICOS SOBRE LISTAS

- La mayoría de los entornos de programación en PROLOG, entre ellos SWI-Prolog, ofrecen predicados predefinidos implementando las operaciones básicas para el manejo de listas.
- A pesar de ello, en lo que sigue se discute la implementación y el uso de algunos de ellos (pertenece, longitud, concatena, último e inversa), con el objeto de ilustrar el manejo típico de las listas en PROLOG (**uso de patrones + recursión**) y de destacar su **versatilidad** (predicados con distintos usos dependiendo de si se invocan con ciertos parámetros de entrada o de salida).
- Para comprender mejor el modo en el que PROLOG maneja las listas es muy recomendable *construir los Árboles de Resolución* correspondientes a algunas de las consultas que se proponen a continuación -u otras similares-, comprobando las soluciones obtenidas con las facilitadas por PROLOG.

```
pertenece(?E, ?L) (member en SWI-Prolog)
% cierto si E pertenece a L
```

- Si `L` es vacía, unificará con el término `[]`, y sea quien sea `E`, el resultado de "`?- pertenece(E, []).`" debe ser falso. Para ello *basta con no incluir en el programa ninguna cláusula cuya cabeza unifique con* `pertenece(E, []).`

- Si `L` no es vacía, tiene que unificar con el patrón `[C|R]`:
  - Caso base: cierto cuando el elemento buscado coincide con la cabeza de la lista, lo cual se escribe en PROLOG mediante el hecho

    ```
    pertenece(C, [C|_]).
    ```

  - Caso recursivo: cierto cuando el elemento buscado *pertenece* al resto de la lista, y esto último se escribe en PROLOG con la regla:

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

- En definitiva:

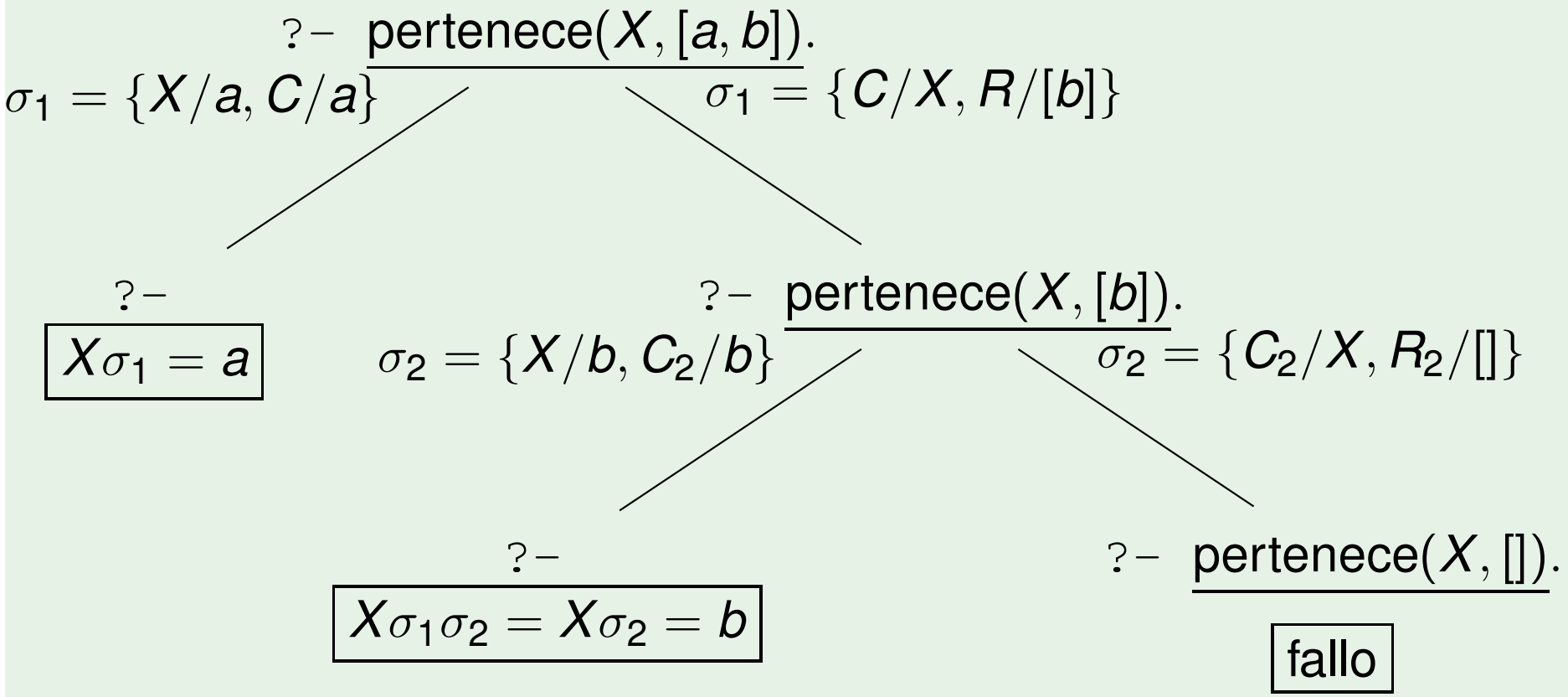
```
pertenece(?E, ?L) (member en SWI-Prolog)
% cierto si E pertenece a L

    pertenece(C,[C|_]).

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

Ejemplos (Algunos usos de pertenece (member), 1/2)

```
 ?- pertenece(b, [a,b,c]).
true

?- pertenece(a, L).          % infinitas soluciones
L = [a|_] ; L = [_, a|_] ; L = [_, _, a |_] ; ...
```

Ejemplos (Algunos usos de pertenece (member), 2/2)

```
% recorrido de una lista mediante backtraking
?- pertenece(X, [a,b]).
X = a ; X = b
```



Observación (Versatilidad de `pertenece`)

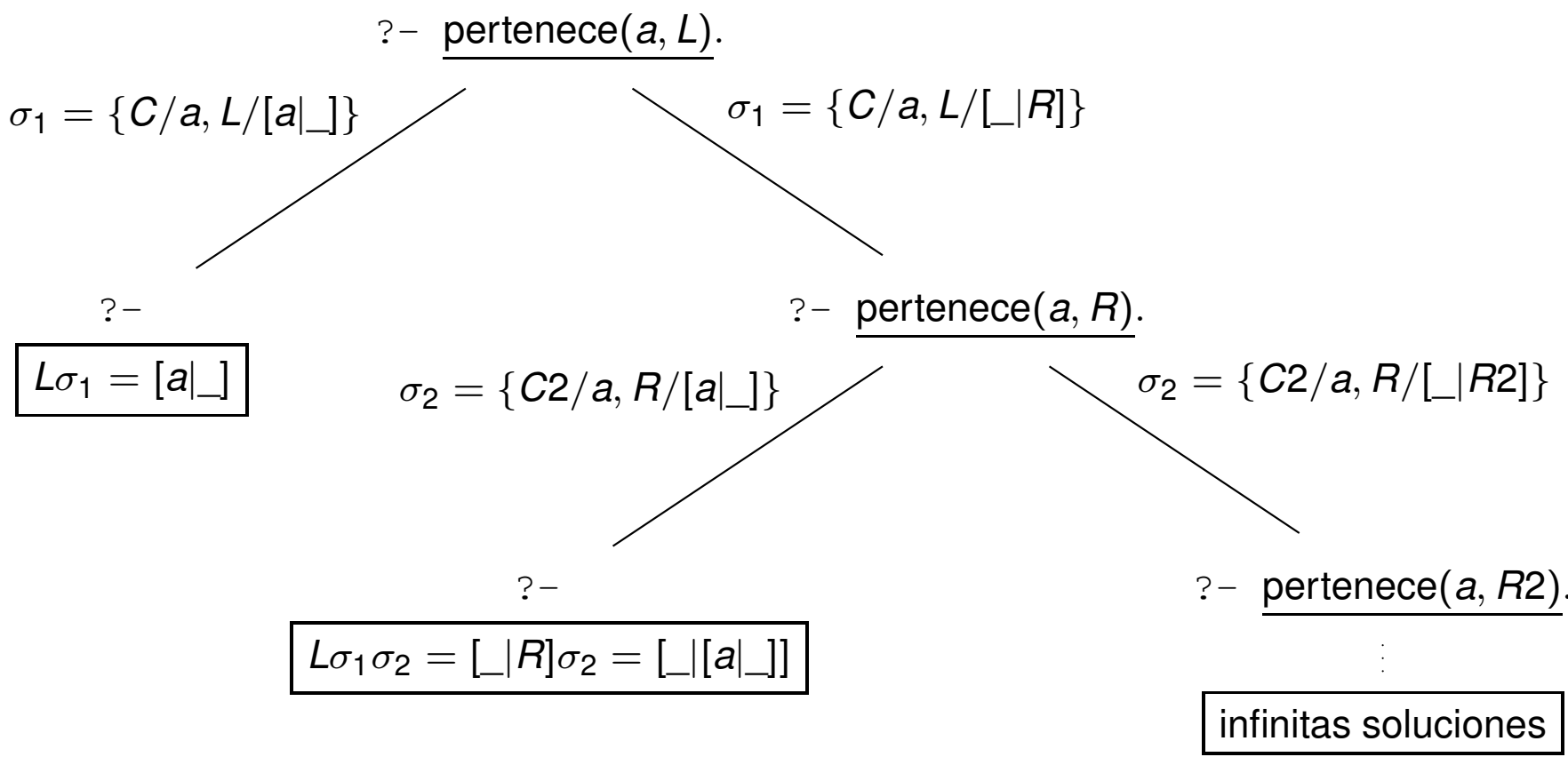
El predicado `pertenece` es un predicado versátil, que sirve para: (1) averiguar si un elemento pertenece a una lista (1er ejemplo anterior) (2) generar listas que contengan un elemento dado (2do ejemplo) (3) recorrer una lista utilizando backtracking (3er ejemplo).

Ejercicios (Predicado `pertenece`)

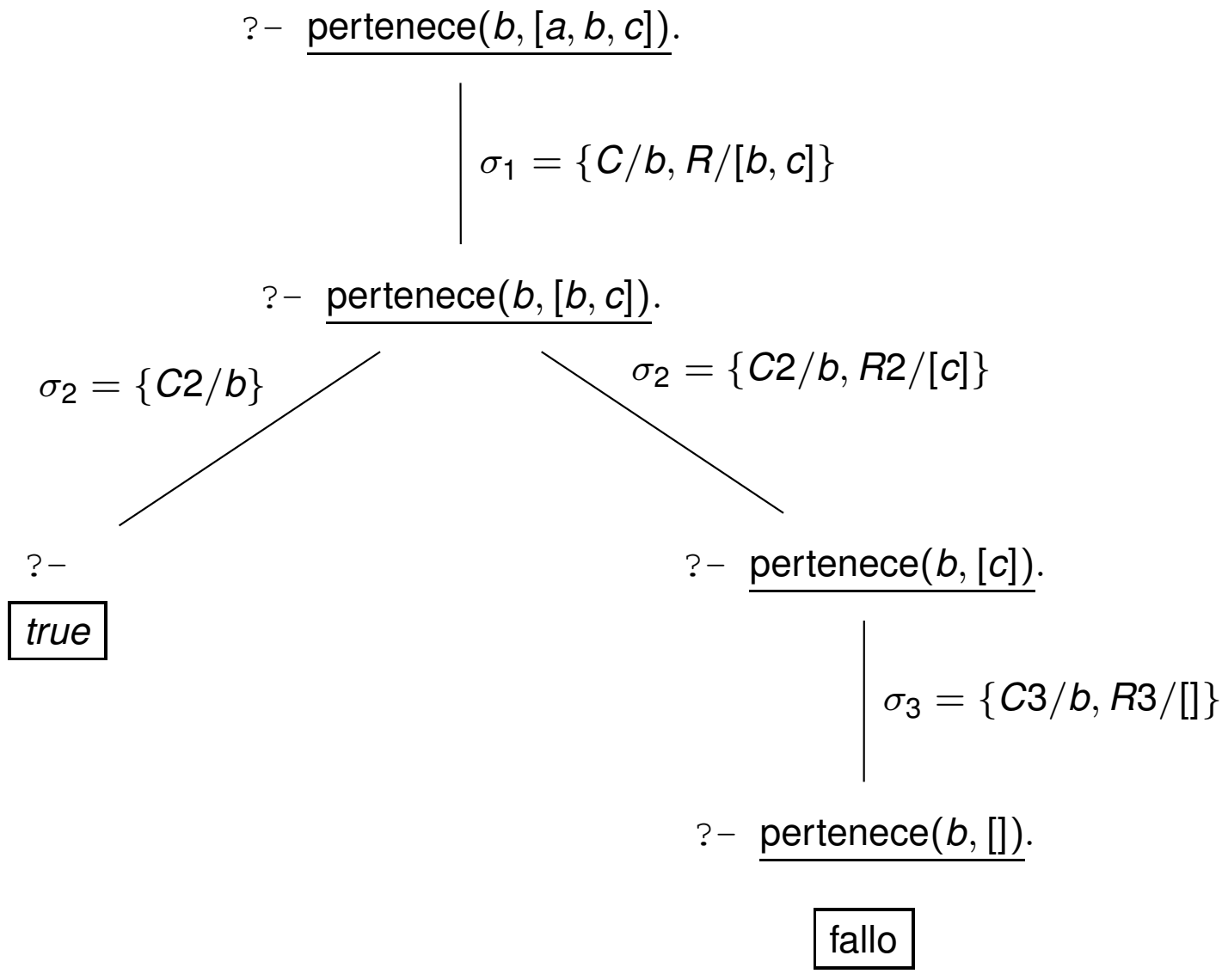
Dado el código facilitado más arriba para el predicado `pertenece`*, construya los Árboles de Resolución correspondientes e indique qué respuestas facilitaría* PROLOG*, y en qué orden, ante las siguientes consultas:*

1. `?- pertenece(a, L).`
2. `?- pertenece(b, [a,b,c]).`
3. `?- pertenece(X, [[1,2]]), pertenece(Y, X).`

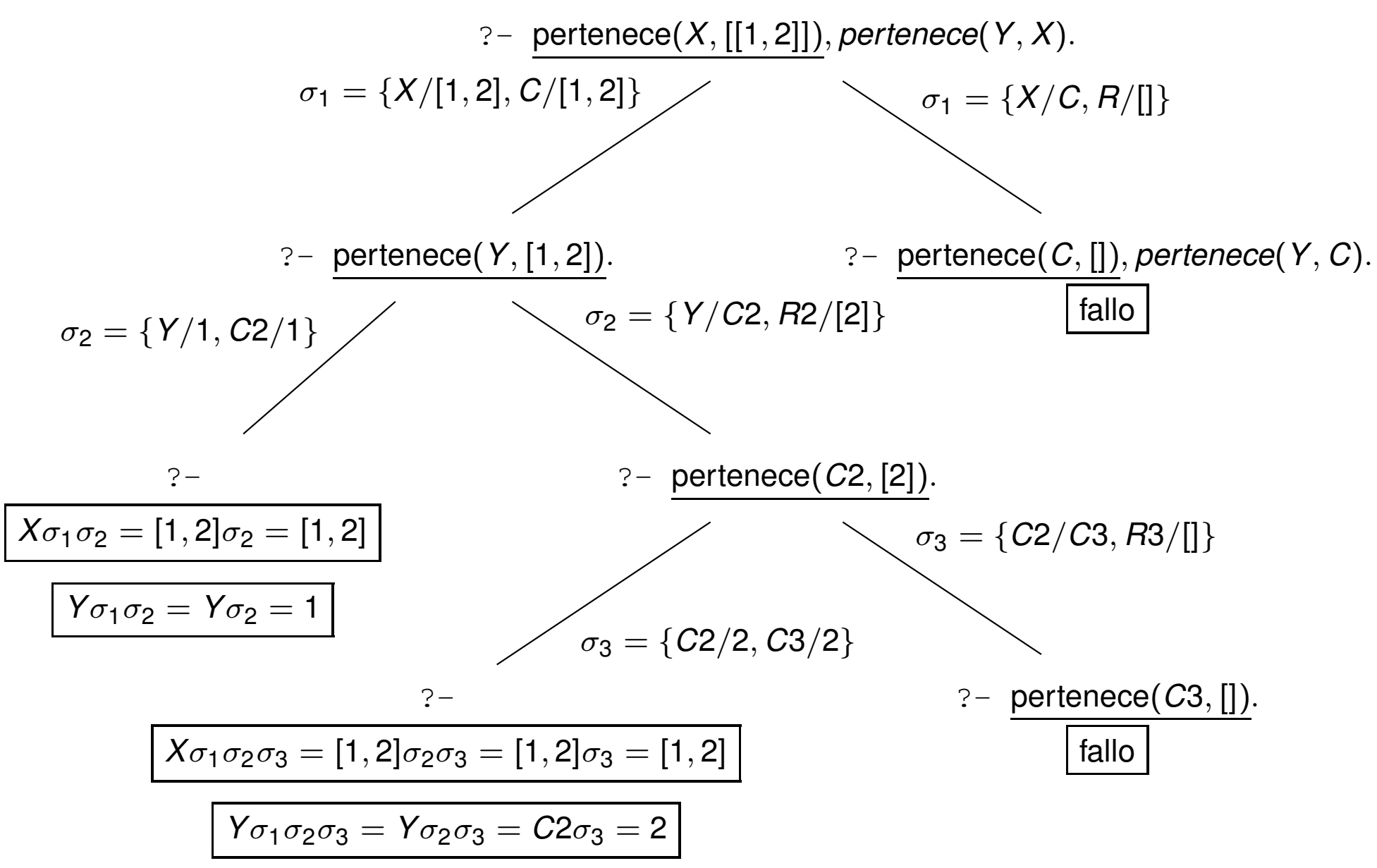
Soluciones propuestas:



Se trata de un Árbol de Resolución infinito en el que van apareciendo sucesivamente todas las posibles listas que contienen `a`: aquellas en las que `a` está en primera posición (`[a|_]`), aquellas en las que `a` está en segunda posición (recuerde que `[_|[a|_]]` no es otra cosa que `[_, a|_]]`), en tercera, `[_,_,a|_]`, etc.



A la vista del Árbol anterior, la respuesta de PROLOG será `true` (y a continuación `false` indicando que no hay más soluciones).



Primera solución `X=[1,2],Y=1`, segunda solución `X=[1,2],Y=2` (y a continuación `false` indicando que no hay más soluciones).

```
longitud(?L, ?N) (length en SWI-Prolog)
% cierto si L tiene N elementos
    longitud([], 0).
    longitud([_|R], N) :-
        longitud(R, LR),
        N is LR + 1.
```

Ejemplos (Algunos usos de longitud (length))

```
 ?- longitud([[a,b,c,d],1], X).
X = 2.

?- longitud(L, Long).
L = [], Long = 0 ;
L = [_], Long = 1 ;
L = [_, _], Long = 2;
..... (infinitas soluciones)
```

$$?- \underline{\text{longitud}([[a,b,c,d],1],X)}.$$

$$\Big| \quad C2 \quad \sigma_1 = \{R/[1], X/N\}$$

$$?- \underline{\text{longitud}([1],LR)}, N \text{ is } LR+1.$$

$$\Big| \quad C2 \quad \sigma_2 = \{R2/[], LR/N2\}$$

$$?- \underline{\text{longitud}([],LR2)}, N2 \text{ is } LR2+1, N \text{ is } N2+1.$$

$$\Big| \quad C1 \quad \sigma_3 = \{LR2/0\}$$

$$?- \underline{N2 \text{ is } 0+1}, N \text{ is } N2+1.$$

$$\Big| \quad \sigma_4 = \{N2/1\}$$

$$?- \underline{N \text{ is } 1+1}.$$

$$\Big| \quad \sigma_5 = \{N/2\}$$

$$?-$$

$$\boxed{\underline{X}\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5 = \underline{N}\sigma_2\sigma_3\sigma_4\sigma_5 = \ldots = N\sigma_5 = 2}$$

Observación

- Observe que en el caso recursivo de la implementación de `longitud` es necesario sumar 1 a la longitud del resto de la lista, para lo cual es imprescindible:
  1. Usar el predicado aritmético `is`, el único capaz de *evaluar* una expresión aritmética y asignar su valor a una variable mediante unificación (no se puede poner simplemente `LR + 1` en la cabeza de la regla).
  2. Hacer primero el cálculo recursivo y *después* la evaluación mediante `is` (de lo contrario `is` produciría un error de instanciación), por lo que la recursión es no final.
- El uso de `longitud` con el primer parámetro de salida y el segundo de entrada (por ejemplo `?- longitud(L,2)`) dará una primera solución y luego entrará en una rama infinita sin soluciones (piense en cómo sería el Árbol de Resolución correspondiente).

La implementación anterior es ineficiente puesto que la recursión no es final. Una implementación alternativa, con recursión final obtenida mediante la introducción de un parámetro de acumulación, sería:

```
%   longitud_rc(?L, ?N}
%   cierto si L tiene N elementos
%   Implementación con recursión de cola

% se añade parámetro de acumulación
longitud_rc(L, N) :-
     longitud(L, 0, N). % 0 = valor caso base

% caso base: se devuelve lo acumulado
longitud([], Ac, Ac).
% caso recursivo: se actualiza el parámetro
longitud([_|R], Ac, N) :-
     NAc is Ac + 1,
     longitud(R, NAc, N).
```

Ejercicios (Predicado `longitud`)

1. *Construya el Árbol de Resolución para la consulta*

   ```
   ?- longitud_rc([[a,b,c,d],1],X), Z is X*X.
   ```

   (implementación con recursión de cola discutida más arriba).

   ¿Qué respuestas ofrecería PROLOG*, y en qué orden?*

2. *Construya el Árbol de Resolución para la consulta*

   ```
   ?- longitud(L, X), X >= 1.
   ```

   ¿Qué respuestas ofrecería PROLOG*, y en qué orden?*

Soluciones propuestas:

?- $\underline{\text{longitud\_rc}([[a,b,c,d],1],X)}, Z \text{ is } X * X.$

$\Big| \ \sigma_1 = \{X/N\}$

?- $\underline{\text{longitud}([[a,b,c,d],1],0,N)}, Z \text{ is } N * N.$

$\Big| \ \sigma_2 = \{R2/[1], A2/0, N2/N\}$

?- $\underline{NA2 \text{ is } 0+1}, \text{longitud}([1], NA2, N), Z \text{ is } N * N.$

$\Big| \ \sigma_3 = \{NA2/1\}$

?- $\underline{\text{longitud}([1],1,N)}, Z \text{ is } N * N.$

$\Big| \ \sigma_4 = \{R4/[], A4/1, N4/N\}$

?- $\underline{N4 \text{ is } 1+1}, \text{longitud}([], NA4, N), Z \text{ is } N * N.$

$\Big| \ \sigma_5 = \{NA4/2\}$

?- $\underline{\text{longitud}([],2,N)}, Z \text{ is } N * N.$

$\Big| \ \sigma_6 = \{N6/2, N/2\}$

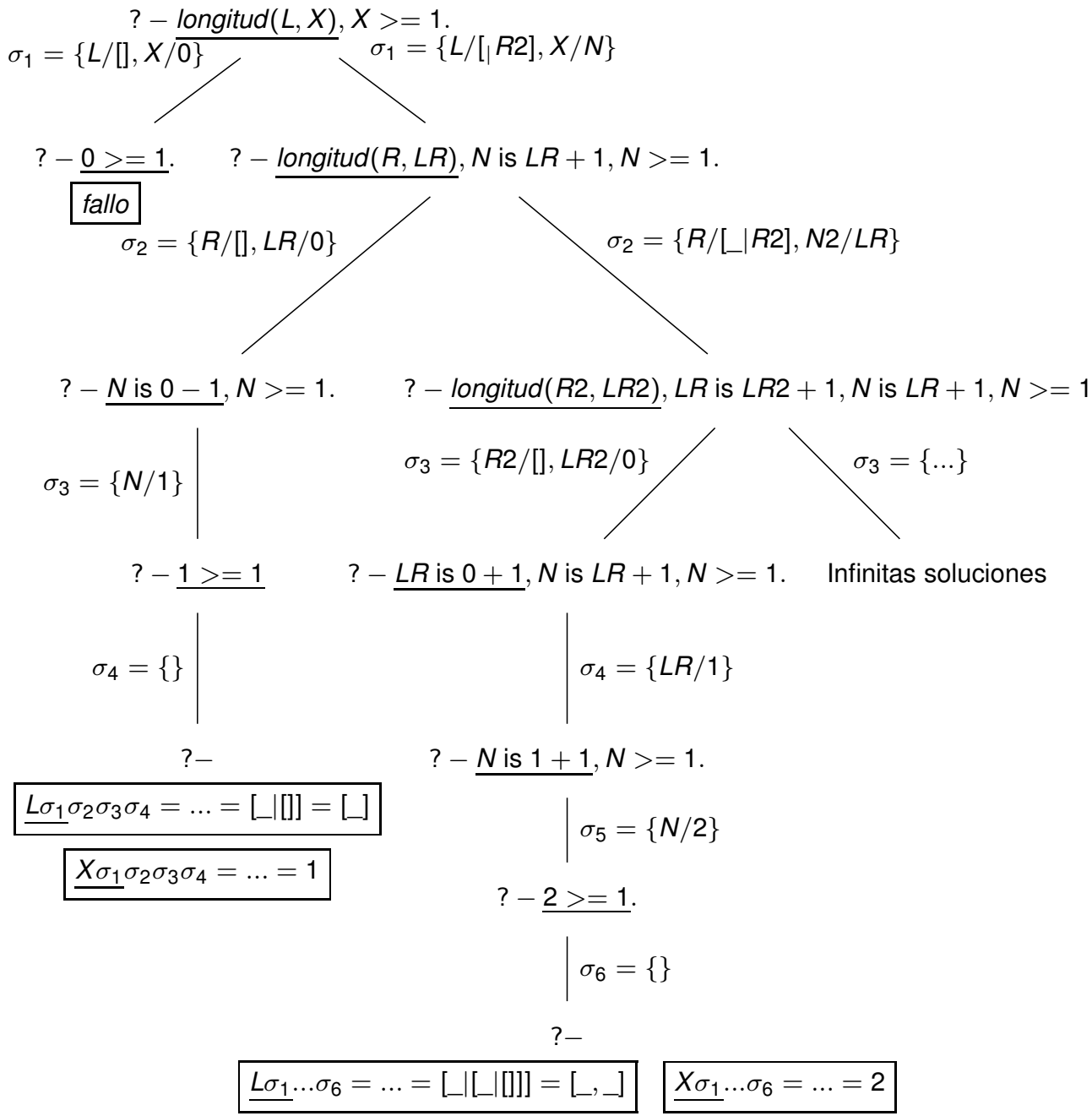
?- $\underline{Z \text{ is } 2*2.}$

$\Big| \ \sigma_7 = \{Z/4\}$

?-

$\boxed{\underline{X\sigma_1}\sigma_2\sigma_3\sigma_4\sigma_5\sigma_6\sigma_7 = \underline{N\sigma_2}\sigma_3\sigma_4\sigma_5\sigma_6\sigma_7 = ... = N\sigma_7 = 2}$ $\boxed{\underline{Z\sigma_1}\sigma_2\sigma_3\sigma_4\sigma_5\sigma_6\sigma_7 = ... = Z\sigma_7 = 4}$

Respuesta: `X=2, Z=4` (y `false` indicando que no hay más soluciones)



Respuestas: 1) `L=[_], X=1` 2) `L=[_, _], X=2`, 3) `L=[_, _, _], X=3`, etc (infinitas sols)

```
concatena(?L1, ?L2, ?L) (append en SWI-Prolog)
% cierto si L es la concatenación de L1 y L2
```

- Caso base: si `L1` es vacía, su concatenación con cualquier otra lista (vacía o no vacía) es esta última, lo cual se escribe en PROLOG mediante el hecho

```
concatena([], L, L).
```

- Caso recursivo: si `L1` no es vacía tiene que unificar con `[C|R]`, y su concatenación con una lista cualquiera `L` será `[C|NL]`, siempre y cuando `NL` sea la concatenación de `R` con `L`, lo cual se escribe en PROLOG mediante la siguiente regla recursiva:

```
concatena([C|R], L, [C|NL]) :-
    concatena(R, L, NL).
```

- En definitiva:

```
concatena(?L1, ?L2, ?L) (append en SWI-Prolog)
% cierto si L es la concatenación de L1 y L2

    concatena([], L, L).
    concatena([C|R], L, [C|NL]) :-
        concatena(R, L, NL).
```

Observación

Note que la recursión de la implementación anterior no es exactamente de cola, puesto que después de la llamada recursiva queda aún una operación por hacer (añadir el elemento C delante de la lista NL). Esta recursión se denomina recursión de cola módulo cons y, a diferencia de lo que ocurre en otros lenguajes, **PROLOG es capaz de optimizar el código** resultante de forma similar al de la recursión de cola normal.

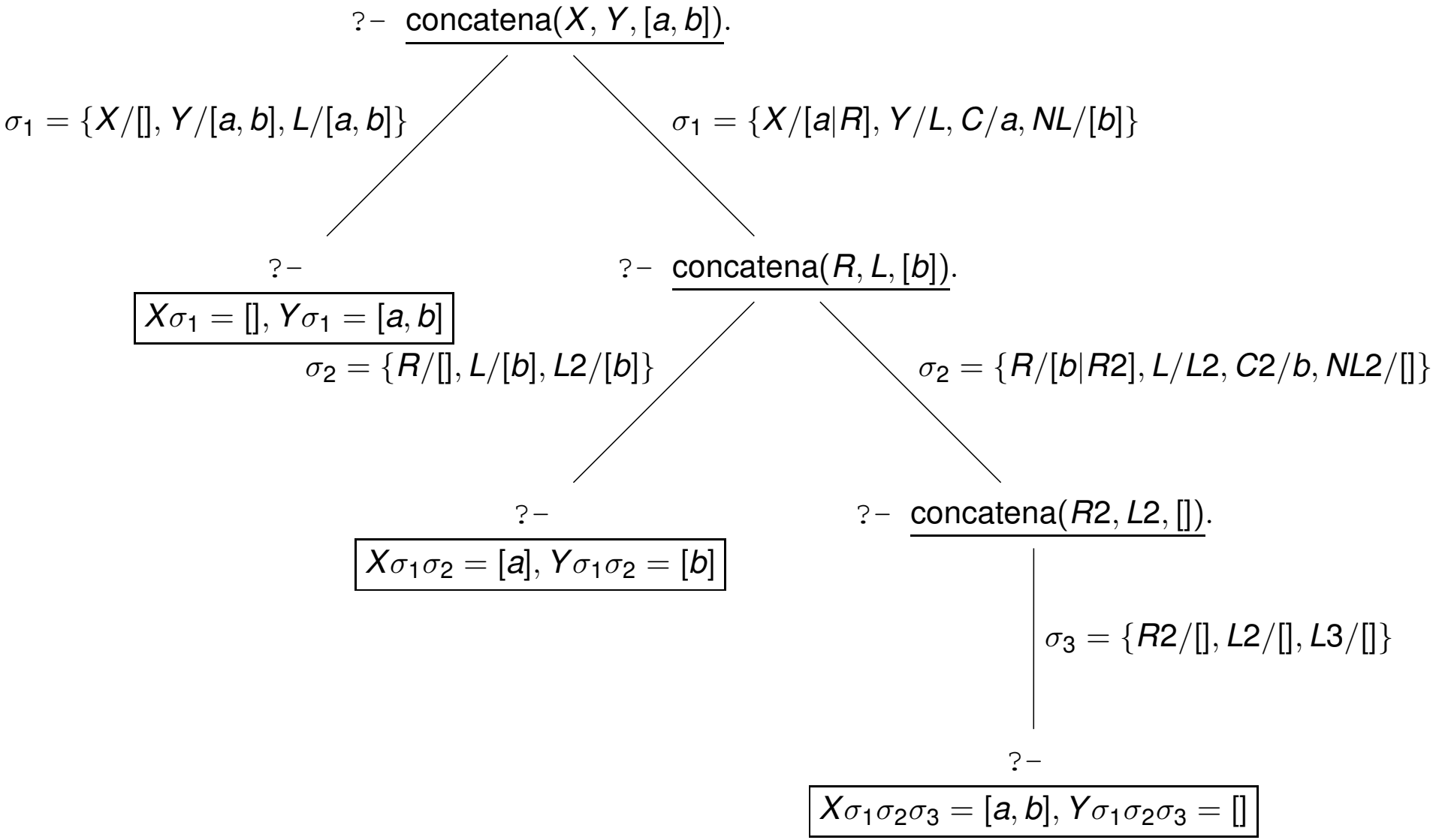
Observación (Versatilidad de `concatena`)

Los ejemplos incluidos a continuación ilustran cómo `concatena`, a pesar de la sencillez de su implementación, es muy flexible y versátil, pudiéndose usar para varios cometidos distintos:

1. Para **concatenar** listas.
2. Para calcular **prefijos**.
3. Para calcular **sufijos**.
4. Para **descomponer** una lista en dos sublistas de todas las formas posibles (mediante backtracking).

Ejemplos (Algunos usos de concatena (append))

```
?- concatena([a], [b,c], L).     % concatenación
L = [a, b, c].

?- concatena([a], L, [a,b,c]).    % sufijo
L = [b, c].

?- concatena(L, [b,c], [a,b,c]). % prefijo
L = [a].

?- concatena(X, Y, [a,b,c]).     % descomposición
                                 % en dos sublistas
X = [], Y = [a, b, c] ;          % (con backtracking)
X = [a], Y = [b, c] ;
X = [a, b], Y = [c] ;
X = [a, b, c], Y = []
```



Cuidado con el cálculo correcto de los u.m.g.'s: por ejemplo, el primer u.m.g. de la rama de la derecha es $\{X/[a|R], Y/L, C/a, NL/[b]\}$ y no $\{X/[C|R], Y/L, C/a, NL/[b]\}$, puesto que cuando se elige la sustitución $\{C/a\}$, esta se debe **componer** con lo que había hasta el momento, $\{X/[C|R], Y/L\}$.

Observación (Uso de concatena -append- para implementar otros predicados)

La capacidad del predicado de concatenación para dividir una lista en dos sublistas lo convierte en una herramienta muy útil para la implementación de otros predicados que se puedan describir mediante esta división. Por ejemplo:

- El predicado `pertenece(E,L)` se puede también implementar mediante `concatena`: será cierto si L se pueda dividir en dos trozos de forma que el segundo de ellos empiece por E:

```
pertenece(E,L) :- concatena(_, [E|_], L).
```

- El predicado `prefijo(Pre,L)`, cierto si la lista P es un prefijo de L, será cierto si existe una lista tal que concatenada con Pre da L:

```
prefijo(Pre,L) :- concatena(Pre, _, L).
```

Ejercicios (Predicado `concatena`)

1. *Razone qué ocurrirá (se producirá un error, la respuesta será false, true, se producirá una computación infinita, la o las respuestas serán ....) al ejecutar la consulta:*
   ```
   ?- concatena(X,_,[a,b]), longitud(X,_N), _N =< 1.
   ```
2. *Implemente de forma recursiva (sin utilizar concatena) el predicado* `prefijo(Pre,L)` *mencionado previamente.*
3. *Implemente el predicado* `sufijo(Suf,L)`*, cierto si la lista Suf es un sufijo de L, tanto de forma recursiva como mediante el predicado concatena.*
4. *Describa en lenguaje natural (con sus propias palabras) el resultado (cierto si ...) del predicado*
   ```
   misterio([A|B]) :- concatena(_, [A], [A|B]).
   ```
5. *Construya el Árbol asociado a* `?- pertenece(b,[a,b])` *suponiendo que* `pertenece` *está implementado por medio de* `concatena` *(ver última observación).*

Soluciones propuestas:

1. La consulta se interesa por los prefijos X de la lista [a,b] con una longitud (cuyo valor no interesa puesto que la variable es semianónima) menor o igual que 1. PROLOG construye el correspondiente Árbol de Resolución en profundidad con backtracking (se recomienda su construcción) por lo que las respuestas serán, en este orden, X=[] y X=[a] (y false indicando que no hay más soluciones). Las respuestas no ofrecen valores ni para la variable anónima _ ni para la semianónima "_N".

2.

```
% prefijo(?Pre, ?L), cierto si la lista Pre es un prefijo de L
prefijoR([], _).
prefijoR([C|R1], [C|R2]) :- prefijoR(R1,R2).
```

3.

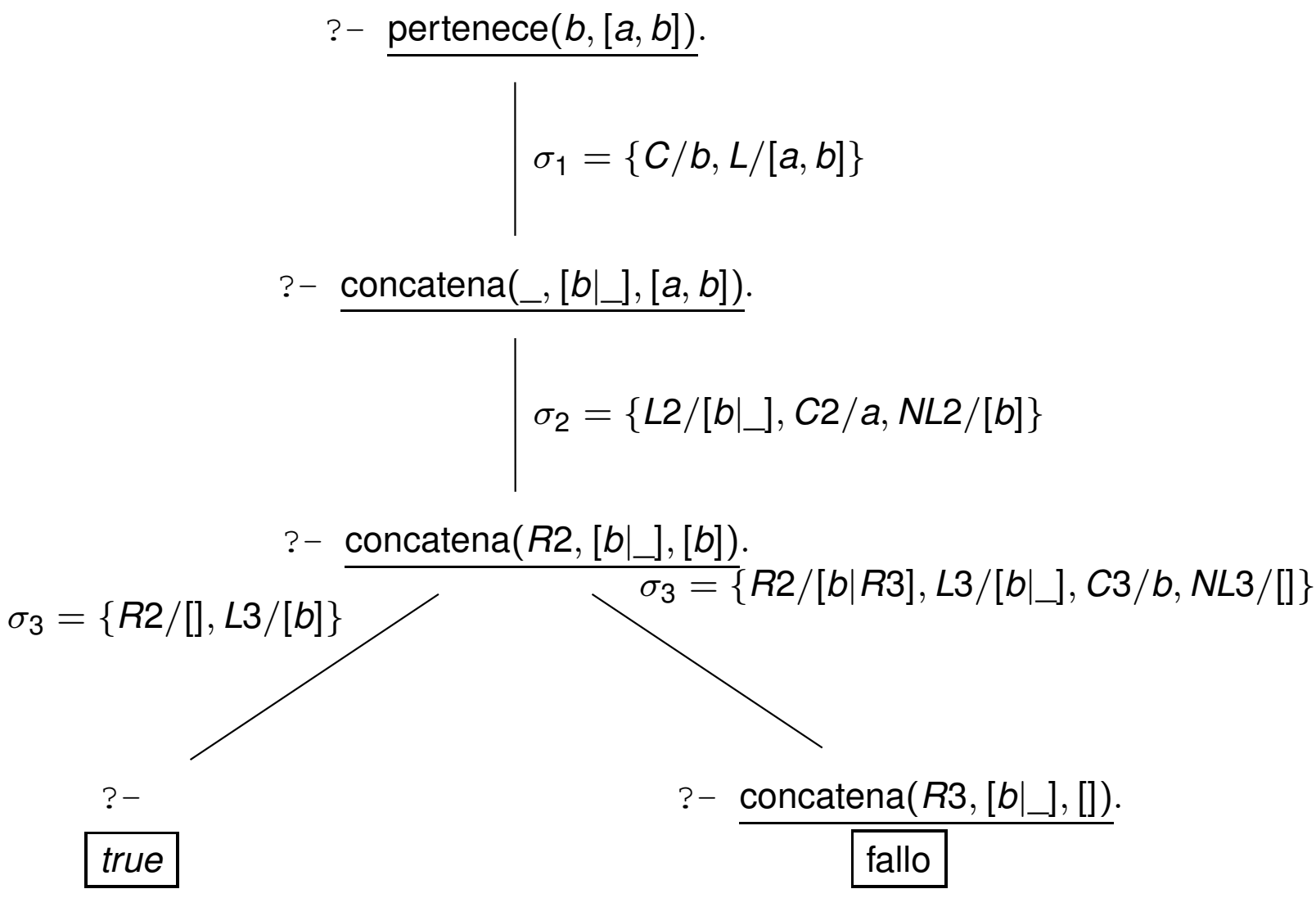
```
% sufijo(?Suf, ?L), cierto si la lista Suf es un sufijo de L
% implementación no recursiva
sufijo(Suf,L) :- concatena(_, Suf, L).
% implementación recursiva
sufijoR(L, L).
sufijoR(L, [_|R]) :- sufijoR(L,R).
```

4.

```
misterio([A|B]) :- concatena(_, [A], [A|B]).
```

El predicado misterio(?L) es cierto si L es una lista no vacía en la que sus elementos primero y último son unificables. En efecto, *append(_, [A], [A|B])* será cierto si la lista [A|B] se puede descomponer en dos, un prefijo (no importa cuál, de ahí el uso de la variable anónima) y un sufijo que consta exclusivamente de un elemento, la variable A, la misma que encabeza la lista [A|B].

5.



Respuesta: true (y false indicando que no hay más soluciones).

```
% ultimo(?L, ?U) (last en SWI-Prolog)
% cierto si U es el último elemento de la lista L

    ultimo([C],C).
    ultimo([_|R],U) :-
        ultimo(R,U).
```

Ejemplos (Algunos usos de ultimo (last))

```
 ?- ultimo([a,b,c], c).
true
 ?- ultimo([a,b,c], U).
U = c
 ?- ultimo(L, c).
L = [c]
L = [_, c]
... (infinitas soluciones)
```

```
% inversa(+L,?LI) (reverse en SWI-Prolog)
% cierto si LI es la inversa de L

    inversa([], []).

    inversa([C|R], LI) :-
       inversa(R, RI),
       append(RI, [C], LI).
```

Ejemplos (Algunos usos de inversa (reverse))

```
 ?- inversa([], []).
true
 ?- inversa([a,b,c], [a,b,c]).
false
 ?- inversa([a,b,c], I).
I = [c,b,a]
```

La implementación anterior del predicado `inversa` es ineficiente debido a que la recursión no es de cola. Una implementación alternativa, con recursión de cola obtenida introduciendo un parámetro de acumulación:

```
% inversa_rc(+L,?LI)
% cierto si LI es la inversa de L
% Implementación con recursión de cola.

% se añade parámetro de acumulación
  inversa_rc(L, LI) :-
     inversa(L, [], LI). % [] = valor caso base
% caso base: se devuelve lo acumulado
    inversa([], Ac, Ac).
% caso recursivo: se actualiza el parámetro
    inversa([C|R], Ac, LI) :-
     inversa(R, [C|Ac], LI).
```

Ejercicios (Predicados básicos sobre listas)

1. *Ejercicio nº 2 de la Práctica de* PROLOG *nº 3 ignorando las preguntas en las que aparece el predicado de corte* `!`.
2. *Dada la consulta* `?- concatena(X,Y,[a]), pertenece(a,X)`, *describa qué se pretende averiguar con ella, construya el Árbol de Resolución correspondiente e indique qué respuestas ofrece* PROLOG.
3. *Implemente el predicado* `ultimo` *de forma no recursiva mediante el uso del predicado de concatenación.*
4. *Implemente de dos formas distintas (una recursiva, la otra usando* `concatena`*) el predicado* `insertarfinal(+L,+E,?NL)`, *cierto si NL es la lista resultante después de insertar el elemento E al final de la lista L.*
5. *Construya los Árboles de Resolución asociados a algunas de las consultas planteadas en los ejemplos de uso de los predicados básicos sobre listas.*

Soluciones propuestas:

1. Ver soluciones comentadas en Práctica de PROLOG nº 3 con soluciones.

2. Ver vídeo Ejemplo Árbol Resolución Listas.

3.

ultimo(L,U) :- concatena(_, [U], L).

4. Implementación recursiva:

insertarfinalR([],E,[E]).
insertarfinalR([C|R], E, [C|NR]) :- insertarfinalR(R,E,NR).

Implementación no recursiva:

insertarfinal(L, E, NL) :- concatena(L, [E], NL).

EJEMPLOS ADICIONALES

Escritura de una lista

- Con el formato `[a1,a2,..,an]`

```
?- read(L), write(L).
|: [a,b,c]. % lista introducida, acabada en punto
[a,b,c]
L = [a, b, c].
```

- Un elemento por línea (la primera cláusula es necesaria para que el predicado devuelva 'true').

```
imprime_lista([]).
imprime_lista([C|R]) :-
    write(C),
    nl, % new line
    imprime_lista(R).
```

Suma de los elementos de una lista

```
% sumalista(+L,?S) (sumlist en SWI-Prolog)
% cierto si S es la suma de los elementos de
% la lista L. S=0 si L=[]
% implementación ineficiente (recursión no de cola)

sumalista([], 0).
sumalista([C|R], S) :-
    sumalista(R, SR),
    S is C + SR.
```

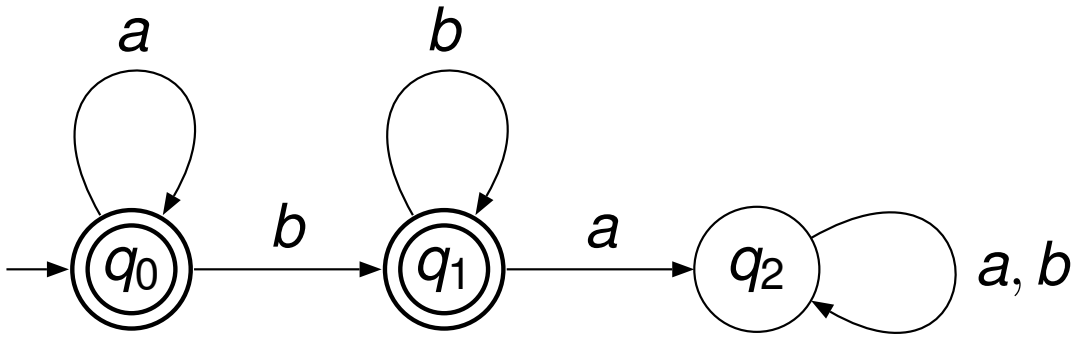
Ejemplos (Algunos usos de sumalista (sumlist))

```
?- sumalista([], 0).                    true
?- sumlista([1,3.5,-1], X).             X = 3.5
?- sumalista([1,3.5,c], X).             ERROR ....
```

Implementación eficiente, con recursión de cola obtenida introduciendo un parámetro de acumulación:

```
% sumalista(+L,?S) (sumlist en SWI-Prolog)
% cierto si S es suma de los elementos de
% la lista L. S=0 si L=[]

% se añade parámetro de acumulación
sumalistaRC(L, S) :-
    sumalista(L, 0, S). % 0 = valor caso base

% caso base: se devuelve lo acumulado
sumalista([], Ac, Ac).

% caso recursivo: se actualiza lo acumulado
sumalista([C|R], Ac, S) :-
    NAc is Ac + C,
    sumalista(R, NAc, S).
```

Implementación de autómatas finitos deterministas (AFDs)



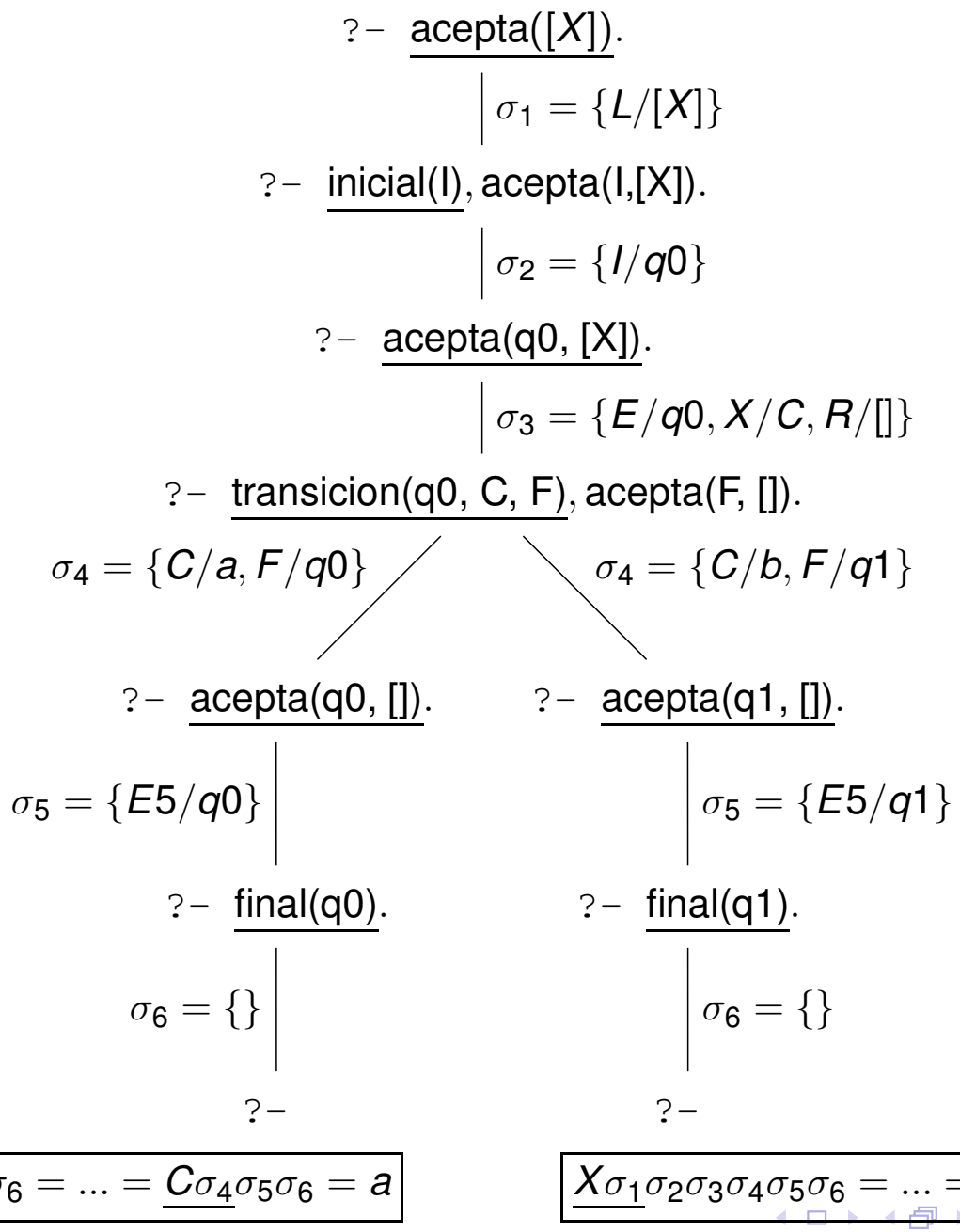
$$L(A) = \{a^n b^m : n, m \geq 0\}$$

```
%% DATOS DEL AFD CONCRETO
% inicial(?E), cierto si E es el estado inicial
   inicial(q0).

% final(?E), cierto si E es un estado final
  final(q0).
  final(q1).

% transicion(?E1, ?S, ?E2), cierto si el autómata
% transita de 'E1' a 'E2' leyendo el símbolo 'S'
  transicion(q0, a, q0).
  transicion(q0, b, q1).
  transicion(q1, a, q2).
  transicion(q1, b, q1).
  transicion(q2, a, q2).
  transicion(q2, b, q2).
```

```
%%%% FUNCIONAMIENTO DE UN AFD CUALQUIERA
% acepta(?L)
% cierto si el autómata acepta la palabra 'L'
% (representada mediante una lista de símbolos)
    acepta(L) :-
        inicial(I),
        acepta(I, L).

% acepta(?E, ?L)
% cierto si el autómata, partiendo del estado E,
% acepta la palabra 'L'

    acepta(E, []) :-
        final(E).
    acepta(E, [C|R]) :-
        transicion(E, C, F),
        acepta(F, R).
```

Ejemplos (Algunos usos de `acepta`)

```
?- acepta([]).
true
?- acepta([a]).
true
?- acepta([b]).
true
?- acepta([b,a]).
false
?- acepta([a,b]).
true
?- acepta([X]). % palabras aceptadas de longitud 1
X = a ;
X = b
```


?- acepta([X]).
σ1 = {L/[X]}
?- inicial(I), acepta(I,[X]).
σ2 = {I/q0}
?- acepta(q0, [X]).
σ3 = {E/q0, X/C, R/[]}
?- transicion(q0, C, F), acepta(F, []).
σ4 = {C/a, F/q0}
σ4 = {C/b, F/q1}
?- acepta(q0, []).
?- acepta(q1, []).
σ5 = {E5/q0}
σ5 = {E5/q1}
?- final(q0).
?- final(q1).
σ6 = {}
σ6 = {}
?-
?-
Xσ1σ2σ3σ4σ5σ6 = ... = Cσ4σ5σ6 = a
Xσ1σ2σ3σ4σ5σ6 = ... = Cσ4σ5σ6 = b

Ejercicios (Manejo de listas, 1/2)

1. *Modifique las dos implementaciones propuestas para* `sumalista` *de modo que el predicado falle, en lugar de ser cierto con S=0, en el caso de que la lista sea vacía.*
2. *Escriba la consulta adecuada para calcular qué palabras de longitud dos acepta el autómata implementado más arriba y construya el Árbol de Resolución asociado para averiguar qué respuestas ofrece* PROLOG*, y en qué orden las da.*
3. *Haga el ejercicio nº 3 de la Práctica de* PROLOG *nº 3 ignorando lo relacionado con el predicado de corte* `!`*.*
4. *Modifique el programa para AFDs discutido más arriba para que sea capaz de simular autómatas finitos no deterministas (AFNDs). Recuerde que las palabras aceptadas por un AFND son aquellas para las que el autómata puede transitar desde el estado inicial hasta algún estado final, pudiendo usar transiciones vacías (transiciones λ). Ver ejemplo a continuación.*

Soluciones propuestas:

1.
Implementación sin recursión de cola:

sumalista([C], C).
sumalista([C|R], S) :- sumalista(R, SR), S is C + SR.

Implementación mediante recursión de cola:

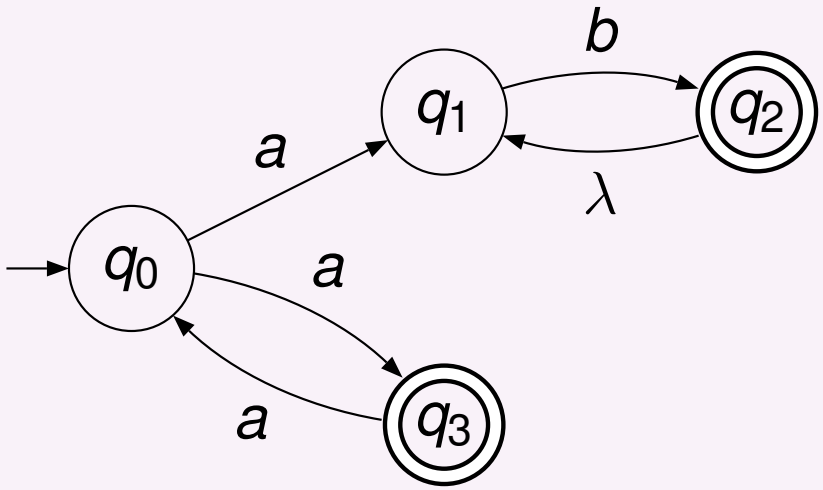
sumalistaRC(L, S) :- sumalista(L, 0, S).
sumalista([C], Ac, S) :- S is Ac + C.
sumalista([C|R], Ac, S) :- NAc is Ac + C, sumalista(R, NAc, S).

2. Para averiguar qué palabras de longitud dos acepta el autómata habría que hacer una consulta con una lista de tamaño dos, por ejemplo `?- acepta([Uno, Dos]).`

3. Ver soluciones comentadas en Práctica de PROLOG nº 3 con soluciones.

Ejercicios (Manejo de listas, 2/2)

El siguiente grafo representa un AFND que acepta todas las palabras de la forma $a^n b^m$ tales que n MOD 2 $\neq 0, m \geq 0$, y ninguna otra:



Para adaptar el programa de AFDs a AFNDs, tendrá que decidir cómo representar las transiciones λ y cómo permitir movimientos mediante esas transiciones. Pruebe su programa con ejemplos significativos, y haga el árbol de Resolución necesario para averiguar qué palabras de longitud 3 acepta el AFND anterior.

Soluciones propuestas:

% DATOS DE UN AFND CONCRETO (grafo anterior)

```
% inicial(?E), cierto si E es el estado inicial
inicial(q0).
% final(?E), cierto si E es un estado final
final(q2).
final(q3).
% transicion(?E1, ?S, ?E2), cierto si el Autómata transita del
% estado 'E1' al 'E2' leyendo el símbolo 'S'
transicion(q0, a, q1).
transicion(q0, a, q3).
transicion(q1, b, q2).
transicion(q3, a, q0).
% lambda(?E1, ?E2)
% cierto si el autómata tiene una transición lambda de E1 a E2
lambda(q2, q1).
```

% FUNCIONAMIENTO DE UN AFND CUALQUIERA

```
% acepta(?L)
% cierto si el autómata acepta la palabra 'L'
acepta(L) :- inicial(I), acepta(I, L).
% acepta(?E, ?L)
% cierto si el autómata, partiendo del estado E, acepta la palabra 'L'
acepta(E, []) :- final(E).
acepta(E, [C|R]) :- transicion(E, C, F), acepta(F, R).
acepta(E, L) :- lambda(E, F), acepta(F, L).
```

BIBLIOGRAFÍA

- L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.
- W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fifth edition, 2003.
- I. Bratko.*Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 2001.
- J. Lloyd. *Foundations of Logic Programming*, (Second Edition). Springer-Verlag, 1987.
- R. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
- U. Nilsson and J. Maluszynski. Logic, Programming and Prolog. John Wiley & Sons Ltd, 1996.
- SWI-Prolog, entorno de programación en Prolog de dominio público.
- comp.lang.prolog. Faq

PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL2: El lenguaje PROLOG, aspectos básicos

8. El predicado de corte

Grado en Ingeniería Informática

URJC

Ana Pradera

Contenido

1. DEFINICIÓN, EFECTOS Y PROPIEDADES
   - Definición
   - Efectos
   - Propiedades

2. USO PARA SIMULACIÓN DE ESTRUCTURAS CONDICIONALES
   - Cálculo de una función definida a trozos
   - Cálculo del máximo de dos números
   - Pertenencia de un elemento a una lista
   - Eliminación de la primera ocurrencia de un elemento en una lista
   - Ejercicios

DEFINICIÓN, EFECTOS Y PROPIEDADES

Definición

- El corte es un predicado predefinido que permite al programador intervenir en el mecanismo de búsqueda de soluciones de PROLOG.
- Se denota mediante un punto de exclamación, `!`, no tiene argumentos, y es siempre cierto.
- Se puede incluir, como un predicado más, en el *cuerpo* de las reglas o en las consultas, en cualquier posición.

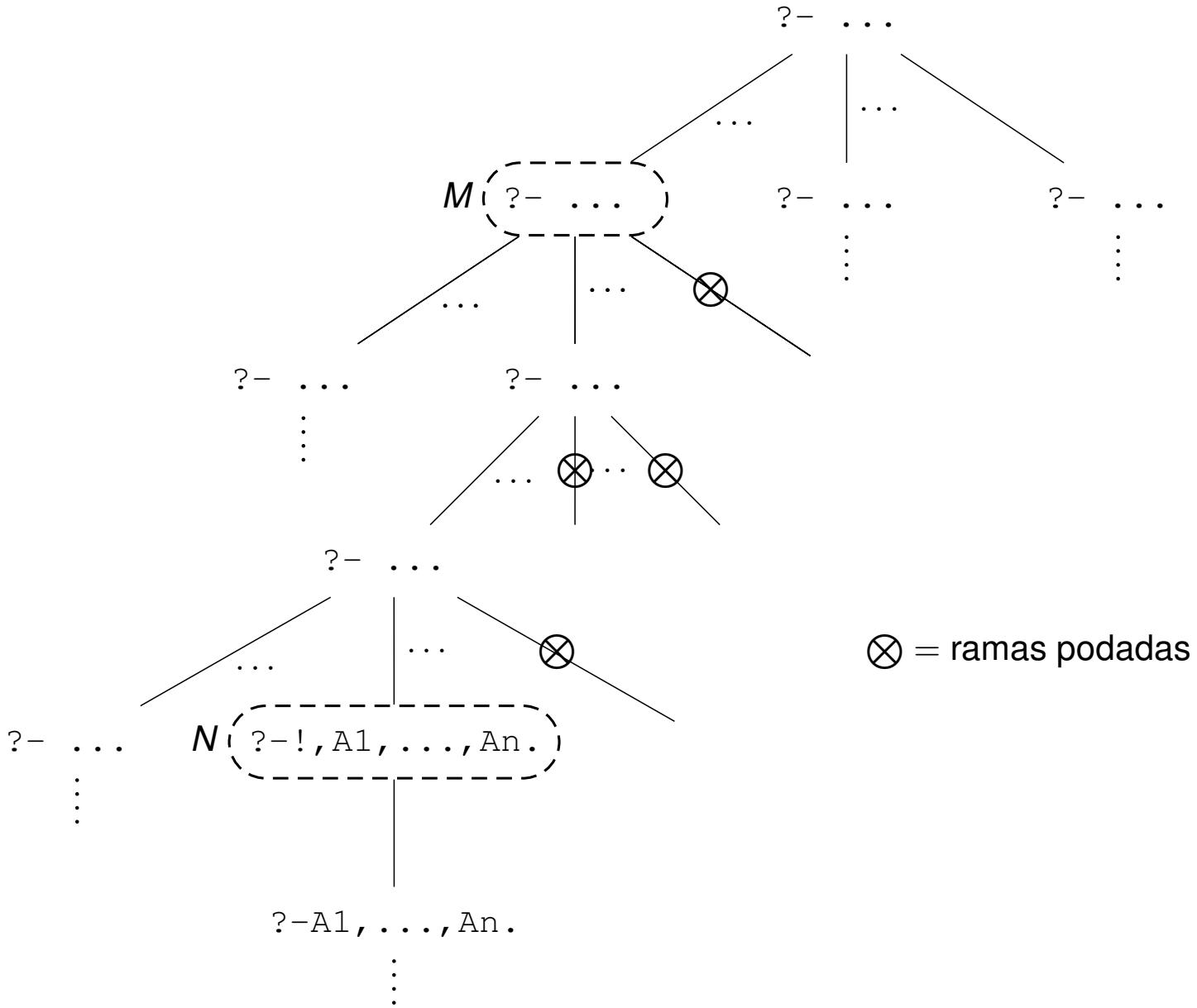
Ejemplos

- `B :- B1, B2, !, B3.` `B :- !.`
  `B :- B1, B2, !.` son reglas válidas.
- `?- A1, !, A2.` o `?- A1, !.` son consultas válidas.

Efectos de un corte en un Árbol de Resolución

- Los cortes podan (impiden el desarrollo) ciertas ramas de los Árboles de Resolución, las elegidas de acuerdo con lo que se explica a continuación.
- Si en un Árbol de Resolución (ver dibujo siguiente):
  - `N` es un nodo que *empieza* con un corte (`N = ?- !,A1,...,An.`)
  - `M` es su ascendiente más cercano que no contiene ese corte (o la raíz del árbol si todos sus ascendientes lo contienen)

  entonces el nodo `N` se expande normalmente (en particular, como el corte es cierto, `N` tiene siempre un único hijo, él mismo sin el corte), pero al retroceder en la construcción del Árbol el sistema poda todas las ramas del árbol que pasan por `M` y no por `N` y que están situadas más a la derecha que la rama que lleva de `M` a `N` (todas las marcadas con $\otimes$ en el dibujo).
- PROLOG retrocede automáticamente al encontrar una rama podada (no reporta las podas).


?- ...
M
?- ...
?- ...
?- ...
?- ...
?- ...
?- ...
N
?-!,A1,...,An.
?-A1,...,An.
⊗ = ramas podadas

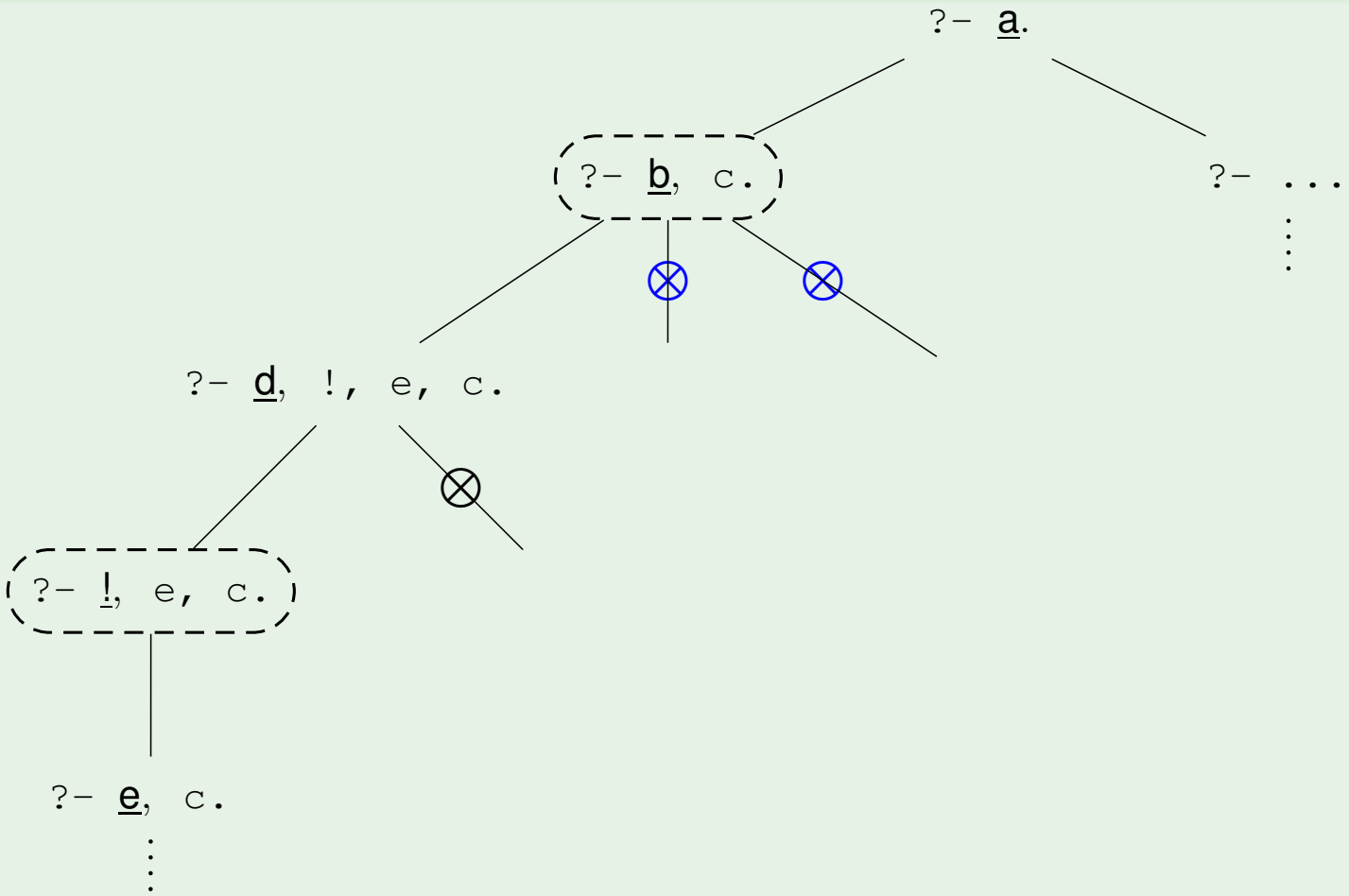
Ejemplo (Efectos del predicado de corte, 1/4)

Considere el programa PROLOG dado por las siguientes cláusulas:

```
a :- b, c.
a :- ....
b :- d, !, e.
b.
b :- ....
d.
d :- ....
e :- ....
```

La página siguiente muestra el Árbol de Resolución para la consulta `?- a`. Las ramas marcadas con $\otimes$ son las que se deben ignorar debido al corte de la regla `b :- d, !, e.`

Ejemplo (Efectos del predicado de corte, 2/4)



El corte de `b :- d, !, e.` poda tanto las alternativas para probar lo que hay a su izquierda (el predicado `d` en este caso, ⊗), como las alternativas para probar la cabeza de la regla (el predicado `b`, ⊗).

Ejemplo (Efectos del predicado de corte, 3/4)

Considere el programa PROLOG dado por las siguientes cláusulas:

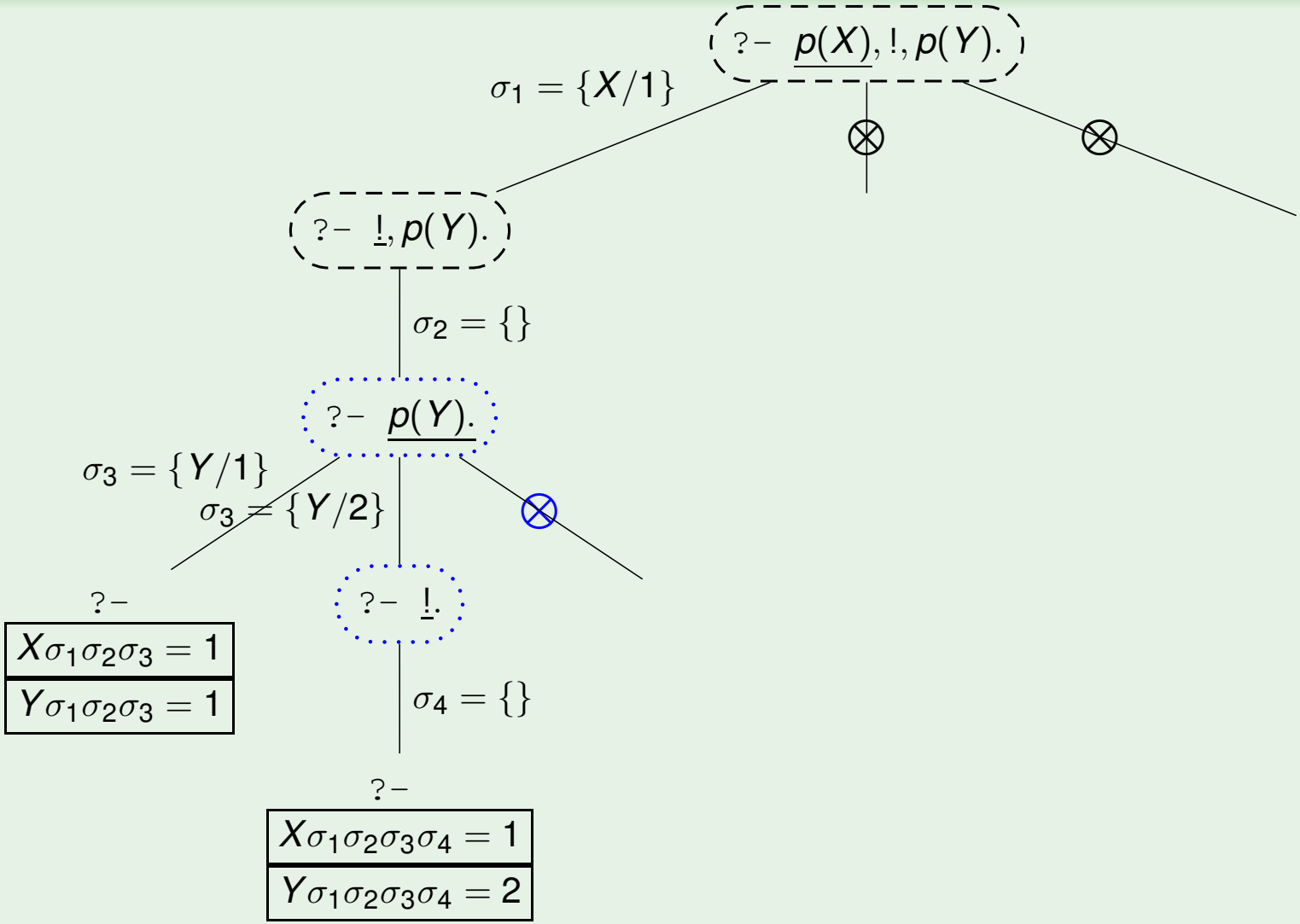
```
p(1).
p(2) :- !.
p(3).
```

La página siguiente muestra el Árbol de Resolución resultante al realizarse la consulta `?- p(X), !, p(Y).`

Este ejemplo ilustra lo siguiente:

1. Un Árbol de Resolución puede verse afectado por *varios* predicados de corte (marcados con distinto color en el dibujo).
2. El nodo `M` mencionado previamente en el caso general puede ser la propia raíz del árbol (lo será cuando la raíz contiene el corte).

Ejemplo (Efectos del predicado de corte, 4/4)



Soluciones: 1) `X=1, Y=1` 2) `X=1, Y=2` 3) `false` indicando que no hay más soluciones (las podas no se reportan).

1. La introducción de un corte en el cuerpo de una regla

```
B :- B1, ..., Bi, !, Bi+1, ..., Bm.
```

   - Impide las posibles reevaluaciones de los objetivos situados *a la izquierda* del corte: si se consigue probar B1, ..., Bi, no se vuelve a intentar su prueba al retroceder en la construcción del árbol (ver podas $\otimes$ en los dos últimos ejemplos). El corte, sin embargo, *no* impide la reevaluación de los objetivos situados a su derecha.
   - Impide las posibles reevaluaciones del predicado `B` mediante cualquier cláusula posterior con cabeza unificable con `B` (ver podas $\otimes$ en los dos últimos ejemplos).

2. La introducción de un corte en una consulta

```
?- A1, ..., Ai, !, Ai+1, ..., An.
```

   - Impide las posibles reevaluaciones de los objetivos situados *a la izquierda* del corte: si se consigue probar A1, ..., Ai, no se vuelve a intentar su prueba al retroceder en la construcción del árbol (ver podas $\otimes$ en el último ejemplo). El corte *no* impide la reevaluación de los objetivos situados a su derecha.

Ejercicios

1. *Dado el programa del último ejemplo, piense qué respuestas ofrecería* PROLOG*, y en qué orden, para cada una de las siguientes consultas, y compruebe sus respuestas construyendo los Árboles de Resolución correspondientes:*

```
?- p(X).                    |    ?- !, p(X), p(Y).
?- p(X), p(Y).              |    ?- p(X), p(Y), !.
```

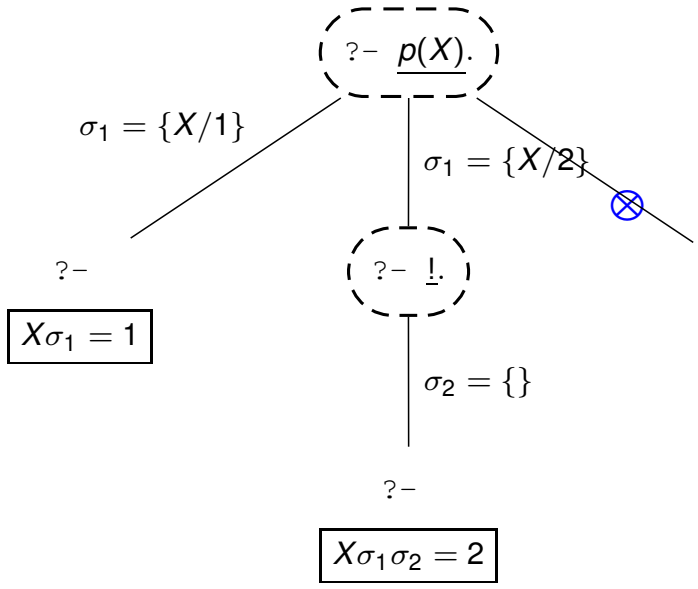
2. *Considere la consulta* `?- s(Z).` *y el programa*

```
s(Y) :- q(X,Y), !, r(Y).    |    q(a,a).
s(X) :- q(X,X).             |    q(a,b).
r(b).
```

   - *¿Qué respuesta(s) ofrecería* PROLOG *a la consulta planteada, y en qué orden? Construya el Árbol de Resolución correspondiente.*
   - *Misma pregunta que en el apartado anterior pero intercambiando las dos cláusulas del predicado* `q`*.*

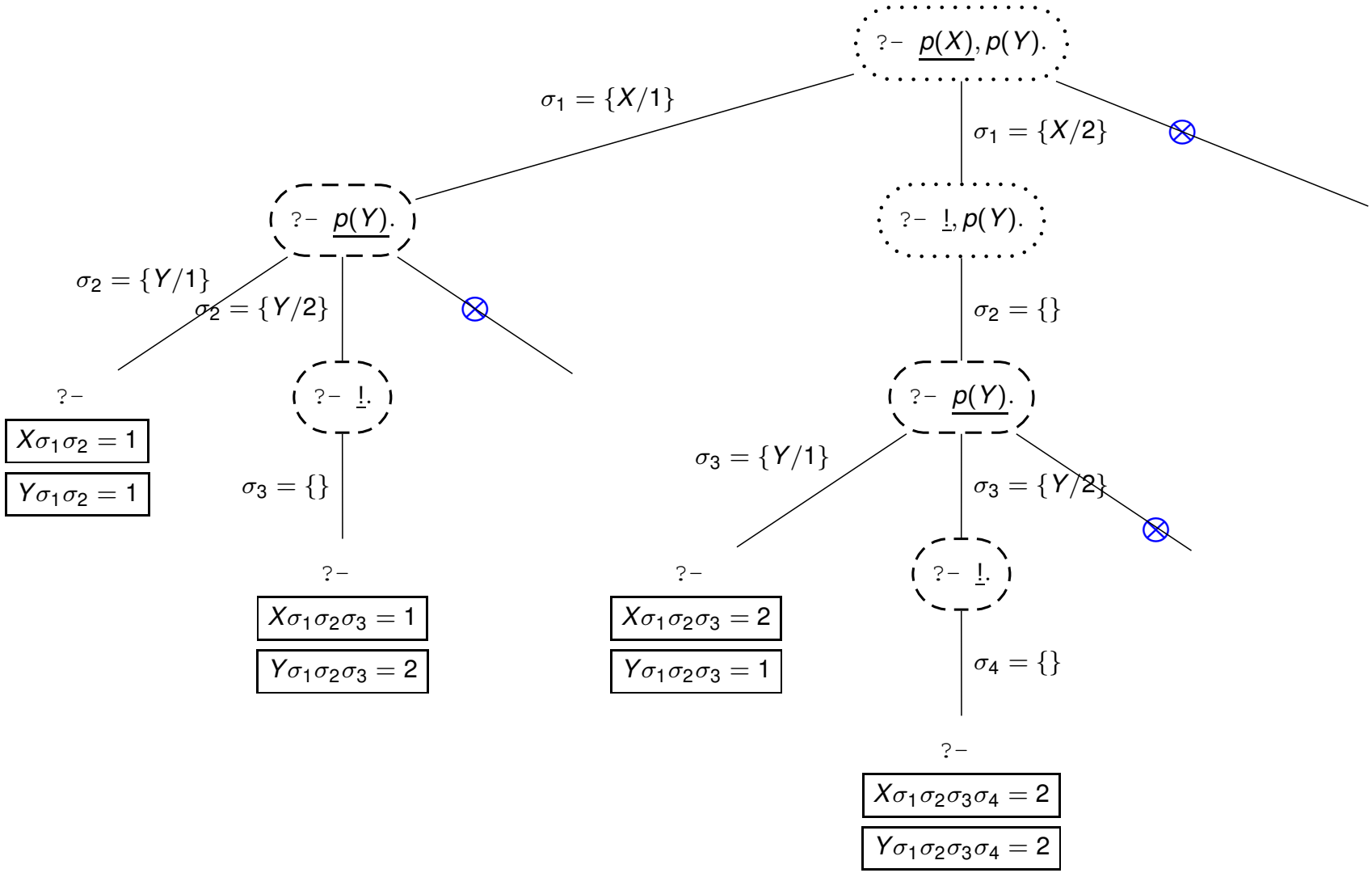
Soluciones propuestas:

1. Consulta `p(X)`. El corte de la regla `p(2) :- !.` impide la reevaluación de `p(X)` con cualquier cláusula posterior a ella, en este caso con el hecho `p(3)`. Por ello, PROLOG solo ofrecerá dos soluciones: `X=1` y `X=2`. En efecto, el Árbol de Resolución correspondiente es el siguiente:



Soluciones: 1) `X=1` 2) `X=2` 3) `false` indicando que no hay más soluciones (las podas no se reportan).

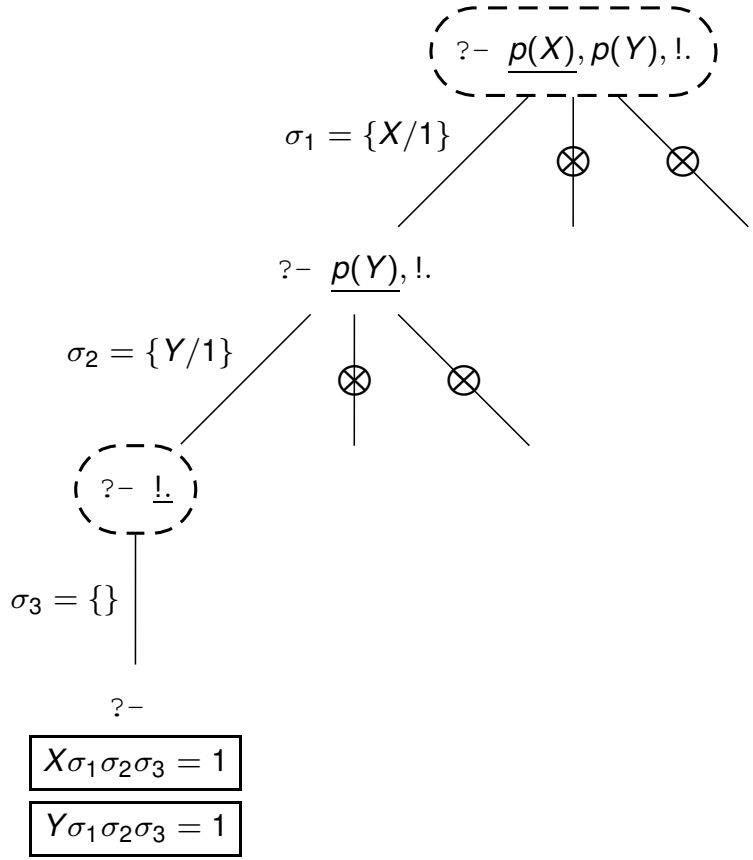
Consulta `?- p(X),p(Y).` El corte de la regla `p(2) :- !.` impide la reevaluación tanto de `p(X)` como de `p(Y)` con el hecho posterior `p(3)`. En efecto:



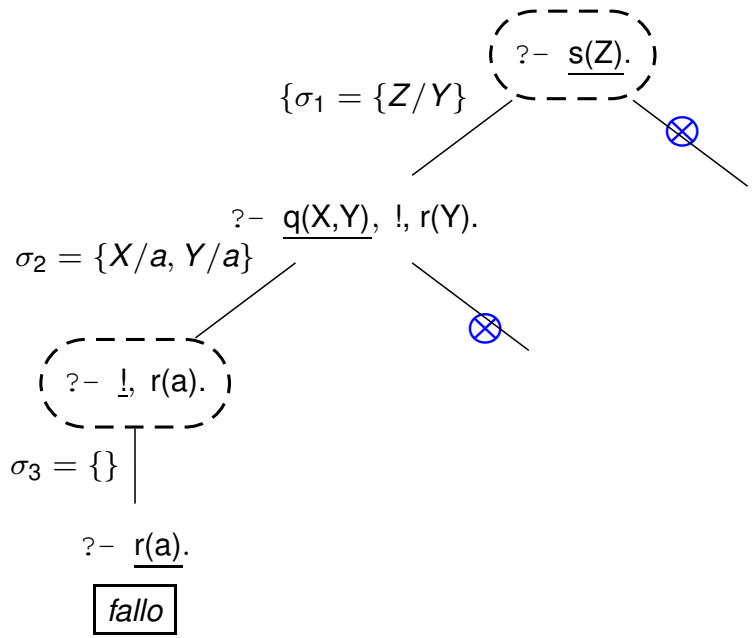
Soluciones: 1) `X=1, Y=1` 2) `X=1, Y=2` 3) `X=2, Y=1` 4) `X=2, Y=2` 5) `false` indicando que no hay más soluciones (las podas no se reportan).

Consulta `?- !, p(X), p(Y).` El Árbol de Resolución correspondiente a esta consulta -y por lo tanto las soluciones ofrecidas- es el mismo que el de la consulta anterior, `?- p(X), p(Y)`, con la única diferencia de que habría que añadirle la consulta `?- !, p(X), p(Y).` como raíz con un único hijo. Colocar un corte en la cabeza de una consulta no aporta nada (recuerde que los cortes en las consultas sirven para impedir la reevaluación de lo que haya a su *izquierda*).

Consulta `?- p(X), p(Y), !.` El corte de la consulta impide las posibles reevaluaciones de todos los objetivos situados a su izquierda, por lo que tanto `p(X)` como `p(Y)` se evaluarán solo con la primera cláusula, dando como única solución `X=1,Y=1`. En efecto, el Árbol de Resolución asociado es el siguiente:



2. El corte de la regla `s(Y) :- q(X,Y), !, r(Y)` impide todas las posibles reevaluaciones tanto de `q` (por estar a la izquierda del corte) como de `s`, provocando que la única respuesta de PROLOG ante la consulta `?- s(Z)` sea `false`: en efecto, la primera (y única debido al corte) forma de probar `q(X,Y)` es utilizando el hecho `q(a,a)` mediante u.m.g. $\{X/a, Y/a\}$, por lo que el siguiente objetivo es `r(a)`, que falla, y `s` no se reevalúa con `s(X) :- q(X,X)` otra vez debido al corte. Todo lo anterior queda reflejado en el Árbol de Resolución correspondiente:



Si se intercambian las dos cláusulas del predicado `q`, la rama fallo se convierte en rama éxito con solución `X=a,Y=b`, puesto que la `Y` se sustituiría por `b` en lugar de por `a`.

Ventajas del corte

- Permite mejorar la eficiencia de los programas, evitando la exploración de partes del Árbol de Resolución de las que se sabe de antemano que no conducirán a ninguna nueva solución.
- Permite aumentar la expresividad del lenguaje (por ejemplo, la negación por fallo finito que se verá más adelante).

Inconvenientes del corte

- Interviene en *cómo* se debe resolver el problema, entrando en contradicción con los principios de la programación lógica pura.
- Puede conducir a programas difíciles de leer y de validar, y puede provocar muchos errores de programación.

Conclusión

El corte es una herramienta de carácter extra-lógico, muy potente pero que debe usarse con mucho cuidado y en casos muy concretos.

Ejercicios

Vuelva al ejercicio nº 2 de la Práctica de PROLOG nº 3 y reflexione sobre las consultas que contienen cortes, comparándolas con las que no los contienen.

USO PARA SIMULACIÓN DE ESTRUCTURAS CONDICIONALES

Uno de los usos más habituales del corte es la simulación de estructuras condicionales.

si b_1 entonces c_1;
si no: si b_2, entonces c_2;
....
si no: si b_n, entonces c_n;
si no: c.

$\Longrightarrow$

```
a :- b1, !, c1.
a :- b2, !, c2.
....
a :- bn, !, cn.
a :- c.
```

- En la estructura condicional anterior, los cortes permiten indicar que las distintas reglas que definen el predicado `a` son incompatibles entre sí (puesto que se rigen por *"si no's"*).

- Si se llega a ejecutar el predicado de corte de una regla `a :- bi, !, ci`, es porque las condiciones `b1,..,b(i-1)` han fallado pero sin embargo se ha probado que `bi` es cierto, y la ejecución del corte tiene los dos siguientes efectos:

  1. Poda las posibles formas alternativas que haya para probar la condición `bi` (en general, para probar todo lo que hubiese delante del corte en el cuerpo de la regla), puesto que ya se ha demostrado que esa condición es cierta.

  2. Poda las posibles formas alternativas posteriores que haya para probar `a`, puesto que esas alternativas solo son válidas cuando `bi` resulta no ser cierto.

Ejemplo (Cálculo de una función definida a trozos, 1/3)

Suponga que necesita definir un predicado en PROLOG que permita calcular la siguiente función *fun*:

$$fun(x) = \begin{cases} 0, & \text{si } x \leq 10 \\ 1, & \text{si } 10 < x \leq 20 \\ 2, & \text{si } x > 20 \end{cases}$$

VERSIÓN 1: sin usar el predicado de corte

```
f(X,0) :- X =< 10.
f(X,1) :- X > 10, X =< 20.
f(X,2) :- X > 20.
```

La incompatibilidad entre las tres reglas se consigue con comprobaciones explícitas (`X>10` en la segunda regla, `X>20` en la tercera).

Ejemplo (Cálculo de una función definida a trozos, 2/3)

VERSIÓN 2: usando el predicado de corte

```
f(X,0) :- X =< 10, !.
f(X,1) :- X =< 20, !.
f(_,2).
```

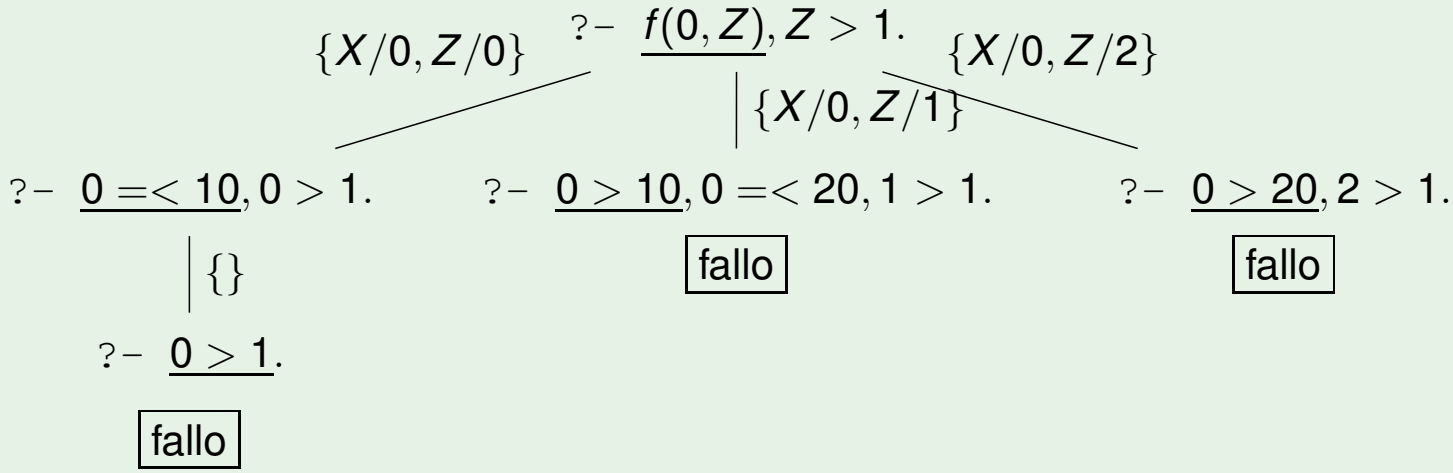
Esta versión es más eficiente: los cortes garantizan la incompatibilidad entre las reglas, evitando así las comprobaciones de la versión anterior (compare los Árboles de Resolución a continuación).

Note sin embargo que la versión 2 no siempre funciona correctamente, resulta por ejemplo `?- f(0,2).` ¡¡true!!. Solución correcta:

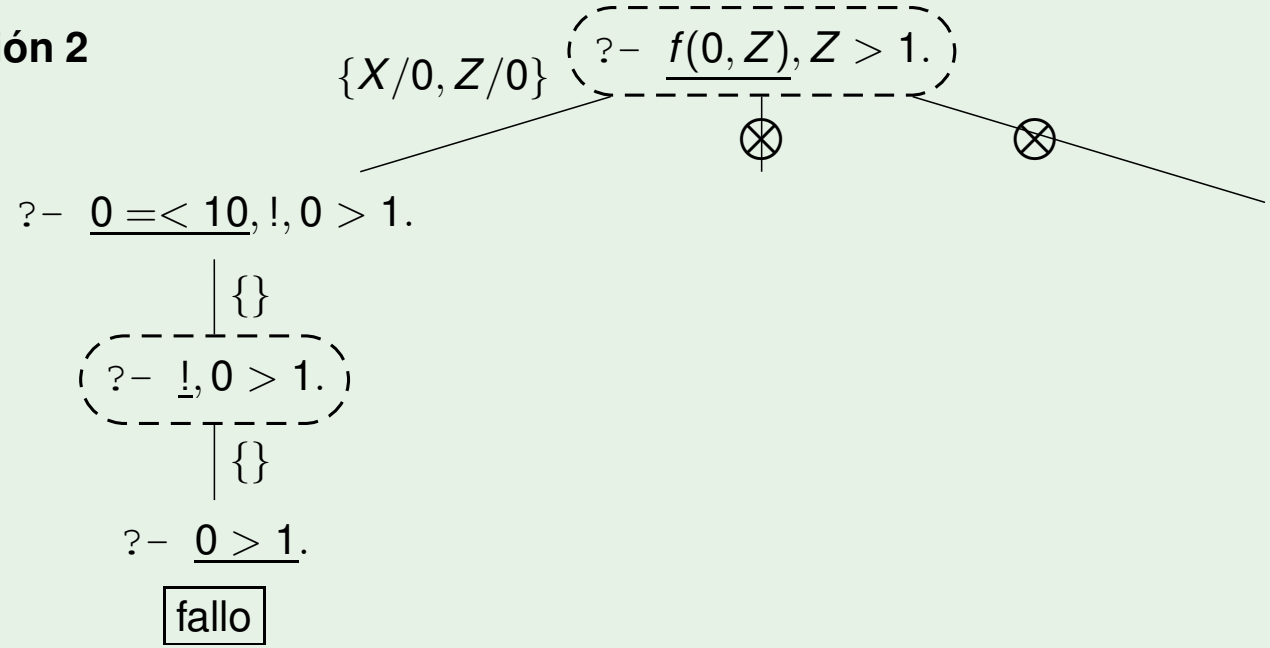
```
f(X,Y) :- X =< 10, !, Y=0.
f(X,Y) :- X =< 20, !, Y=1.
f(_,2).
```

Ejemplo (Cálculo de una función definida a trozos, 3/3)

Versión 1



Versión 2



Ejemplo (Cálculo del máximo de dos números)

VERSIÓN 1: sin usar el predicado de corte

```
maximo(X,Y,X) :- X >= Y.
maximo(X,Y,Y) :- X < Y. % comprobación necesaria
```

VERSIÓN 2: usando el predicado de corte

```
maximo(X,Y,X) :- X >= Y, !.
maximo(_,Y,Y).  % comprobación innecesaria
```

La segunda versión es más eficiente pero incorrecta en ciertos casos (por ejemplo `?-maximo(3,0,0).` ¡¡true!!). Solución correcta:

```
maximo(X,Y,Z) :- X >= Y, !, Z=X.
maximo(_,Y,Y).
```

Ejemplo (Pertenencia de un elemento a una lista, 1/2)

VERSIÓN 1: con las dos reglas compatibles

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

Como se ha visto previamente, esta versión (`member` en SWI-Prolog) no solo sirve para averiguar si un elemento pertenece a una lista sino también para recorrer una lista (mediante backtracking):

```
?- pertenece(X,[a,b]).   X=a  ; X=b
```

VERSIÓN 2: con las dos reglas incompatibles, sin corte

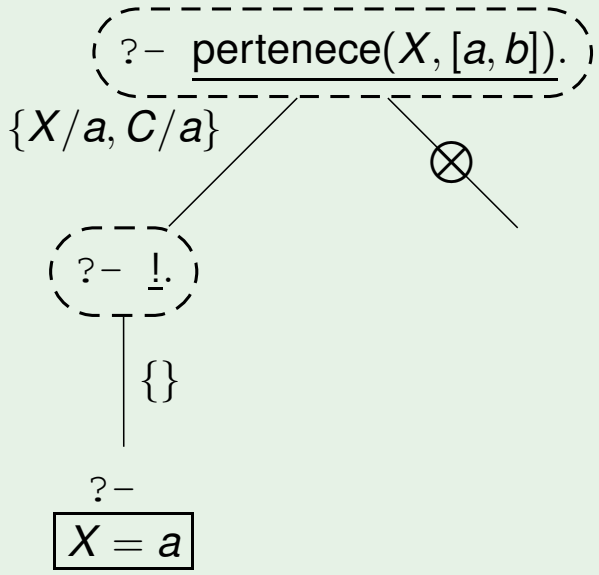
```
pertenece(C, [C|_]).
pertenece(E, [C|R]) :-
    C \= E,
    pertenece(E,R).
```

Ejemplo (Pertenencia de un elemento a una lista, 2/2)

VERSIÓN 3: con las dos reglas incompatibles, con corte

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

Esta última versión (`memberchk` en SWI-Prolog) es más eficiente que la anterior pero ambas son menos potentes que la primera (ya no sirven para recorrer una lista: `?- pertenece(X,[a,b])` solo da `X=a`):



Ejemplo (Eliminación de la primera ocurrencia de un elemento en una lista, 1/2)

VERSIÓN 1: con las dos últimas reglas compatibles

```
borrarelemento([],_,[]).                % C1
borrarelemento([C|R],C,R).              % C2
borrarelemento([C|R],E,[C|NR]) :-       % C3
    borrarelemento(R,E,NR).
```

Esta implementación es incorrecta (construya el Árbol de Resolución):

```
?- borrarelemento([a,b,a], a, L).
L = [b, a] ;      solución correcta
L = [a, b] ;      ¡Solución incorrecta!
L = [a, b, a].    ¡Solución incorrecta!
```

Para solucionar el problema, es necesario indicar que las cláusulas C2 y C3 son incompatibles: C3 *solo* se debe usar cuando la cabeza de la lista, `C`, no coincida con el elemento a borrar, `E`.

Ejemplo (Eliminación de la primera ocurrencia, 2/2)

VERSIÓN 2: con las dos últimas reglas incompatibles, pero sin usar el corte

```
borrarelemento([],_,[]).
borrarelemento([C|R],C,R).
borrarelemento([C|R],E,[C|NR]) :-
    C \= E, % incompatible con la anterior
    borrarelemento(R,E,NR).
```

VERSIÓN 3: con las dos últimas reglas incompatibles, pero usando el corte

```
borrarelemento([],_,[]).
borrarelemento([C|R],C,NL) :-
    !, % incompatible con la regla siguiente
    NL = R.
borrarelemento([C|R],E,[C|NR]) :-
    borrarelemento(R,E,NR).
```

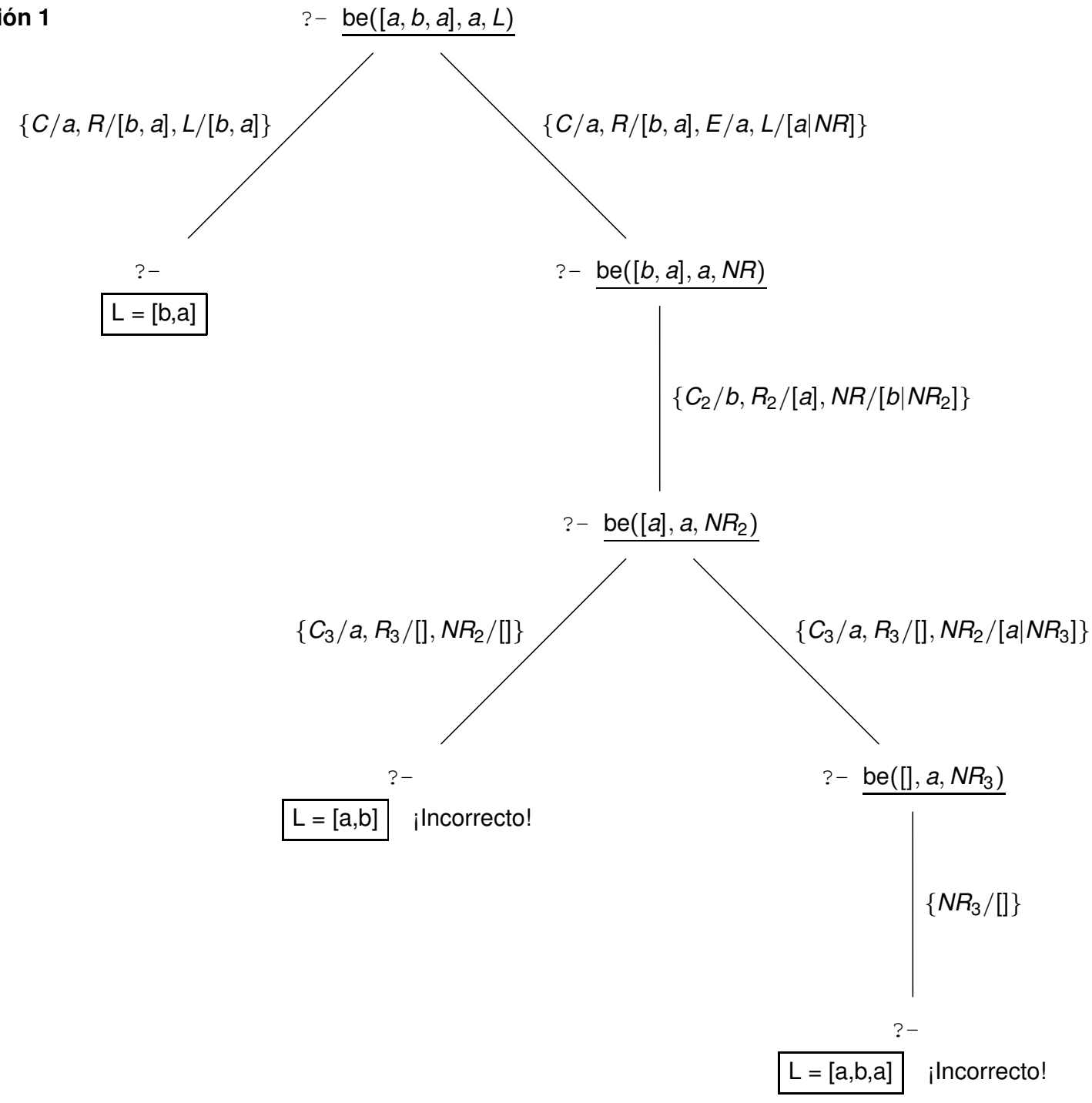
Observación

La sustitución de la segunda regla de la Versión 3 por la más sencilla `borrarelemento([C|R],C,R) :-!.` haría que el predicado no funcionase correctamente en ciertos casos, como por ejemplo con la consulta `?- borrarelemento([1],1,[1])`, que daría `true` en lugar de `false`.

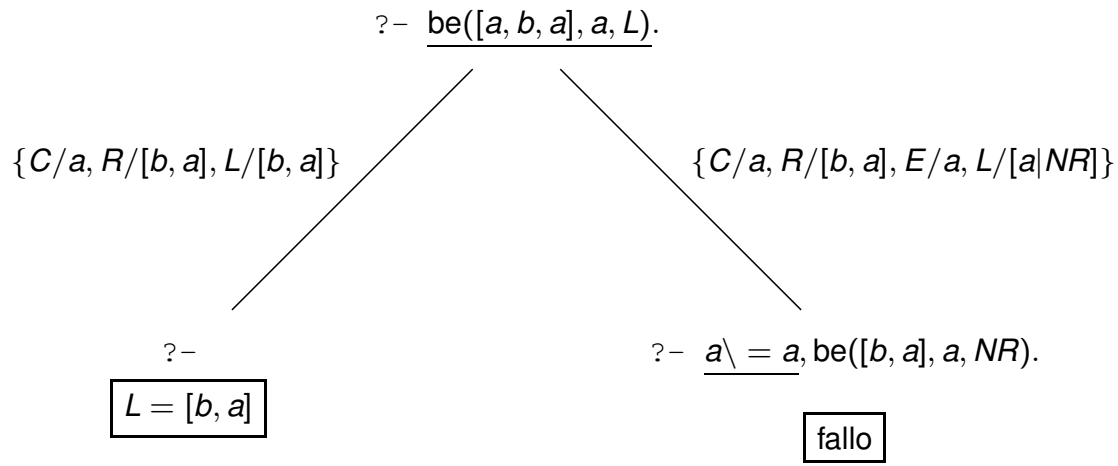
Ejercicios

1. *Construya y compare los Árboles de Resolución para la consulta* `?- borrarelemento([a,b,a], a, L)` *y las tres versiones del predicado* `borrarelemento` *discutidas más arriba.*
2. *Construya y compare los Árboles de Resolución para la consulta* `?- borrarelemento([1], 1, [1])` *con la versión 3 de* `borrarelemento` *y la versión dada en la observación posterior.*

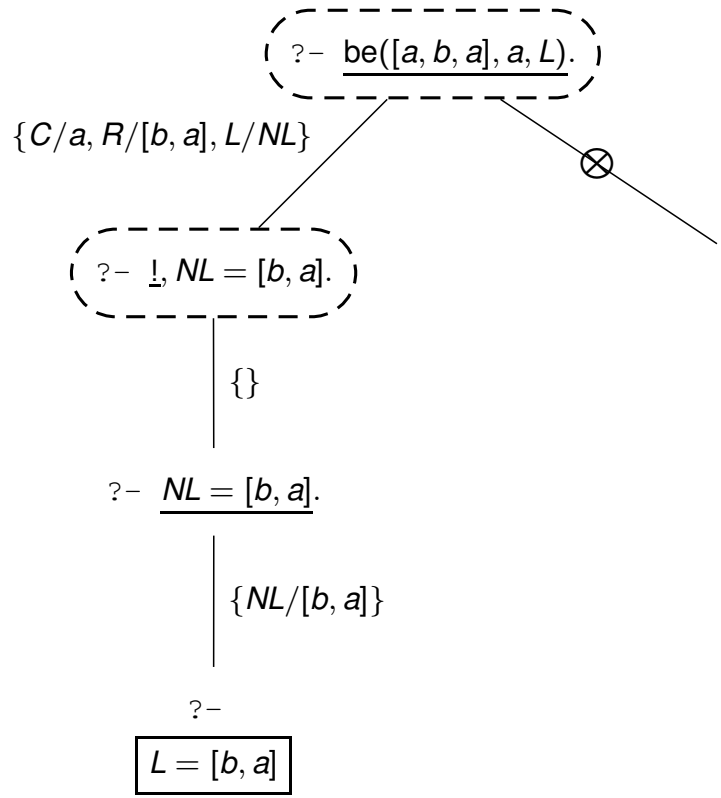
1. Versión 1


?- be([a, b, a], a, L)
{C/a, R/[b, a], L/[b, a]}
{C/a, R/[b, a], E/a, L/[a|NR]}
?-
L = [b,a]
?- be([b, a], a, NR)
{C2/b, R2/[a], NR/[b|NR2]}
?- be([a], a, NR2)
{C3/a, R3/[], NR2/[]}
{C3/a, R3/[], NR2/[a|NR3]}
?-
L = [a,b]
¡Incorrecto!
?- be([], a, NR3)
{NR3/[]}
?-
L = [a,b,a]
¡Incorrecto!

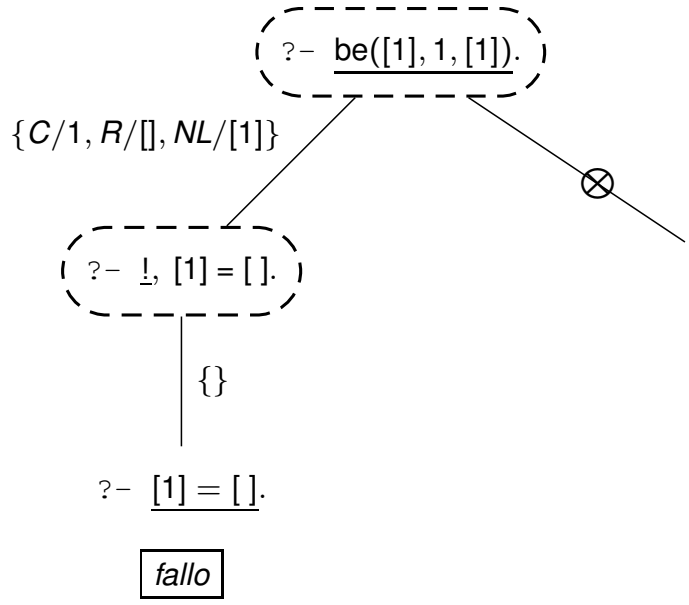
Versión 2



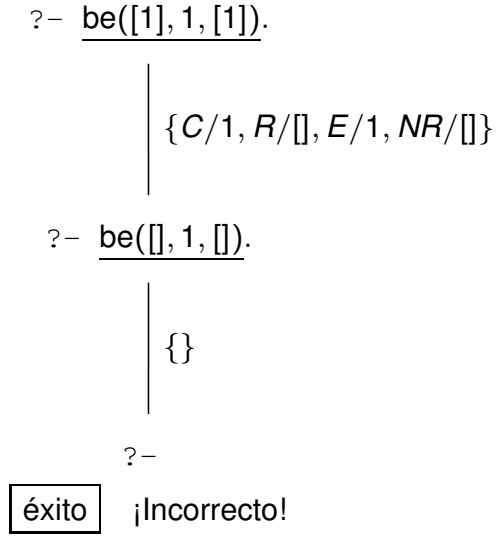
Versión 3



2. Versión 3



Versión de la "Observación"



Ejercicios

1. *Vuelva al ejercicio nº 3 de la Práctica de* PROLOG *nº 3 y reflexione sobre los posibles usos del predicado de corte en sus implementaciones.*
2. *Implemente el predicado* `reemplazar(+L,+E,+F,?NL)`*, cierto si NL es la lista resultante de reemplazar en la lista L todas las ocurrencias de E por F. Haga las consultas adecuadas para comprobar el funcionamiento de su código. Pruebe en particular la consulta* `reemplazar([a,c,a],a,b,L)` *y pida a* PROLOG *todas las posibles soluciones. En caso de que su implementación resulte ser incorrecta, explique por qué y haga las modificaciones necesarias para solucionar el problema. Construya el Árbol de Resolución correspondiente a alguna consulta.*

Soluciones propuestas:

1. Ver soluciones comentadas en Práctica de PROLOG nº 3 con soluciones.

2.

```
% reemplazar(+L,+E,+F,?NL)
% cierto si NL es la lista resultante de reemplazar
% en L todas las ocurrencias de E por F
```

reemplazar([],_,_,[]).

reemplazar([E|R],E,F,[F|NR]) :- !, reemplazar(R,E,F,NR).

reemplazar([C|R],E,F,[C|NR]) :- reemplazar(R,E,F,NR).

Si de la implementación anterior se suprimiese el predicado de corte, la implementación resultante ofrecería una primera solución correcta pero el resto de soluciones serían incorrectas. El motivo es que el corte es necesario para indicar a PROLOG que la tercera cláusula sólo debe utilizarse en caso de que no se pueda utilizar la segunda. Una versión alternativa sin usar el corte, correcta pero algo menos eficiente, consistiría en añadir la comprobación de que *C* no es unificable con *E* al comienzo del cuerpo de la tercera regla. El último ejemplo, en el que se discute la implementación del predicado `borrarelemento` presenta un problema similar a este.

BIBLIOGRAFÍA

- L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.
- W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fifth edition, 2003.
- I. Bratko.*Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 2001.
- J. Lloyd. *Foundations of Logic Programming*, (Second Edition). Springer-Verlag, 1987.
- R. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
- U. Nilsson and J. Maluszynski. Logic, Programming and Prolog. John Wiley & Sons Ltd, 1996.
- SWI-Prolog, entorno de programación en Prolog de dominio público.
- comp.lang.prolog. Faq

Programación Declarativa

3er Curso, Grado en Ingeniería en Informática
Universidad Rey Juan Carlos

Programación Lógica

Construcción paso a paso de un Árbol de Resolución con cortes y negaciones

---

Considere el programa Prolog

```
p(A,B) :- q(A), !, r(B,A).          |          q(1).
r(A,B) :- q(A).                     |          q(2).
t(2).                               |          q(3).
```

y la consulta `?- p(Y,X), \+ t(X).` (cuyo objetivo es averiguar si, dado el conocimiento expresado en el programa anterior, existen objetos `Y` y `X` tales que el primero está relacionado con el segundo mediante la relación "`p`" de forma que el segundo, además, no cumple la propiedad "`t`").

En lo que sigue se detalla la construcción, paso a paso, del árbol de Resolución correspondiente a la consulta anterior y se explica cómo, a partir de él, se deduce qué respuesta(s) ofrecería Prolog ante esa consulta, y en qué orden facilitaría dichas respuestas.

Los árboles de Resolución de Prolog, descritos en el apartado 3.2. del tema PL-2 de los apuntes, se construyen en profundidad por la izquierda y con retroceso. Como el programa y la consulta dados incorporan tanto un corte (`!`) como una negación (`\+`), es necesario conocer además los efectos que estos dos predicados tienen sobre la construcción de los árboles de Resolución. Ambos predicados están descritos, respectivamente, en los apartados 8 y 1 de los temas PL-2 y PL-3.

Construcción del Árbol de Resolución

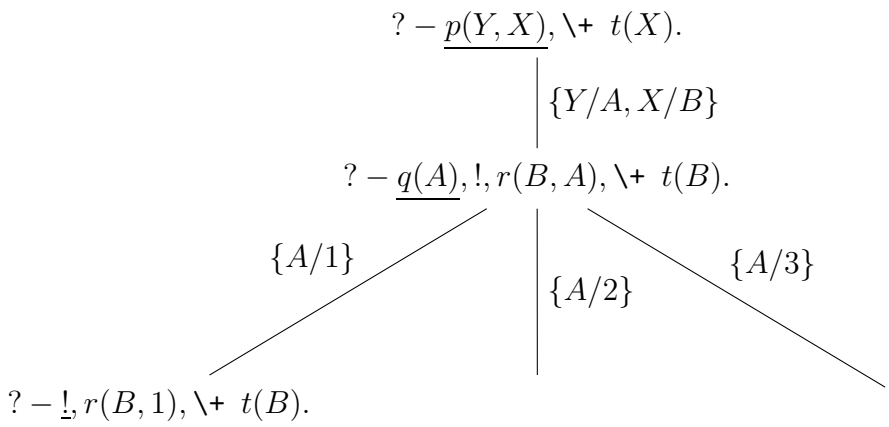
La raíz del árbol no es otra cosa que la consulta, que tendrá tantos hijos como cláusulas haya en el programa cuya cabeza unifique con el predicado más a la izquierda de la consulta, es decir, `p(Y,X)` (Prolog siempre resuelve sus objetivos empezando por el sub-objetivo de más a la izquierda). Sólo hay una cláusula en el programa, la primera, cuya cabeza unifica con `p(Y,X)`, y el nodo hijo será la cláusula resolvente obtenida al aplicar la Regla de Resolución entre la consulta y esa cláusula. Un unificador de máxima generalidad (u.m.g.) que permite hacer lo anterior es $\{Y/A, X/B\}$, obteniéndose la cláusula resolvente `?- q(A), !, r(B,A), \+ t(B)` (este u.m.g. no es el único posible, se podría haber elegido, por ejemplo, $\{Y/A, B/X\}$, en cuyo caso la cláusula resolvente sería `?- q(A), !, r(X,A), \+ t(X)`, pero los resultados finales serían los mismos). Por lo tanto, el primer nivel del árbol de Resolución es el siguiente:

$$? - \underline{p(Y,X)}, \texttt{\textbackslash+}\ t(X).$$

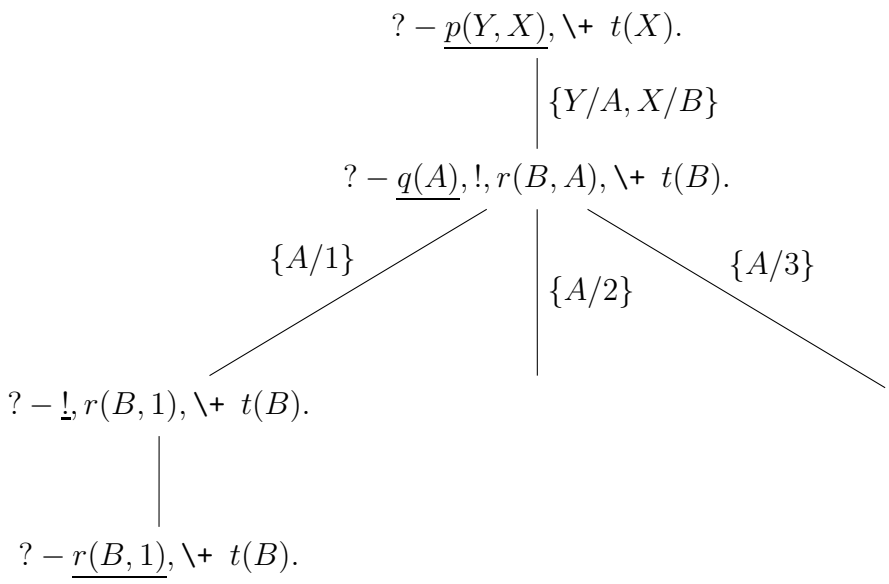
$$\Big| \{Y/A, X/B\}$$

$$? - \underline{q(A)}, !, r(B,A), \texttt{\textbackslash+}\ t(B).$$

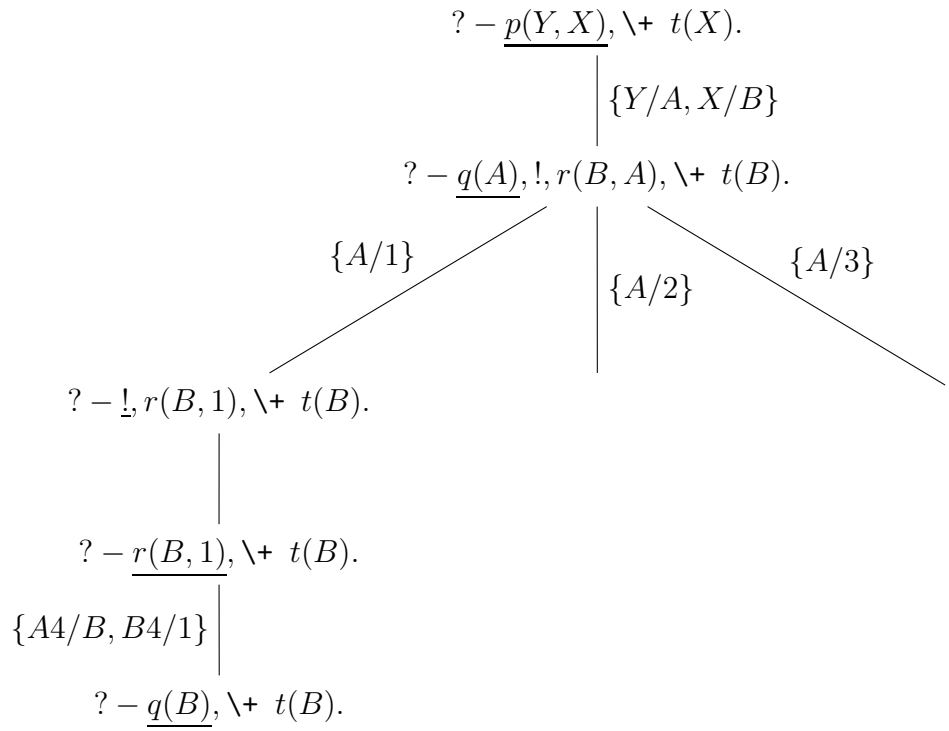
En el siguiente nivel del árbol habrá tres nodos, puesto que el objetivo de más a la izquierda, `q(A)`, unifica con la cabeza de las tres últimas cláusulas del programa, mediante u.m.g.'s $\{A/1\}$, $\{A/2\}$ y $\{A/3\}$, respectivamente. Como Prolog desarrolla el árbol en profundidad por la izquierda y es posible que algunas de las ramas de la derecha acaben siendo podadas, en este momento sólo se calcula el nodo correspondiente a la rama de más a la izquierda (aunque se dejan trazadas las demás ramas por si hubiese que desarrollarlas al retroceder en la construcción del árbol). El árbol resultante es el siguiente:



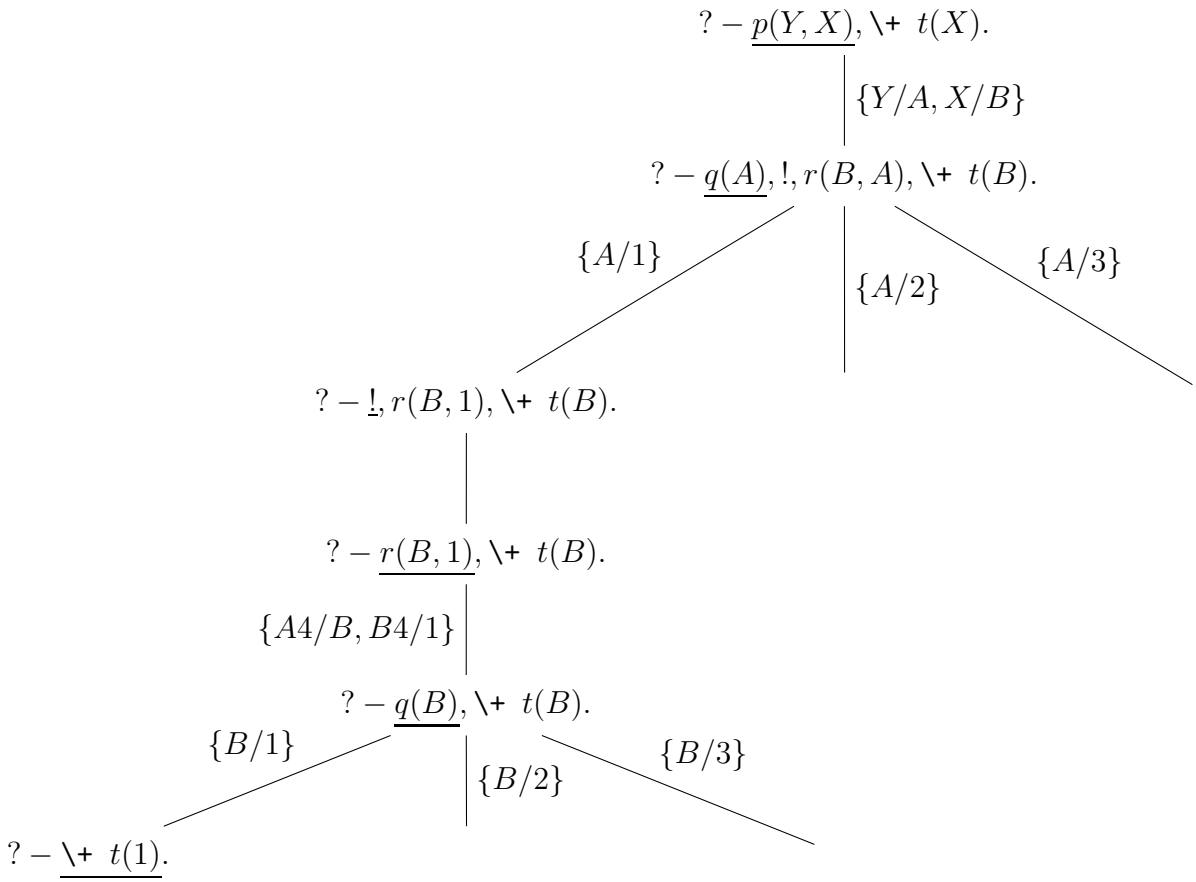
Siguiendo el desarrollo del árbol en profundidad por la izquierda, el nuevo nodo a expandir es `?- !, r(B,1), \+ t(B)`, que tiene un único hijo, igual a él pero sin el corte inicial (el corte es un predicado que es siempre cierto; sus efectos se producirán más adelante, al retroceder en la construcción del árbol).



De forma similar, se trata ahora de expandir el nodo `?- r(B,1), \+ t(B)`, viendo con qué cabezas de las cláusulas del programa unifica el sub-objetivo de más a la izquierda, `r(B,1)`. Éste sólo unifica con la segunda cláusula, que habrá que renombrar puesto que hay conflicto de variables (la variable `B` aparece tanto en la regla del programa como en `r(B,1)`). Como estamos en el cuarto nivel del árbol, renombramos la cláusula del programa como `r(A4,B4) :- q(A4)`, y mediante u.m.g. $\{A4/B, B4/1\}$ se obtiene la cláusula resolvente `?- q(B), \+ t(B)`:



El sub-objetivo de más a la izquierda, `q(B)`, unifica con las tres últimas cláusulas del programa, por lo que el nodo tendrá tres hijos, aunque, como siempre, por el momento basta con calcular el de más a la izquierda, que será `?- \+ t(1)` ya que el u.m.g. es $\{B/1\}$.

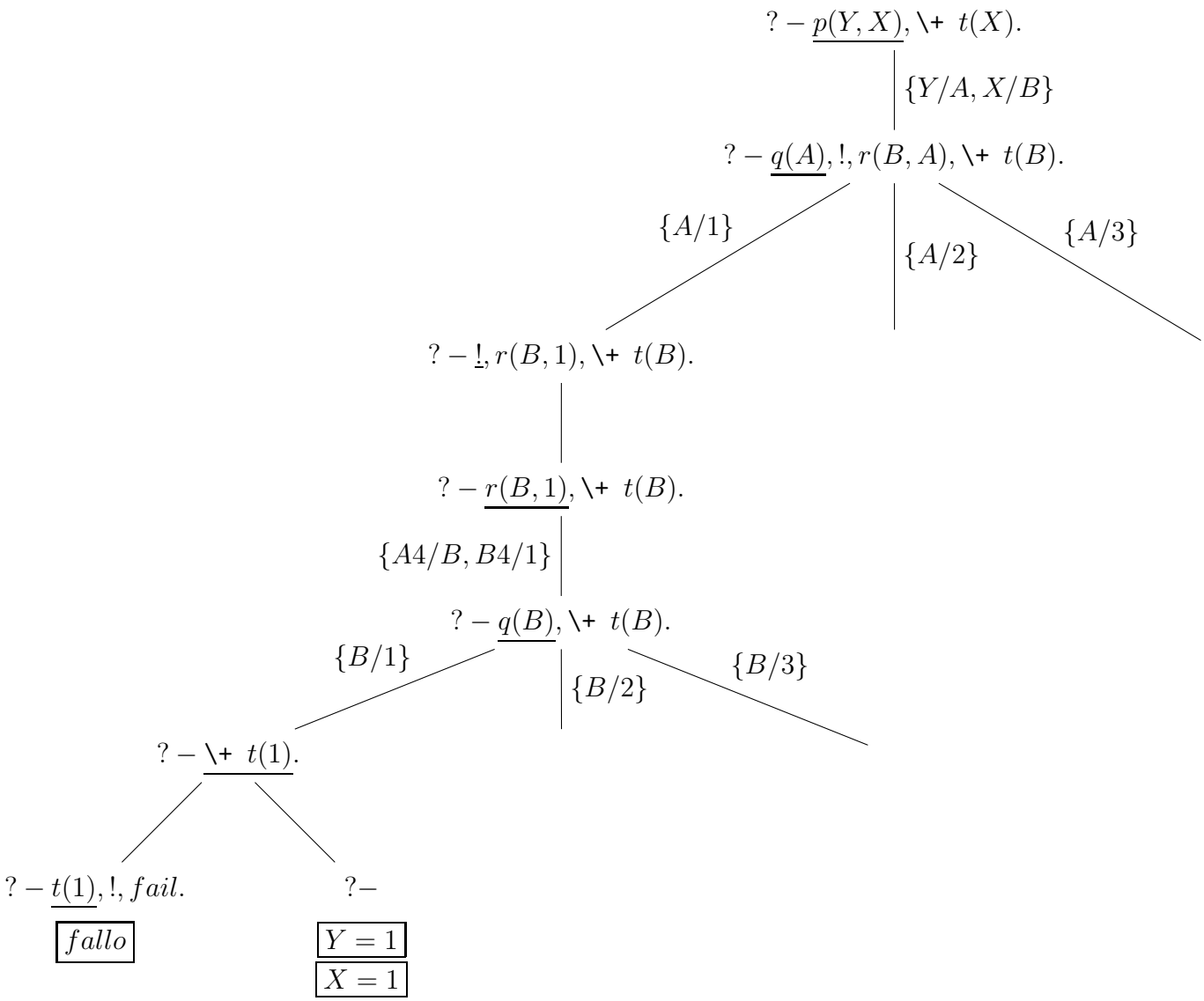


Hay que desarrollar ahora el nodo `?- \+ t(1)`, que, al ser un objetivo negado, tendrá dos hijos (ver tema PL-3), de los cuales por el momento solo se calcula el de la izquierda, que es `?- t(1), !, fail.` Este último resulta ser un nodo fallo, puesto que su primer sub-objetivo, `t(1)`, no unifica con la cabeza de ninguna cláusula del programa. Al llegar a un nodo fallo, Prolog retrocede para seguir desarrollando el árbol en profundidad. El siguiente hijo del nodo padre `?- \+ t(1)` es directamente un nodo éxito, por lo que Prolog calcula la solución asociada aplicando a las variables de la consulta, `Y` y `X`, todos los u.m.g.'s de la rama, en orden, empezando desde la raíz:

$$Y\{Y/A, X/B\}(\{A/1\}\{A4/B, B4/1\}\{B/1\}) = A\{A/1\}(\{A4/B, B4/1\}\{B/1\}) = 1$$

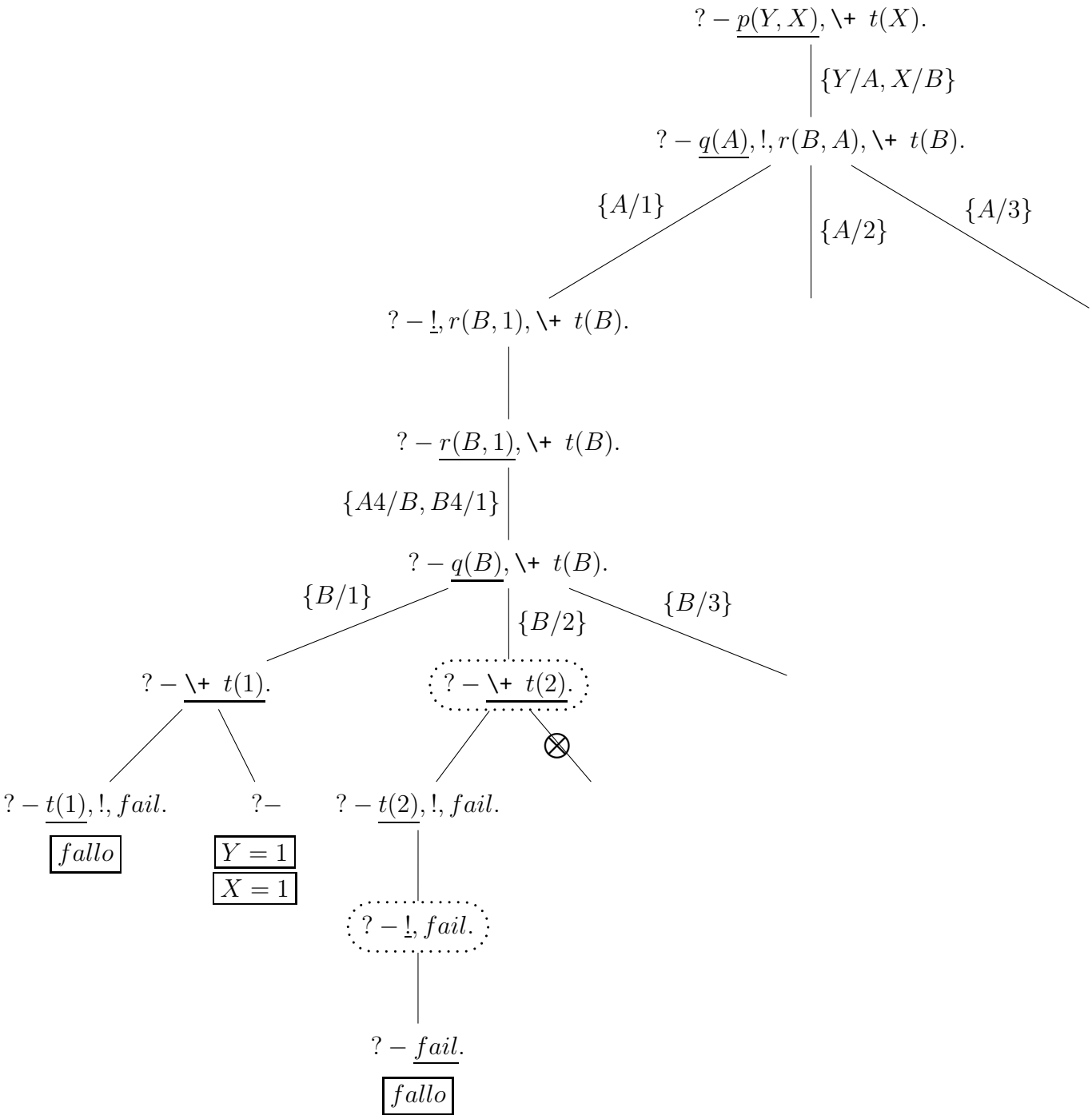
$$\begin{aligned} X\{Y/A, X/B\}(\{A/1\}\{A4/B, B4/1\}\{B/1\}) &= B\{A/1\}(\{A4/B, B4/1\}\{B/1\}) \\ &= B\{A4/B, B4/1\}(\{B/1\}) = B\{B/1\} = 1 \end{aligned}$$

El árbol resultante hasta el momento es por lo tanto el siguiente:

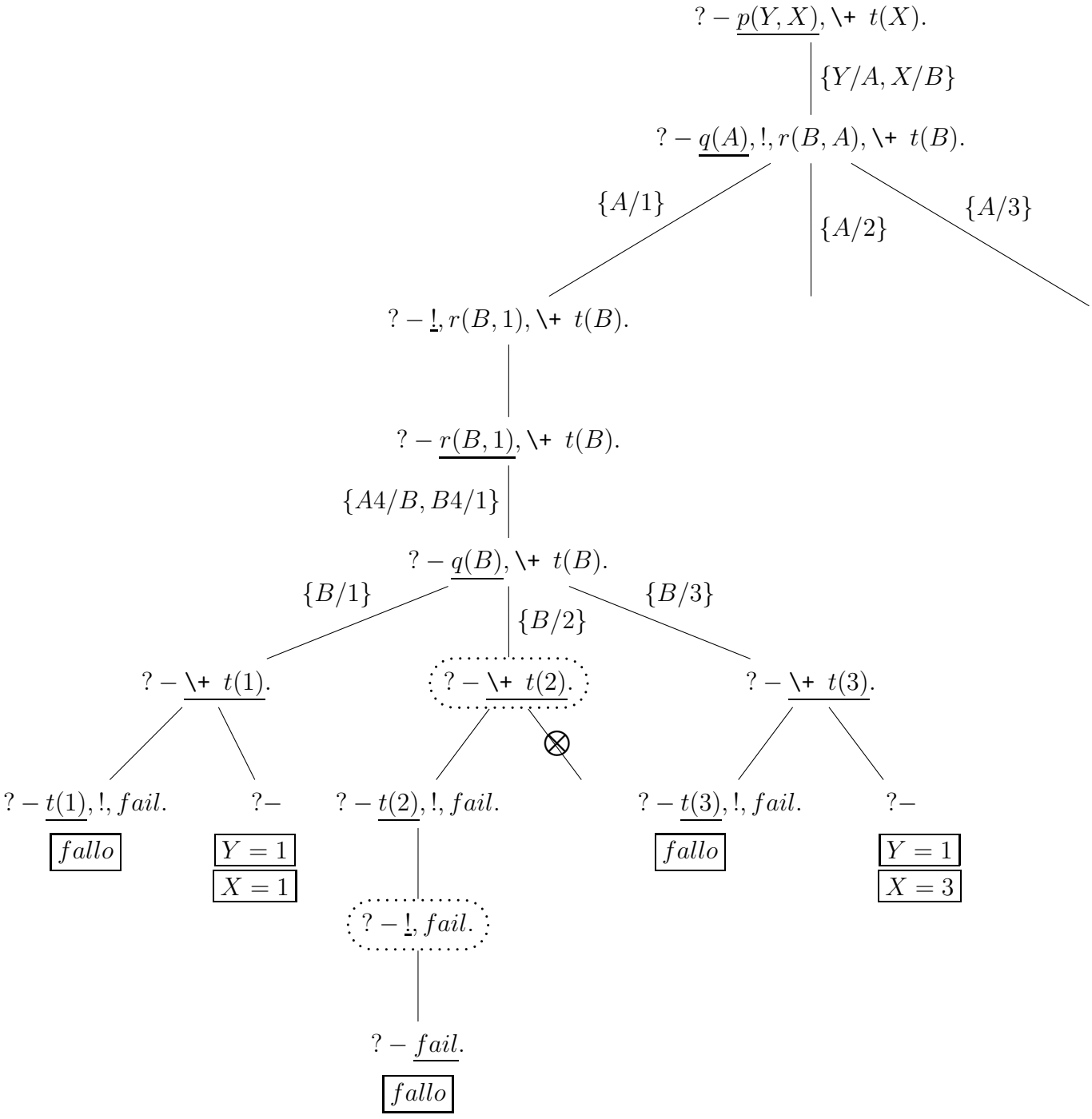


Al seguir desarrollando el árbol en profundidad, Prolog retrocede hasta el siguiente nodo pendiente de cálculo, el segundo hijo del nodo `?- q(B), \+ t(B)`, que se obtiene me-

diante la cláusula `q(2)` y la sustitución $\{B/2\}$, dando lugar al nodo `?- \+ t(2).` Este último, al empezar por un objetivo negado, tendrá dos hijos, siendo el de la izquierda `?- t(2), !, fail.` Como `t(2)` es cierto, este nodo tiene como único hijo `?- !, fail` que a su vez, al ser el corte siempre cierto, da lugar al nodo `?- fail`, que falla (el predicado predefinido `fail` siempre falla). El fallo anterior hace que Prolog retroceda por la rama, y como al hacerlo encuentra un nodo empezando por corte, en el retroceso poda todas las ramas comprendidas entre el nodo que empieza por corte y su ancestro más cercano que no contiene corte (ambos marcados en el árbol a continuación mediante una elipse intermitente). En este caso no hay más que una rama por podar, la marcada en el árbol mediante el símbolo $\bigotimes$.

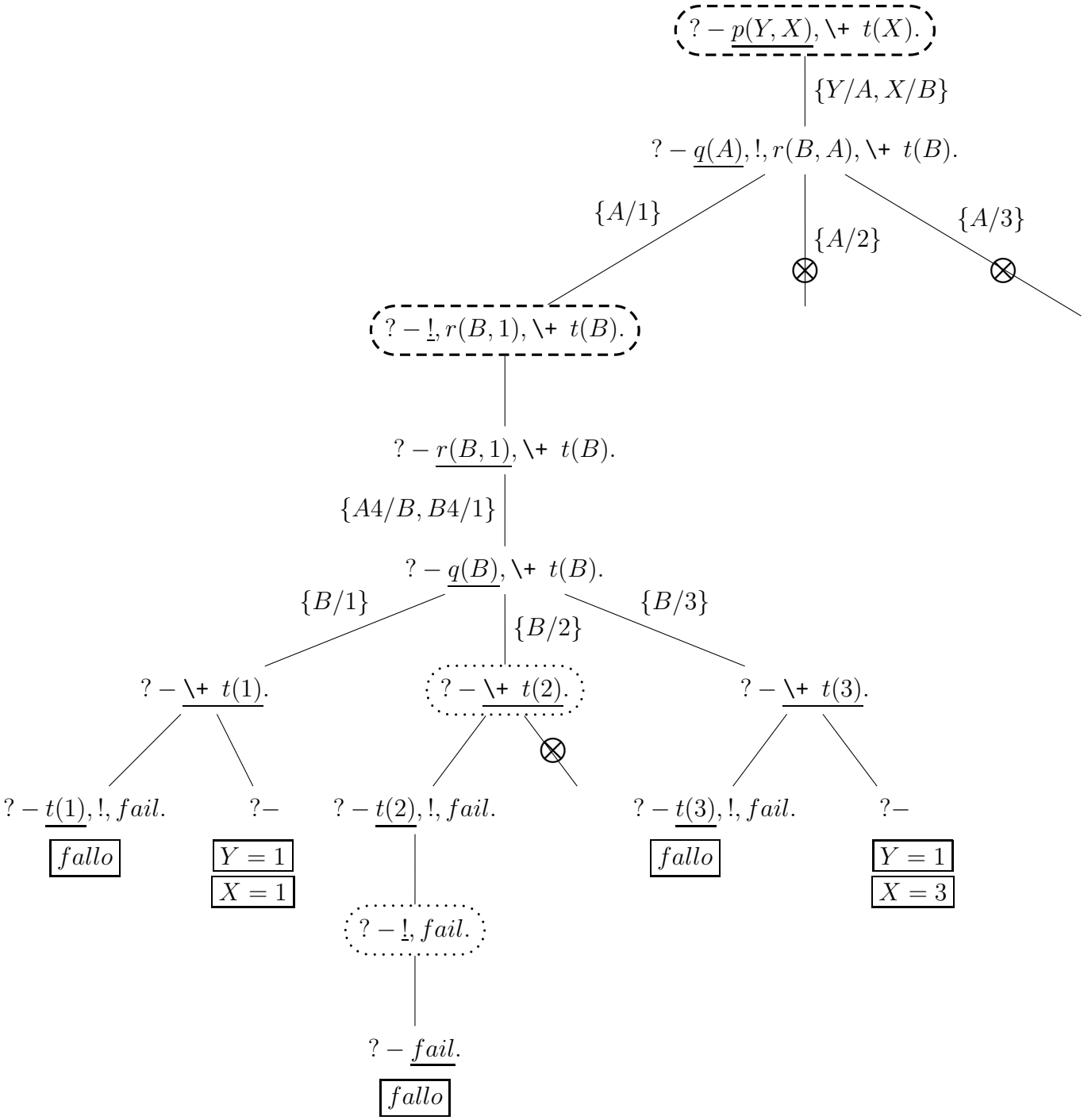


El siguiente nodo a expandir es el tercer hijo del nodo `?- q(B), \+ t(B)`, que resulta análogo al primer hijo, dando lugar al siguiente árbol:



Una vez terminada la rama en la que ha encontrado la solución `Y=1, X=3`, Prolog retrocede en busca del siguiente nodo pendiente de explorar. En el camino hacia arriba acaba llegando al nodo `?- !, r(B,1), \+ t(B)`, que empieza por corte, por lo que tiene que buscar el ancestro más cercano que no contiene el corte, que resulta ser la raíz del árbol, y podar todas las ramas que salgan a la derecha entre uno y otro. En el árbol que se dibuja a continuación los dos nodos que determinan la poda aparecen marcados con una elipse intermitente, mientras que las ramas podadas se marcan con el símbolo $\bigotimes$.

Ya no queda ningún nodo por expandir, por lo que la construcción del Árbol de Resolución termina. El árbol completo obtenido es el siguiente:



Respuestas de Prolog

Ante una consulta, Prolog construye el árbol de Resolución, en profundidad, y facilita las soluciones según las va encontrando. Por lo tanto, ante la consulta dada, y a la vista del árbol anterior, la respuesta de Prolog será la siguiente:

```
Y=1, X=1 ;    % primera solución encontrada
Y=1, X=3 ;    % segunda solución encontrada
false (o no)  % indicando que no quedan más soluciones
```

PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL3: El lenguaje PROLOG, aspectos avanzados

1. El predicado de negación

Grado en Ingeniería Informática

URJC

Ana Pradera

Contenido

1. DEFINICIÓN
   - Negación en programas
   - Negación en consultas
2. IMPLEMENTACIÓN Y FUNCIONAMIENTO
   - Implementación
   - Funcionamiento
3. NO MONOTONÍA
4. LIMITACIONES

DEFINICIÓN

- La sintaxis de PROLOG estudiada en el tema anterior sólo permite trabajar con información *positiva*.
- Para poder expresar o deducir información negativa se utiliza el operador prefijo `\+`, aplicable a cualquier predicado `P`, escribiendo `\+ P` y leyendo `no P`, siempre que `P` forme parte del cuerpo de una regla o de una una consulta (ni los hechos ni las cabezas de las reglas se pueden negar).
- Para ello se extiende la Programación Lógica Definida pasando a la denominada *Programación Lógica General*, cuyo sistema de demostración, la Resolución SLDNF, implementa una forma de negación conocida como negación por fallo finito. Su funcionamiento, distinto al de la negación de la Lógica Matemática, se explica más adelante.

Ejemplos (Negación en programas, 1/2)

- Todos los pájaros vuelan, a menos (= salvo) que sean especiales.
  - Formalización en LPO:
    $\forall X[(\mathit{pájaro}(X) \wedge \neg \mathit{especial}(X)) \rightarrow \mathit{vuela}(X)]$
  - Formalización en PROLOG:

```
vuela(X) :-
    pájaro(X),
    \+ especial(X).
```

- Abel es amigo de todos los que son amigos de Caín y no son enemigos de Abilio.
  - Formalización en LPO:
    $\forall X[(\mathit{amigo}(X, \mathit{cain}) \wedge \neg \mathit{enemigo}(X, \mathit{abilio})) \rightarrow \mathit{amigo}(\mathit{abel}, X)]$
  - Formalización en PROLOG:

```
amigo(abel,X) :-
    amigo(X, cain),
    \+ enemigo(X, abilio).
```

Ejemplos (Negación en programas, 2/2)

- Nadie es enemigo de sí mismo.
  - Formalización en LPO:
    $\forall X(\neg enemigo(X, X))$
  - Formalización en PROLOG:
    No admisible en PROLOG (hecho negado)
- Los que son amigos de Abel no son enemigos de Abilio.
  - Formalización en LPO:
    $\forall X[(amigo(X, abel) \rightarrow \neg enemigo(X, abilio)]$
  - Formalización en PROLOG:
    no admisible en PROLOG (regla con cabeza negada)

Ejercicios (Negación en programas)

Suponga disponibles los predicados `amigo(X,Y)`*, cierto si X es amigo de Y, y* `enemigo(X,Y)`*, cierto si X es enemigo de Y.*

Para cada una de las siguientes frases:

- *Formalícela en Lógica de Primer Orden (LPO).*
- *Razone si podría formar parte de un programa en* PROLOG*, y, en caso afirmativo, facilite su escritura en* PROLOG*.*

1. *Los que no son amigos ni de Abel ni de Caín son amigos de Abilio.*
2. *Nadie es amigo de Caín y de Abel a la vez.*
3. *Los que no son amigos de Abel tampoco son amigos de Abilio.*
4. *Caín no es amigo de nadie.*
5. *Abel es amigo de todos salvo de los que son amigos de Caín.*

Soluciones propuestas:

1. Los que no son amigos ni de Abel ni de Caín son amigos de Abilio.
   LPO: $\forall X[(\neg amigo(X, abel) \wedge \neg amigo(X, cain)) \rightarrow amigo(X, abilio)]$
   PROLOG: *amigo(X, abilio) :- \+ amigo(X, abel), \+ amigo(X, caín).*

2. Nadie es amigo de Caín y de Abel a la vez.
   LPO: $\forall X[amigo(X, abel) \rightarrow \neg amigo(X, cain))]$, o, equivalentemente,
   gracias a las equivalencias lógicas $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_2 \rightarrow \neg\varphi_1$ (ley contrapositiva)
   y $\neg\neg\varphi \equiv \varphi$ (ley de la doble negación), a
   $\forall X[amigo(X, cain) \rightarrow \neg amigo(X, abel))]$
   PROLOG: no admisible en PROLOG (regla con cabeza negada)

3. Los que no son amigos de Abel tampoco son amigos de Abilio.
   LPO: $\forall X[\neg amigo(X, abel) \rightarrow \neg amigo(X, abilio)]$, o, equivalentemente,
   $\forall X[amigo(X, abilio) \rightarrow amigo(X, abel)]$, gracias a la ley contrapositiva.
   PROLOG: *amigo(X, abel) :- amigo(X, abilio).*

4. Caín no es amigo de nadie.
   LPO: $\forall X[\neg amigo(cain, X)]$
   PROLOG: no admisible en PROLOG (hecho negado)

5. Abel es amigo de todos salvo de los que son amigos de Caín.
   LPO: $\forall X[\neg amigo(X, cain) \rightarrow amigo(abel, X)]$
   PROLOG: *amigo(abel, X) :- \+ amigo(X, cain).*

Ejemplos (Negación en consultas)

- ¿Existe algún pájaro que no vuele? ¿Cuál o cuáles?
  - Formalización en LPO: $\exists X(\mathit{pájaro}(X) \wedge \neg \mathit{vuela}(X))$
  - Formalización en PROLOG:

```
?- pájaro(X), \+ vuela(X).
```

- ¿Es cierto que Abel no es amigo de Caín?
  - Formalización en LPO: $\neg \mathit{amigo}(\mathit{abel}, \mathit{cain})$
  - Formalización en PROLOG:

```
?- \+ amigo(abel, cain).
```

- ¿Qué animales existen que no le gusten a Adán?
  - Formalización en LPO: $\exists X(\mathit{animal}(X) \wedge \neg \mathit{gusta}(X, \mathit{adan}))$
  - Formalización en PROLOG:

```
?- animal(X), \+ gusta(X, adan).
```

Ejercicios (Negación en consultas)

Suponga disponibles los predicados `amigo(X,Y)`*, cierto si X es amigo de Y, y* `enemigo(X,Y)`*, cierto si X es enemigo de Y.*

Para cada una de las siguientes preguntas:

- *Formalícela en Lógica de Primer Orden (LPO).*
- *Escriba la correspondiente consulta en* PROLOG*.*

1. *¿Qué personas hay que no sean amigas ni de Caín ni de Abel?*
2. *¿Hay alguien -no importa quién sea- que no sea enemigo de Caín?*
3. *¿Es cierto que Abel no es enemigo de Caín?*
4. *¿Hay alguien que no sea amigo de Abel y sea enemigo de Caín? ¿Quién o quiénes son?*

Soluciones propuestas:

1. ¿Qué personas hay que no sean amigas ni de Caín ni de Abel?

   LPO: $\exists X[\neg amigo(X, cain) \wedge \neg amigo(X, abel)]$
   PROLOG: *?- \+ amigo(X, cain), \+ amigo(X, abel).*

2. ¿Hay alguien -no importa quién sea- que no sea enemigo de Caín?

   LPO: $\exists X[\neg enemigo(X, cain)]$
   PROLOG: *?- \+ enemigo(_, cain).*

   % uso de la variable anónima _

3. ¿Es cierto que Abel no es enemigo de Caín?

   LPO: $\neg enemigo(abel, cain)$
   PROLOG: *?- \+ enemigo(abel, cain).*

4. ¿Hay alguien que sea enemigo de Caín y no sea amigo de Abel? ¿Quién o quiénes son?

   LPO: $\exists X[enemigo(X, cain) \wedge \neg amigo(X, abel)]$
   PROLOG: *?- enemigo(X, cain), \+ amigo(X, abel).*

IMPLEMENTACIÓN Y FUNCIONAMIENTO

Implementación

El predicado `\+` está implementado internamente mediante las dos siguientes cláusulas (una regla y un hecho):

```
\+ P :-
    P,
    !,
    fail.


\+ P.
```

donde `!` es el predicado de corte y `fail` es un predicado predefinido que siempre falla (siempre devuelve *false*).

Funcionamiento

Ante un objetivo de la forma `\+ P`, PROLOG aplica la implementación anterior, y el resultado será el siguiente:

- PROLOG devolverá cierto si `P` no se puede probar, es decir, si al realizarse la consulta "`?- P.`" se obtiene un Árbol de Resolución finito y con todas sus ramas fallo.
- Devolverá falso si `P` es cierto, es decir, si al realizarse la consulta "`?- P.`" se obtiene al menos un éxito (el Árbol de Resolución contiene al menos una rama éxito más a la izquierda que cualquier rama infinita).
- No terminará si la consulta "`?- P.`" no termina (es decir, si al realizarse esta consulta se obtiene un Árbol de Resolución conteniendo una rama infinita sin ninguna rama éxito a su izquierda).

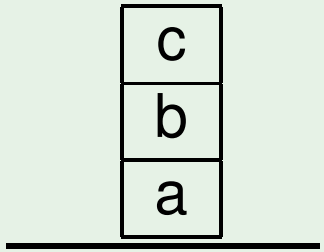
En efecto, dada la implementación del predicado `\+ P`, los nodos de los Árboles de Resolución de la forma `?- \+ P, O1, ..., On.` siempre tendrán los dos hijos siguientes (con u.m.g.'s vacíos):

1. El hijo izquierdo, obtenido con la regla `\+ P :- P, !, fail.`, será `?- P, !, fail, O1, ..., On.` y:
   - Si `P` resulta ser cierto, la respuesta será `false`, puesto que la combinación del corte y el `fail` posteriores hará que todos los hijos del nodo `?- \+ P, O1, ..., On.` sean ramas fallo o podadas (ver ejemplo (3/4) a continuación).
   - Si por el contrario `P` resulta ser falso (todos los posibles intentos de probar `P` fracasan), no se llega a ejecutar ni el corte ni `fail` y PROLOG retrocede para entrar por el hijo derecho (ver ejemplo (2/4) a continuación).
2. El hijo derecho, obtenido con el hecho `\+ P.`, será `?- O1, ..., On.` Sólo se llegará si `P` es falso, y el resultado dependerá de la evaluación de `O1, ..., On`.

Ejemplo (El mundo de los bloques -Blocks World-, 1/4)

Ejemplo clásico en Inteligencia Artificial para ilustrar problemas de planificación y robótica.

```
% encima(X,Y)
% cierto si X está justo encima de Y
encima(b,a).
encima(c,b).

% apilado(X,Y)
% cierto si X está apilado sobre Y
apilado(X,Y) :-  % justo encima
    encima(X,Y).

apilado(X,Y) :-  % más arriba
    encima(X,Z),
    apilado(Z,Y).
```

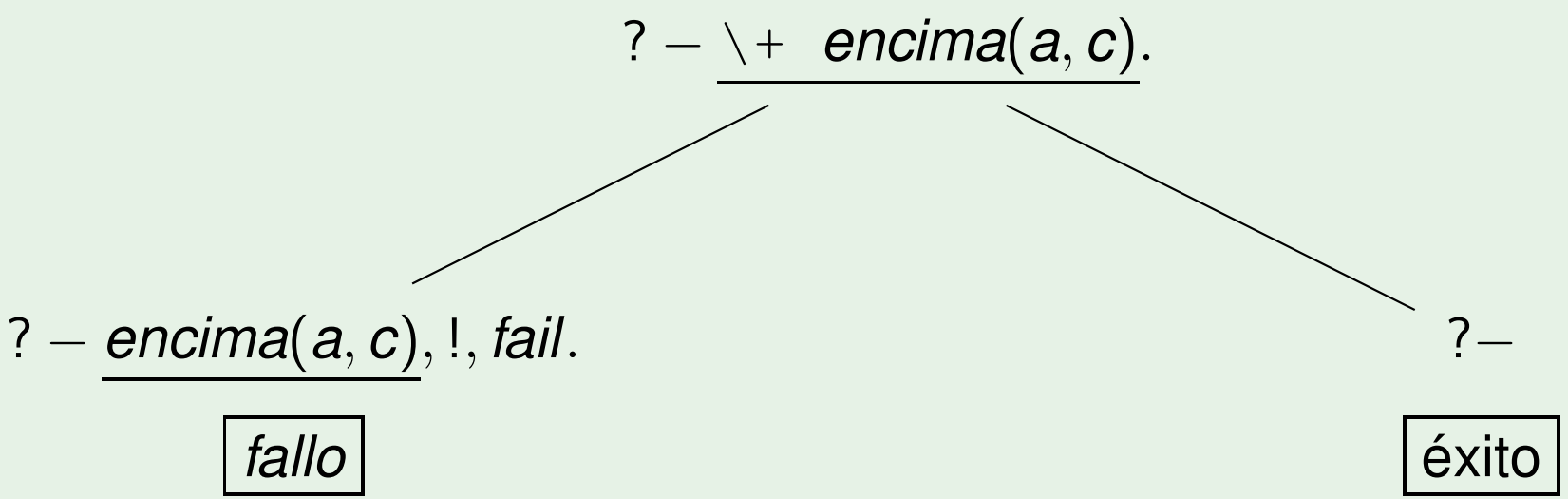


c está encima de b y apilado sobre él

c está apilado sobre a pero no encima suyo

Ejemplo (El mundo de los bloques, 2/4)

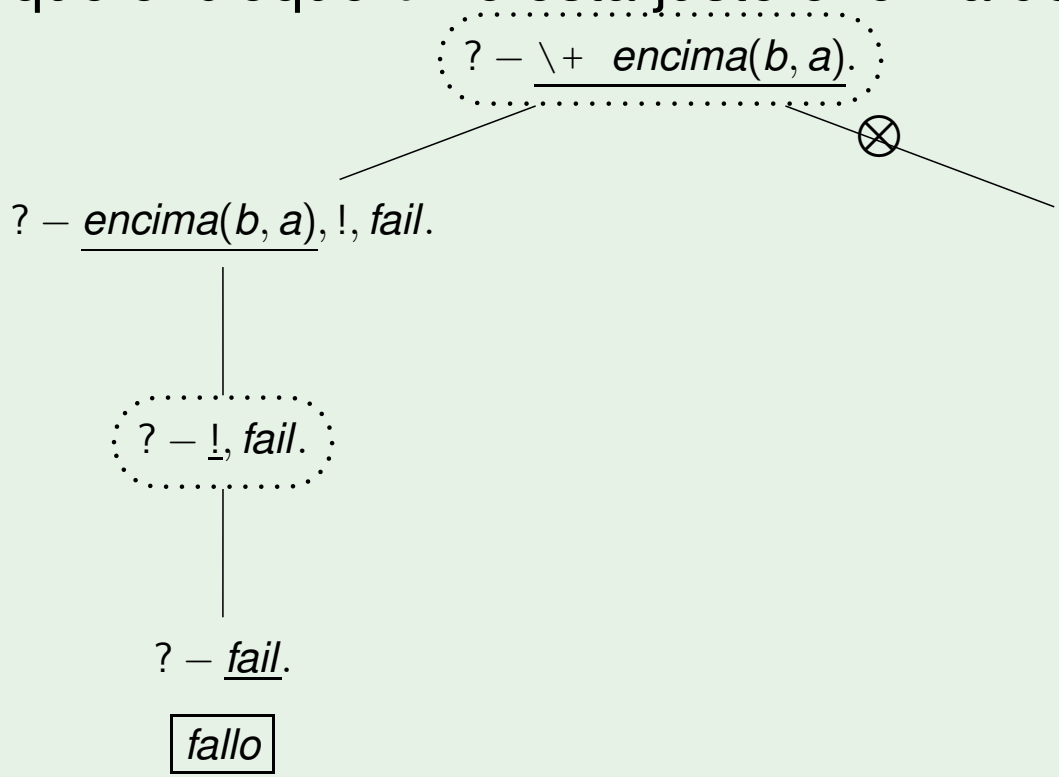
¿Es cierto que el bloque `a` no está justo encima del bloque `c`?



A la vista del árbol anterior, la respuesta de PROLOG es `true`: efectivamente, el bloque `a` no está justo encima del bloque `c`.

Ejemplo (El mundo de los bloques, 3/4)

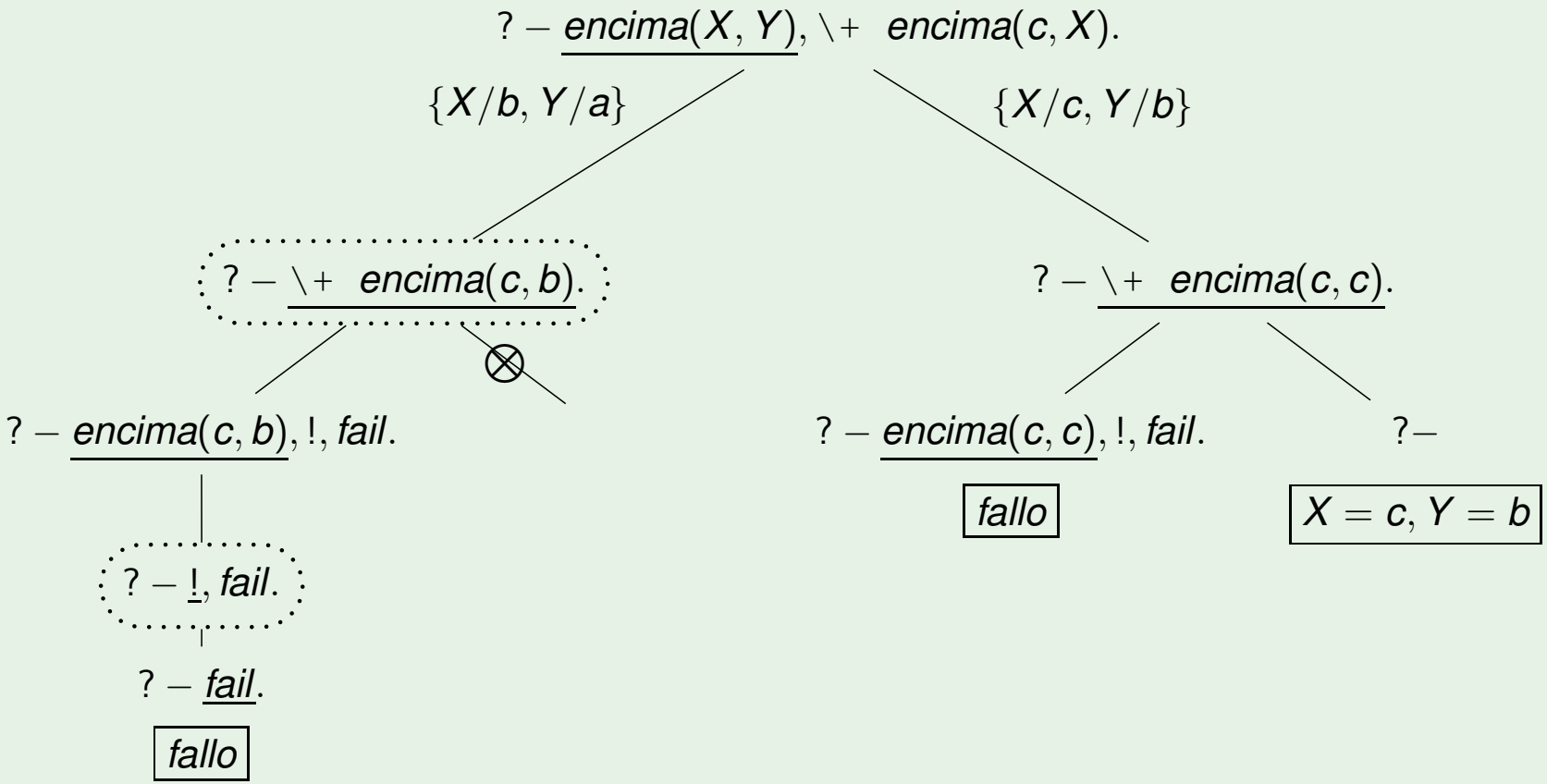
¿Es cierto que el bloque `b` no está justo encima del bloque `a`?



A la vista del árbol anterior, la respuesta de PROLOG es `false`: efectivamente, no es cierto que el bloque `b` no está justo encima del bloque `a`, porque sí que lo está.

Ejemplo (El mundo de los bloques, 4/4)

¿Existen dos bloques tales que el primero está justo encima del segundo y el bloque c no está justo encima del primero?



Respuesta de PROLOG: sí, y la única solución es X=c, Y=b.

Ejercicios (Árboles de Resolución con negaciones)

Considere el programa en PROLOG *relativo al mundo de los bloques del último ejemplo.*

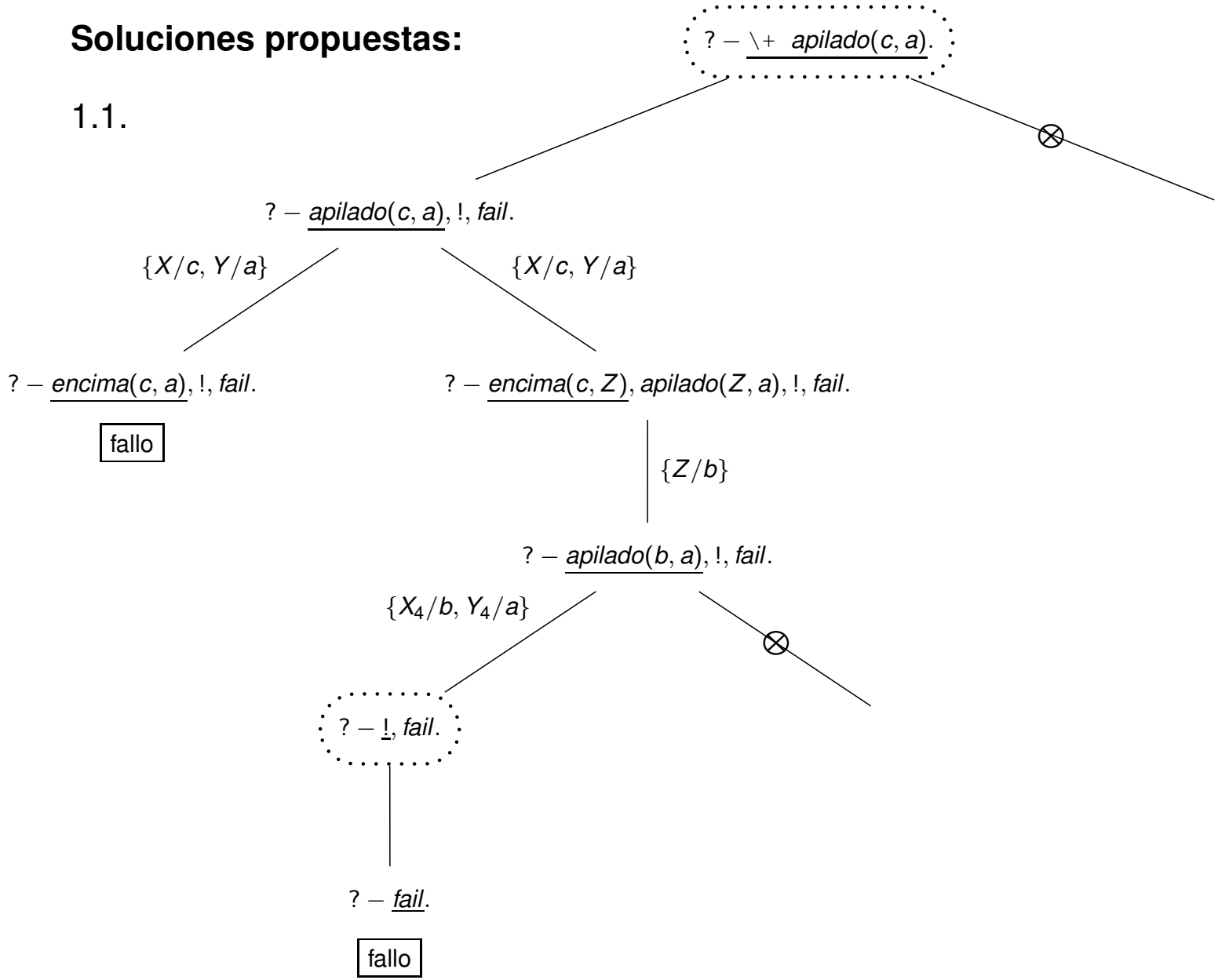
1. *Escriba en* PROLOG *cada una de las consultas que se describen a continuación y construya los Árboles de Resolución adecuados para averiguar qué respuestas ofrecería* PROLOG*, y en qué orden.*
   1. *¿Es cierto que el bloque c no está apilado sobre el bloque a?*
   2. *¿Es cierto que el bloque b no está apilado sobre el bloque c?*
   3. *¿Qué bloques hay tales que están apilados sobre el bloque a pero no están justo encima suyo?*
2. *Reflexione sobre qué respuestas ofrecería* PROLOG *ante la consulta*

   ```
   ?- apilado(X,a), !, \+ encima(X,a).
   ```

   Compruebe su respuesta construyendo el Árbol de Resolución correspondiente.

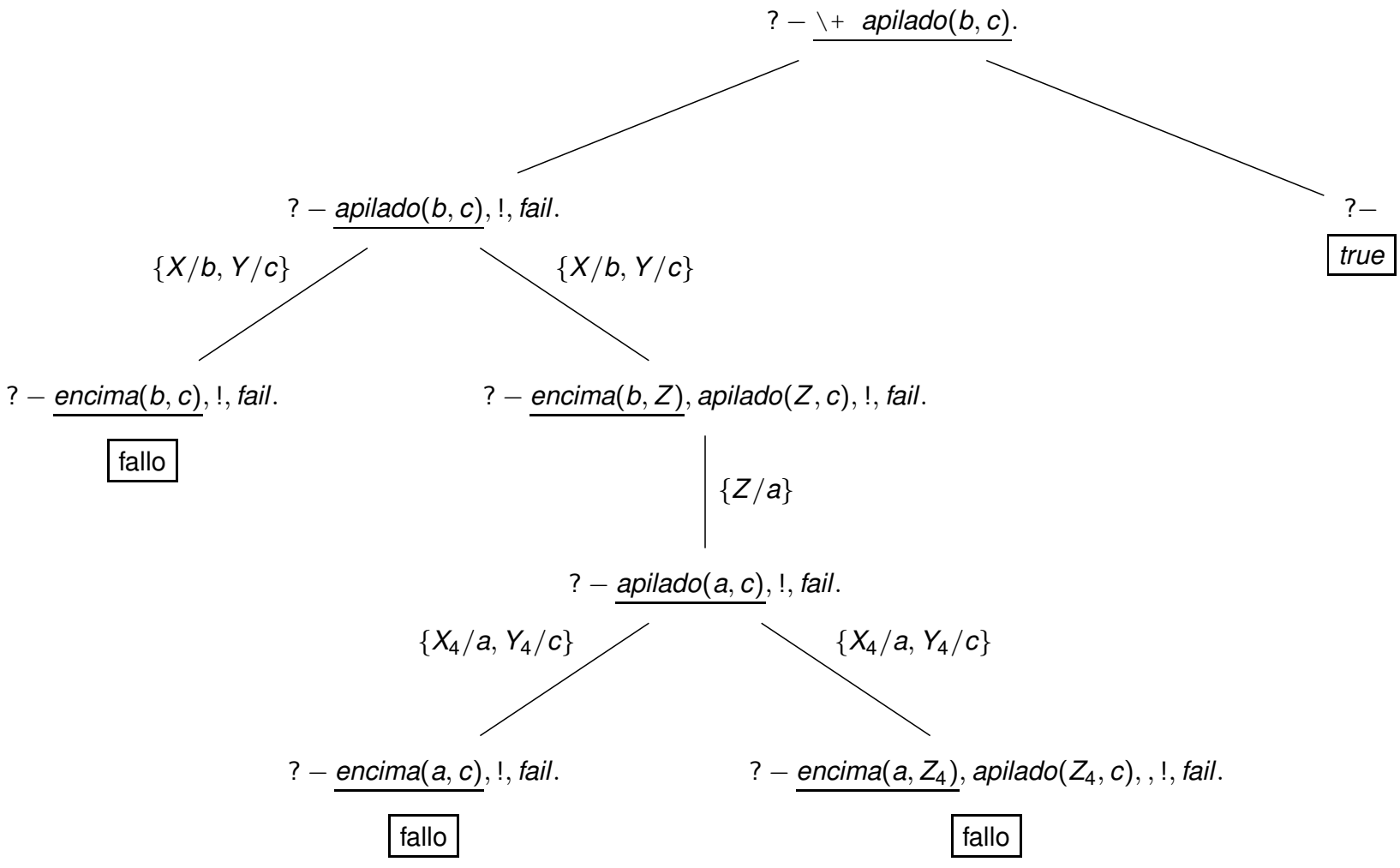
Soluciones propuestas:

1.1.



A la vista del Árbol anterior, la respuesta de PROLOG es `false`: efectivamente, es falso que c no está apilado sobre a, ¡sí que lo está!.

1.2.



A la vista del Árbol anterior, la respuesta de PROLOG es `true`: efectivamente, es cierto que b no está apilado sobre c.

1.3. La consulta correspondiente en PROLOG sería `?- apilado(X,a), \+ encima(X,a)`, dando como única solución `X=c`. El Árbol de Resolución asociado, de izquierda a derecha en profundidad, consta de las siguientes ramas: rama fallo, rama podada, tres ramas fallo, rama éxito con solución `X=c` y dos ramas fallo.

2. Esta consulta es la misma que la última del punto anterior, ¿qué bloques están apilados sobre a pero no están justo encima suyo?, pero de forma que el corte impide la reevaluación del primer subobjetivo, `apilado(X,a)`. Esto hace que la consulta falle, puesto que la primera -y única, debido al corte- respuesta que encuentra `apilado(X,a)` es `X=b`, el bloque que está justo encima suyo, por lo que el siguiente subobjetivo, `\+ encima(b,a)`, falla.

NO MONOTONÍA

El predicado `\+` se basa en la Hipótesis del Mundo Cerrado (Closed Word Assumption, CWA) y permite implementar Razonamiento No Monótono (Non Monotonic Reasoning, NMR).

- Hipótesis del Mundo Cerrado.

  Si de una base de conocimientos no se puede deducir un cierto hecho `P`, entonces ese hecho se puede considerar falso, y por lo tanto su negación, `\+ P`, se puede considerar cierta.

- Razonamiento No Monótono.

  Razonamiento en el que un aumento de las premisas puede provocar una disminución de las conclusiones, cosa que nunca ocurre en el razonamiento de la Lógica Matemática, que es monótono.

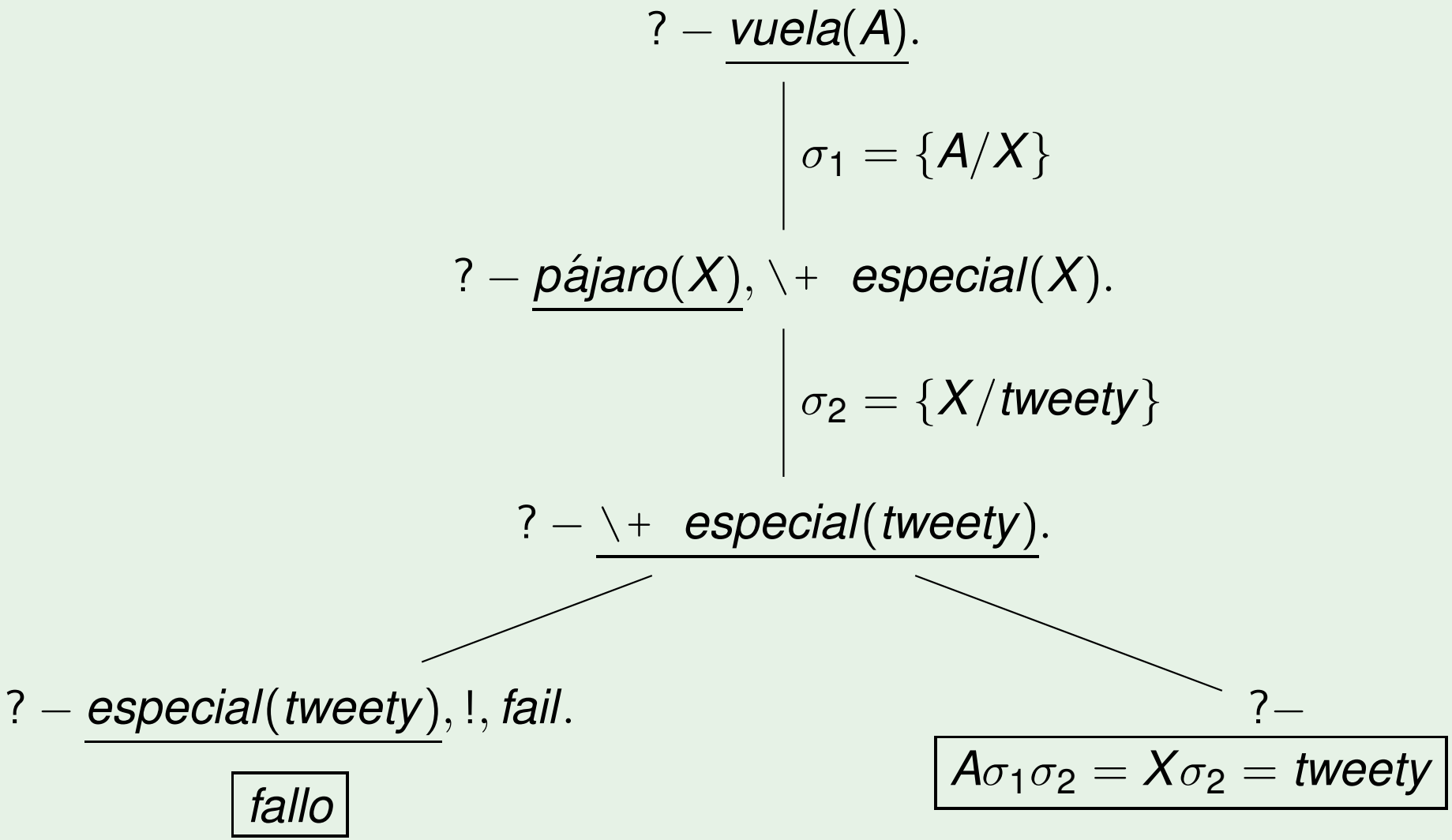
Ejemplo (El pájaro Tweety, 1/3)

Ejemplo clásico en IA para ilustrar razonamiento no monótono.

```
%todos los pájaros vuelan, salvo que sean especiales
        vuela(X) :-
            pájaro(X),
            \+ especial(X).
% tweety es un pájaro
        pájaro(tweety).
% pepito es especial
        especial(pepito).
```

Con la información anterior, PROLOG puede concluir que `tweety` vuela (ver Árbol de Resolución a continuación), dado que sabe que es un pájaro y, *al no poder demostrar que sea especial*, deduce que no lo es (en Lógica Matemática no sería posible demostrar que `tweety` no es especial y por lo tanto no se podría deducir que vuela).

Ejemplo (El pájaro Tweety, 2/3)



Ejemplo (El pájaro Tweety, 3/3)

Si el programa relativo al pájaro Tweety se extiende con las dos siguientes cláusulas adicionales:

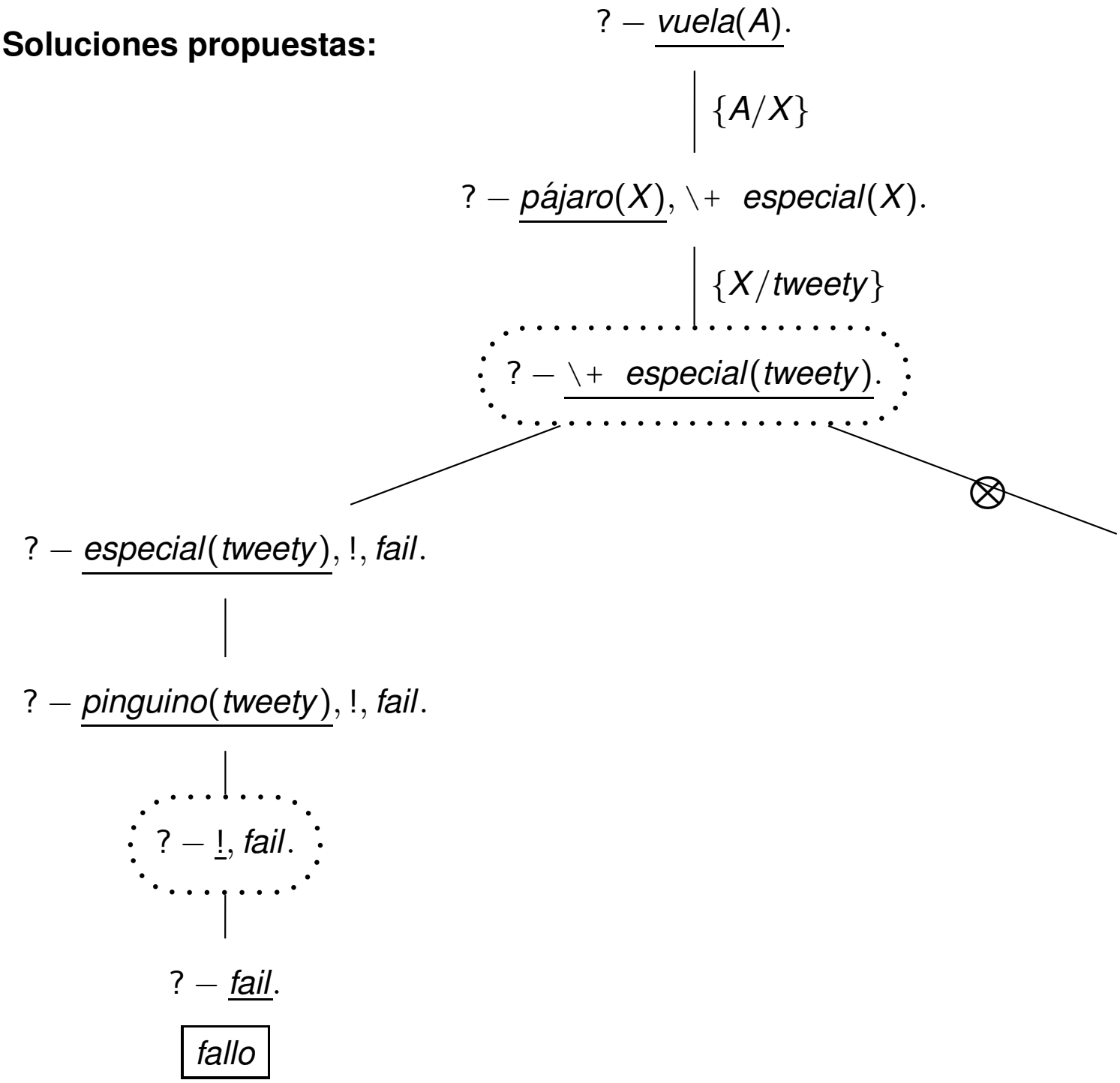
```
especial(X) :-
    pingüino(X).

pingüino(tweety).
```

ya no es posible deducir que Tweety puede volar, puesto que ahora sí se puede demostrar que Tweety es especial. Se trata de un ejemplo de razonamiento no monótono: un aumento de las premisas obliga a retractar una de las conclusiones obtenidas previamente.

Ejercicios

Compruebe lo anterior haciendo el Árbol de Resolución para `?- vuela(A)` *una vez añadidas las nuevas cláusulas.*

Soluciones propuestas:



LIMITACIONES DEL PREDICADO DE NEGACIÓN

El predicado de negación puede funcionar de forma distinta a la esperada en ciertas ocasiones $\Rightarrow$ ¡debe usarse con cuidado!

1. La invocación de un objetivo del tipo "`\+ P`" no asigna valores a variables, es decir, no computa.

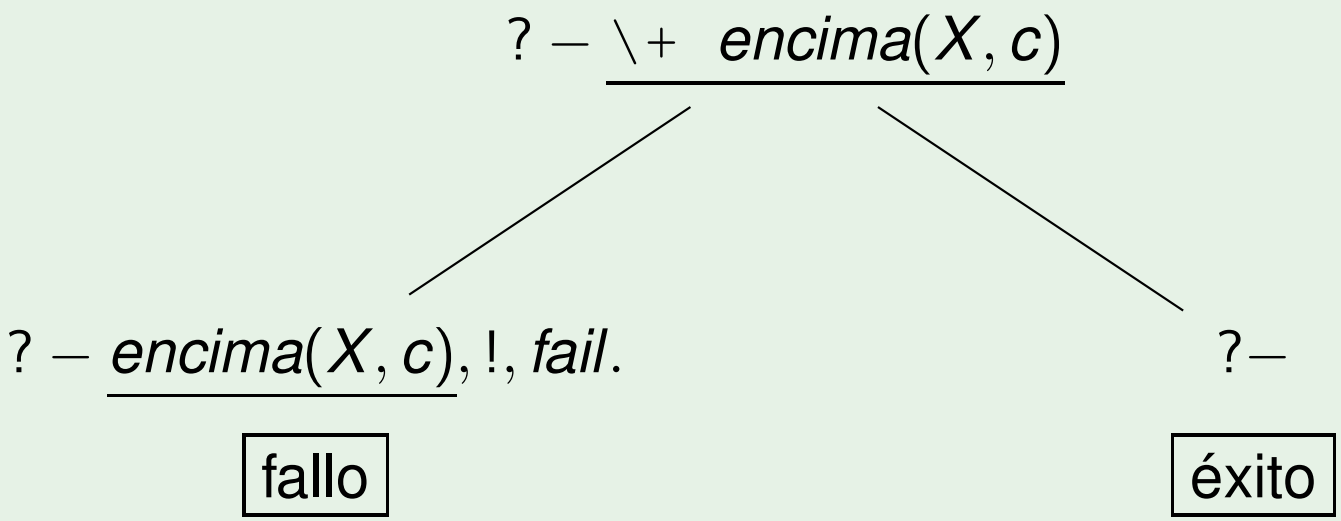
   $\Rightarrow$ Este tipo de objetivos se puede usar para realizar comprobaciones, pero *no para computar*.

2. La invocación de un objetivo del tipo "`\+ P`" con variables sin instanciar (sin valor) puede dar resultados erróneos: ¡PROLOG puede contestar negativamente a una consulta cuya respuesta debiera ser afirmativa!

   $\Rightarrow$ Es importante tener en cuenta este hecho a la hora de programar.

Ejemplo (El mundo de los bloques)

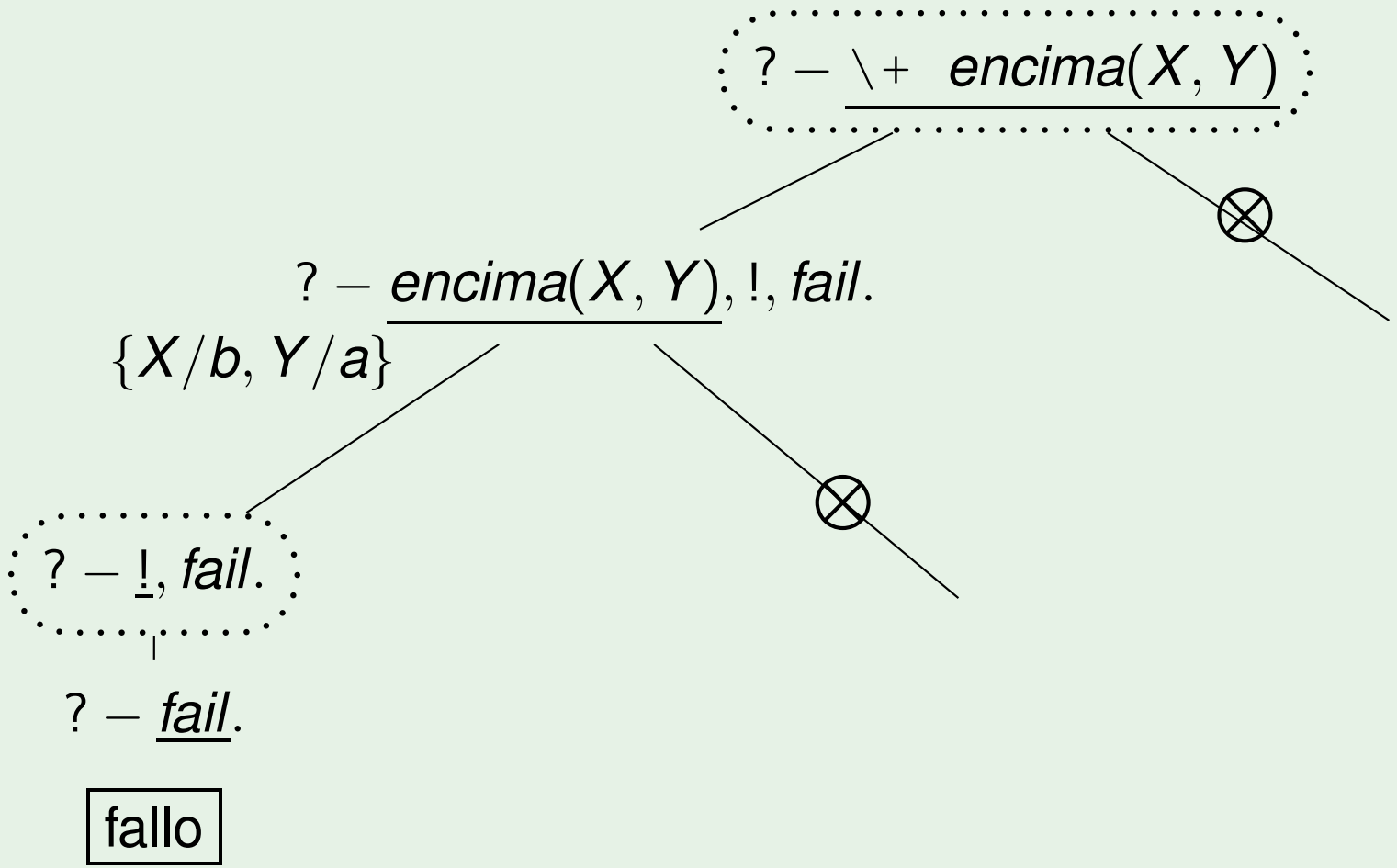
¿Existe algún bloque que no esté justo encima del bloque `c`?



PROLOG contesta `true` y su respuesta es correcta, pero no devuelve los valores de X que hacen cierta la consulta, es decir, ¡no computa!

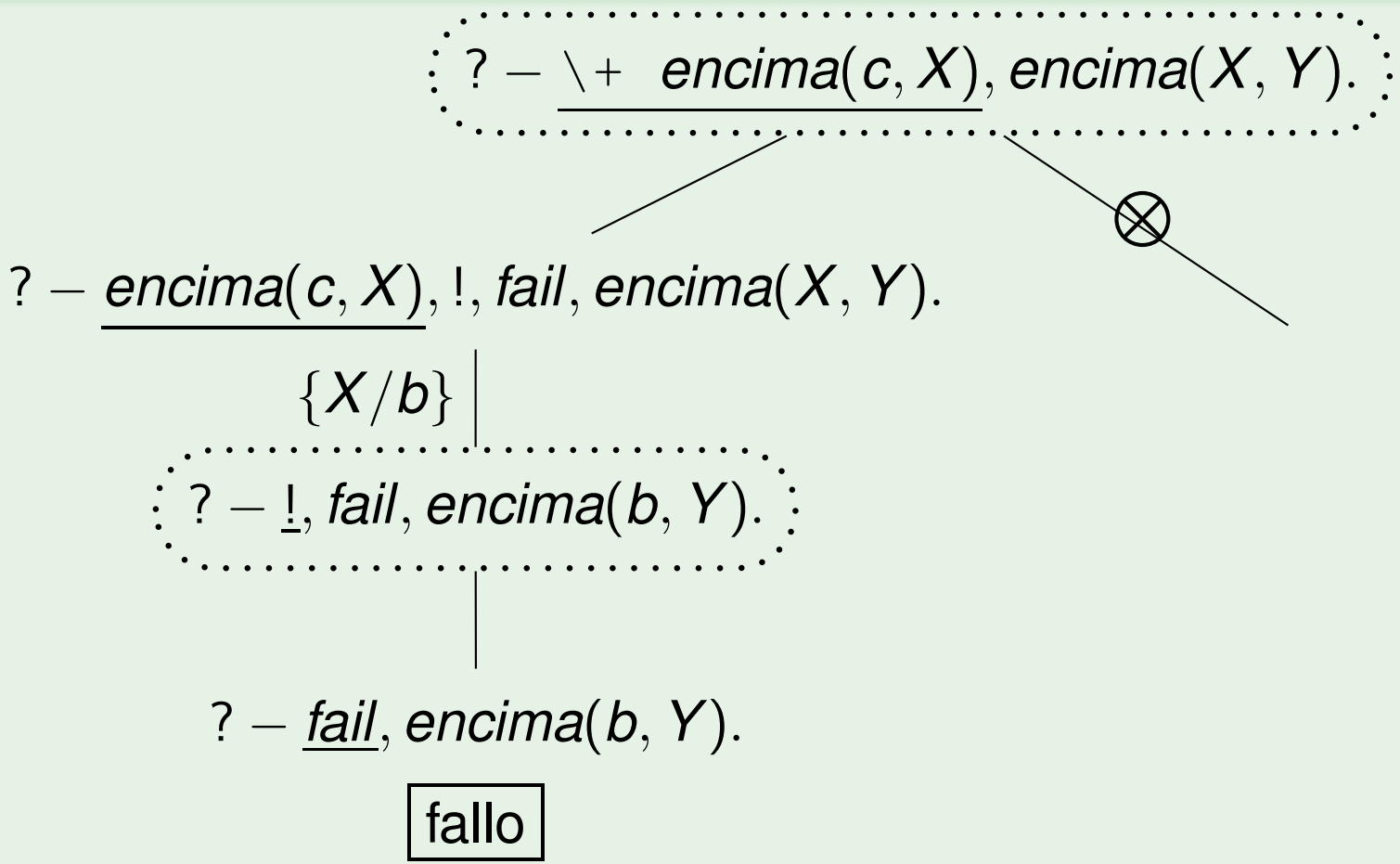
Ejemplo (El mundo de los bloques)

¿Existen dos bloques tales que el 1º no esté justo encima del 2º?



La respuesta obtenida, `false`, es ¡incorrecta!

Ejemplo (El mundo de los bloques)



La respuesta obtenida, `false`, es ¡incorrecta! (compare este árbol con el ▸ Árbol de Resolución hecho previamente para la misma consulta pero con los dos subobjetivos intercambiados).

Las respuestas incorrectas pueden producirse en cualquier nivel de un Árbol de Resolución en el que el predicado a evaluar (el de más a la izquierda) sea un predicado negado que en ese momento contenga alguna variable sin instanciar (sin valor concreto).

$\Longrightarrow$ Siempre que sea posible (no lo es por ejemplo si solo hay un predicado negado, como en la consulta `?- \+ encima(X,Y)` mencionada más arriba), habrá que colocar los predicados negados lo más a la derecha posible: conviene escribir

```
encima(X,Y), \+ encima(c,X).
```

en lugar de

```
\+ encima(c,X), encima(X,Y).
```

ya que en el ▸ primer caso, a diferencia del ▸ segundo, cuando PROLOG evalúa el predicado negado, la variable `X` ya está instanciada.

Ejercicios

Dados el programa

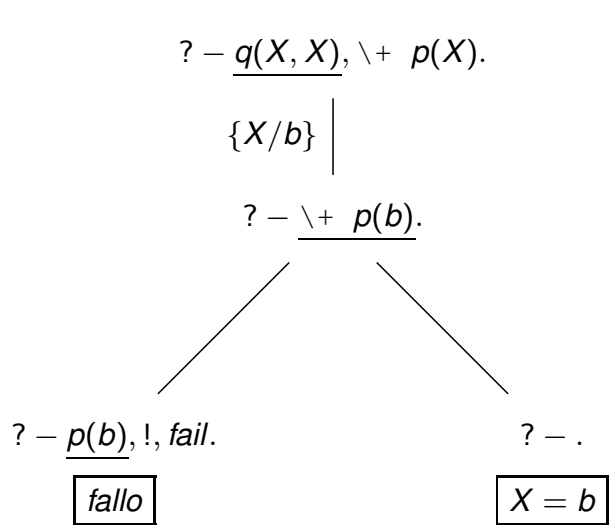
```
p(a).
q(b,b).
```

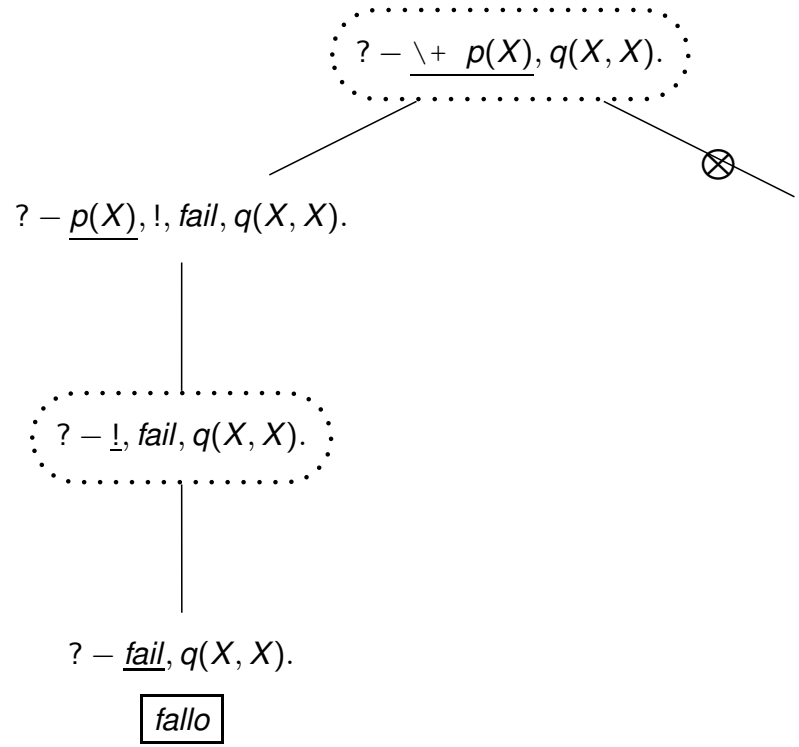
y las consultas (iguales, salvo en el orden de sus objetivos)

1. `?- q(X,X),\+ p(X).`
2. `?- \+ p(X), q(X,X).`

construya los Árboles de Resolución pertinentes para averiguar qué contestaría PROLOG *y explique las diferencias en las respuestas obtenidas.*

Soluciones propuestas:





PROLOG construye los árboles anteriores y contesta `X=b` y `false` (indicando que no hay más soluciones) en la primera consulta y `false` en la segunda. La diferencia en las dos respuestas anteriores se debe a que en el segundo caso se está resolviendo un predicado negado, `\+ p(X)`, con una variable que no está instanciada, y en estos casos PROLOG no siempre funciona correctamente (porque interpreta las variables como si estuvieran cuantificadas universalmente en lugar de existencialmente).

Ejercicios

Dado el programa

```
q(a,b).
q(b,c).
q(a,c).
p(X,Z) :- q(X,Z).
p(X,Z) :- q(X,Y), p(Y,Z).
```

construya los Árboles de Resolución pertinentes para averiguar el comportamiento de PROLOG *ante las siguientes consultas:*

1. `?- \+ p(V,c), q(a,V).`
2. `?- q(a,V), \+ p(V,c).`

¿Qué ocurriría si en las consultas anteriores se añadiese un corte entre los dos subobjetivos?

Soluciones propuestas:

La respuesta de PROLOG ante la primera consulta es `false` (el Árbol de Resolución correspondiente tiene, de izquierda a derecha, una rama fallo y tres ramas podadas por un corte). La respuesta es errónea debido a que se evalúa el predicado negado `\+ p(V,c)` con la variable `V` sin instanciar.

La respuesta de PROLOG ante la segunda consulta es `V=c` y a continuación `false` indicando que no hay más soluciones (el Árbol de Resolución correspondiente tiene, de izquierda a derecha, una rama fallo, dos ramas podadas por un corte, dos ramas fallo y una rama éxito con solución `V=c`). La respuesta ahora es correcta porque ya no se evalúa `\+ p(V,c)` sino `\+ p(b,c)` (que resulta ser falso) y después `\+ p(c,c)` (que da cierto).

Si se añadiese un corte entre los dos subobjetivos:

- En la primera consulta no afectaría en nada, nunca se llegaría a evaluar ese corte puesto que el primer subobjetivo falla.
- En la segunda consulta el corte introducido podaría el hijo derecho de la raíz, que es el que encuentra la solución, por lo que la respuesta de PROLOG pasaría a ser `false`.

Ejercicios

Considere el programa PROLOG

```
p(A,B) :- q(A), !, r(B,A).   |   q(1).
r(A,B) :- q(A).              |   q(2).
t(2).                        |   q(3).
```

y la consulta

```
?- p(Y,X), \+ t(X).
```

1. *¿Qué se pretende averiguar con esa consulta?*
2. *Construya el Árbol de Resolución correspondiente.*
3. *Deduzca de lo anterior qué respuesta(s) ofrecería* PROLOG *ante la consulta dada, y en qué orden las facilitaría.*

Soluciones propuestas:

1. El objetivo de la consulta es averiguar si, dado el conocimiento expresado en el programa, existen objetos Y y X tales que el primero está relacionado con el segundo mediante la relación “p” de forma que el segundo, además, no cumple la propiedad “t”.
2. La construcción paso a paso del Árbol de Resolución solicitado está disponible aquí.
3. Ante una consulta, PROLOG construye el árbol de Resolución, en profundidad, y facilita las soluciones según las va encontrando. Por lo tanto, ante la consulta dada, y a la vista del árbol anterior, la respuesta de PROLOG será la siguiente:

Y=1, X=1 ;% primera solución encontrada
Y=1, X=3 ;% segunda solución encontrada
false % indicando que no quedan más soluciones

BIBLIOGRAFÍA

- L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.
- W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fifth edition, 2003.
- I. Bratko.*Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 2001.
- J. Lloyd. *Foundations of Logic Programming*, (Second Edition). Springer-Verlag, 1987.
- R. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
- U. Nilsson and J. Maluszynski. Logic, Programming and Prolog. John Wiley & Sons Ltd, 1996.
- SWI-Prolog, entorno de programación en Prolog de dominio público.
- comp.lang.prolog. Faq

PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL3: El lenguaje PROLOG, aspectos avanzados

2. Recolección de soluciones

Grado en Ingeniería Informática

URJC

Ana Pradera

Contenido

1. INTRODUCCIÓN
2. EL PREDICADO bagof
3. EL PREDICADO setof
4. EL PREDICADO findall
5. EJERCICIOS

INTRODUCCIÓN

- Cuando un problema tiene más de una solución, a menudo es muy conveniente poder recolectar todas las posibles soluciones para su posterior tratamiento (visualizarlas, almacenarlas, ordenarlas, transformarlas, contarlas, etc).
- PROLOG ofrece tres predicados predefinidos básicos para recolectar soluciones: `bagof(?T, +Obj, ?L)`, `setof(?T, +Obj, ?L)` y `findall(?T, +Obj, ?L)`.
- Los tres predicados anteriores son ciertos si `L` es una **lista** almacenando **todas** las instancias del término `T` (variable o término compuesto) **que cumplen el objetivo** `Obj`.
- Para ello, construyen el **Árbol de Resolución** correspondiente al objetivo `Obj` retrocediendo automáticamente para generar todas las posibles soluciones.
- Difieren entre ellos en algunos detalles que se explican e ilustran a continuación.

Ejemplo (Programa “Gustos”)

```
% gusta(?X,?Y)
% cierto si a X le gusta el lenguaje Y

gusta(pepa, prolog).
gusta(pepa, haskell).
gusta(pepito, java).
gusta(pepita, prolog).
gusta(pepita, scala).
gusta(pepin, prolog).
gusta(pepin, Algo) :-
        gusta(pepita, Algo).
```

EL PREDICADO bagof(?T, +Obj, ?L)

Cierto si `L` es una lista formada por todas las instancias del término `T` para las que se cumple el objetivo `Obj`.

Ejemplo

```
% Personas a las que les gusta Scala
?- bagof(P, gusta(P, scala), Scaleros).
Scaleros = [pepita, pepin]
```

Si `Obj` no es cierto para ninguna instancia de `T`, el predicado `bagof` falla (la lista `L` de un `bagof` nunca será vacía).

Ejemplo

```
% Personas a las que les gusta Pascal
?- bagof(X, gusta(X, pascal), Pascaleros).
false
```

La lista `L` puede contener soluciones repetidas (*bag* = bolsa) y el orden en el que aparecen las soluciones en la lista *depende del intérprete* (normalmente es el orden en el que PROLOG las encuentra).

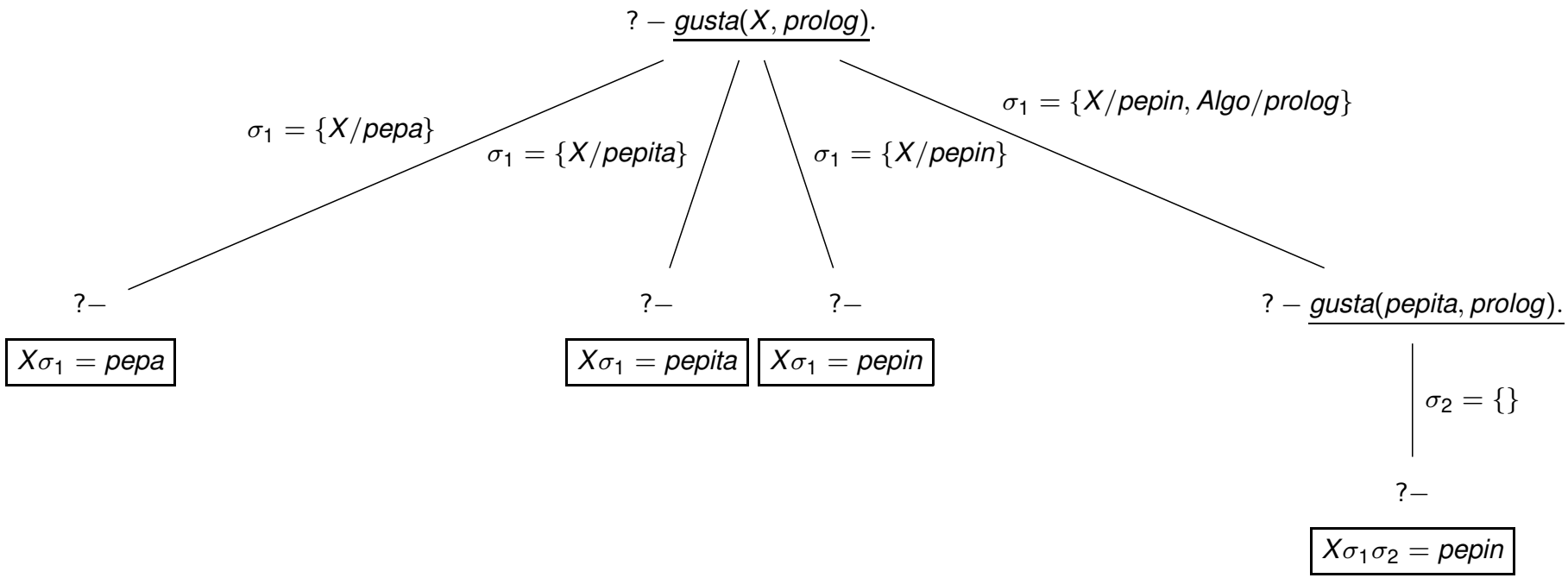
Ejemplo

```
 % Personas a las que les gusta Prolog
?- bagof(X, gusta(X, prolog), Prologueros).
Prologueros = [pepa, pepita, pepin, pepin]
```

Ejercicios

Construya el Árbol de Resolución correspondiente a la consulta `?- gusta(X, prolog)` *para comprender por qué pepin aparece dos veces en la lista del ejemplo anterior. Observe también que las soluciones aparecen en* `Prologueros` *de acuerdo con el recorrido en profundidad del Árbol de Resolución.*

Soluciones propuestas:



A la vista del Árbol anterior, las respuestas de PROLOG serían:
1) `X=pepa`, 2) `X=pepita`, 3) `X=pepin`, 4) `X=pepin` (y `false` indicando que no hay más soluciones).

En efecto, se puede demostrar que a `pepin` le gusta PROLOG de dos formas distintas: directamente mediante el último hecho, o bien mediante la regla “a pepin le gusta todo lo que le gusta a pepita”, unida al hecho de que a `pepita` le gusta PROLOG.

`Obj` puede ser un objetivo compuesto por varios subobjetivos colocados entre paréntesis y conectados tanto de forma conjuntiva ("y") como disyuntiva ("o"):

- El predicado predefinido `,/2` se usa en notación infija y representa la *conjunción* de sus dos argumentos, es decir, el objetivo `(Obj1 , Obj2)` será cierto si y sólo si tanto `Obj1` como `Obj2` son ciertos.
- El predicado predefinido `;/2` se usa en notación infija y representa la *disyunción* de sus dos argumentos, es decir, el objetivo `(Obj1 ; Obj2)` será cierto si y sólo si o bien `Obj1` o bien `Obj2` (o ambos) son ciertos.
- Ambos se pueden **extender** a *n* argumentos, es decir, es posible escribir objetivos de la forma `(Obj1 , Obj2 , ... , Objn)` o `(Obj1 ; Obj2 ; ... ; Objn)`, y por supuesto **combinarlos** entre sí, escribiendo, por ejemplo, `((Obj1, Obj2) ; Obj3).`

Ejemplo

```
% Personas a las que les gusta scala Y prolog
?- bagof(X,
         (gusta(X, scala), gusta(X, prolog)),
         L).
L = [pepita, pepin, pepin].


% Personas a las que les gusta scala O prolog
?- bagof(X,
          (gusta(X, scala); gusta(X, prolog)),
          L).
L = [pepita, pepin, pepa, pepita, pepin, pepin]
```

En la última consulta `pepita` aparece dos veces porque le gustan tanto scala como prolog, y pepin aparece tres veces porque le gustan tanto scala como prolog y este último además por dos motivos distintos.

Si `Obj` tiene variables libres (sin valor asignado) distintas a las que aparecen en `T`, el predicado `bagof` dará varias soluciones, una para cada posible valor de la variable libre con algún resultado.

Ejemplo

```
?- bagof(X, gusta(X,Y), L).     % Y está libre
L = [pepa], Y = haskell
L = [pepito], Y = java
L = [pepa, pepita, pepin, pepin], Y = prolog
L = [pepita, pepin], Y = scala
```

Pero cuidado porque no toda variable que aparece en `Obj` (y no en `T`) está libre, puesto que se le puede haber asignado valor antes:

Ejemplo

```
?- Y = scala, bagof(X, gusta(X,Y), L).
 L = [pepita, pepin], Y = scala
```

Es posible cuantificar existencialmente una o más variables de `Obj`, escribiendo `Y1^...Yn^(Obj)`, cuyo significado es "existen `Y1 ... Yn` tales que se cumple `Obj`".

Ejemplo

```
% Personas a las que les gusta algún lenguaje
?- bagof(X, Y^gusta(X,Y), L).
L = [pepa, pepa, pepito, pepita, pepita,
     pepin, pepin, pepin]

% Personas a las que les gustan al menos 2 lenguajes
?- bagof(P,
         L1^L2^(gusta(P,L1),gusta(P,L2),L1 @< L2),
         L).
L = [pepa, pepita, pepin, pepin]
```

La comprobación `L1 @< L2`, L1 menor alfabéticamente que L2, evita la repetición de soluciones en las que L1 y L2 tienen los mismos valores en distinto orden.

El término `T` no tiene por qué ser solo una variable, puede ser un término compuesto.

Ejemplo

```
?- bagof((P,X),
         (gusta(P,X), (X=prolog ; X=scala)),
         D).
D = [(pepa,prolog), (pepita,prolog), (pepita,scala),
     (pepin,prolog), (pepin,prolog), (pepin,scala)]

?- bagof(gustos(Y,L),
         bagof(X,gusta(X,Y),L),
         LL).
LL = [gustos(haskell,[pepa]), gustos(java,[pepito]),
      gustos(prolog,[pepa, pepita, pepin, pepin]),
      gustos(scala,[pepita, pepin])]
```

La última consulta muestra que el objetivo `Obj` puede ser -en general, contener-, a su vez, un recolector de soluciones.

EL PREDICADO `setof(?T, +Obj, ?L)`

Funciona exactamente igual que `bagof` salvo:

1. `L` aparece ordenada de forma ascendente.
2. `L` no contiene repeticiones (*set* = conjunto).

Ejemplo (Uso de setof en comparación con bagof)

```
?- setof(X, gusta(X, prolog), L).
 L = [pepa, pepin, pepita].

?- setof(X, Y^gusta(X,Y), L).
 L = [pepa, pepin, pepita, pepito].

?- setof(X, (gusta(X, scala) ; gusta(X, prolog)), L).
L = [pepa, pepin, pepita].
```

EL PREDICADO `findall(?T, +Obj, ?L)`

Funciona exactamente igual que `bagof` salvo:

1. Todas las variables libres de `Obj` (salvo las que están en `T`) se consideran automáticamente cuantificadas existencialmente.
2. Si `Obj` no es cierto para ninguna instancia de `T`, `findall` devuelve una lista vacía (en lugar de fallar como hace `bagof`).

Ejemplo (Uso de findall en comparación con bagof)

```
?- findall(X, gusta(X,Y), L).
% equivalente a bagof(X, Y^gusta(X,Y), L).
L = [pepa,pepa,pepito,pepita,pepita,
     pepin,pepin,pepin]

?- findall(X, gusta(X, pascal), L).
 % lista vacía cuando no hay ninguna solución
 L = [].
```

Ejercicios (Recolección de soluciones)

Dado el programa ▸ Gustos*, utilice los predicados de recolección de soluciones que considere oportunos para implementar los siguientes predicados:*

1. `numforofos(+Y, ?N)`, *cierto si* `N` *es el número de personas a las que les gusta el lenguaje* `Y`.
2. `lenguajes(-L)`, *cierto si* `L` *contiene el conjunto de todos los lenguajes que gustan a al menos una persona (lista vacía si no hubiese ninguno).*
3. `gustos(-L)`, *cierto si* `L` *es una lista conteniendo todos las listas [P,LL] donde P es una persona y LL es la lista de lenguajes que gustan a P.*

Soluciones propuestas:

1. `numforofos(+Y, ?N)`, cierto si `N` es el número de personas a las que les gusta el lenguaje `Y`.

   numforofos(Y, N) :- setof(X, gusta(X, Y), L), !, length(L,N).
   numforofos(_, 0).

   - setof en lugar de findall o bagof para evitar repeticiones si a una misma persona (como a pepin) le gustase un mismo lenguaje por varios motivos.

   - Como setof falla cuando no encuentra ningún resultado, y numforofos no debe fallar sino devolver 0, se añade el segundo hecho, al que sólo se llegará (gracias al corte de la cláusula anterior) si setof falla.

2. `lenguajes(-L)`, cierto si `L` contiene el conjunto de todos los lenguajes que gustan a al menos una persona (lista vacía si no hubiese ninguno).

   lenguajes(L) :- setof(Y, X^gusta(X,Y), L), !.
   lenguajes([]).

   - Es necesario usar el predicado `setof` para evitar repeticiones y así obtener, como se pide en el enunciado, un *conjunto* de lenguajes.
   - Como setof falla cuando no encuentra ningún resultado, y en ese caso L debería ser el conjunto vacío, se añade el hecho siguiente, al que sólo se llegará (gracias al corte anterior) si setof falla.

3. `gustos(-L)`, cierto si `L` es una lista conteniendo todos las listas [P,LL] donde P es una persona y LL es la lista de lenguajes que gustan a P.

```
gustos(L) :-
bagof(
[P,LL],
setof(Leng, gusta(P,Leng),LL),
L).
```

Observe que el `bagof` anterior se podría reemplazar por `setof` (que ordenaría las soluciones) o por `findall` (que devolvería una lista vacía, en lugar de fallar, si no hubiese ningún par [P,LL]), dado que el enunciado no detalla qué debe pasar en este caso.

Sin embargo, el `setof` interno no se puede reemplazar ni por `bagof` (por si a una persona le gusta un lenguaje por varios motivos) ni por `findall`. Este último cuantificaría la variable P ("dame todos aquellos Leng para los que *existe* un P tal que a P le gusta Leng") y se obtendría como solución `[(_, [prolog, haskell, java, prolog, scala, prolog, prolog, scala])]`, es decir, una lista con un único par donde la P es cualquiera y su lista asociada está compuesta por todos los lenguajes que gustan a alguien (con repeticiones).

Ejercicios (Recolección de soluciones)

Implemente los siguientes predicados:

1. `concisoYordenado(+L, ?Conj)`, *cierto si* `L` *es una lista no vacía y* `Conj` *es una lista conteniendo los mismos elementos que* `L` *pero sin repeticiones y ordenados de menor a mayor.*
2. `esconjunto(+L)`, *cierto si L es un conjunto, es decir, L es una lista que no contiene elementos repetidos.*
3. `interseccion(+L1, +L2, L)`, *cierto si L es un conjunto conteniendo todos los elementos comunes a L1 y L2. Tenga en cuenta que L no debe contener elementos repetidos, y que si L1 y L2 no tienen elementos comunes, L debe ser igual a la lista vacía.*
4. `union(+L1, +L2, L)`, *cierto si L es un conjunto conteniendo todos los elementos que están en L1 o bien en L2 (o en ambos). Tenga en cuenta que L no debe contener elementos repetidos, y que si L1 y L2 son vacías, L debe ser también vacía.*

Soluciones propuestas:

1. `concisoYordenado(+L, ?Conj)`, cierto si `L` es una lista no vacía y `Conj` es una lista conteniendo los mismos elementos que `L` pero sin repeticiones y ordenados de menor a mayor.

   concisoYordenado([C|R], Conj) :- setof(X, member(X, [C|R]), Conj).

2. `esconjunto(+L)`, cierto si L es un *conjunto*, es decir, L es una lista que no contiene elementos repetidos.

   esconjunto([]) :- !.
   esconjunto(L) :-
   setof(X, member(X,L), LConj), length(LConj, LC), length(L, LC).

3. `interseccion(+L1, +L2, L)`, cierto si L es un *conjunto* conteniendo todos los elementos comunes a L1 y L2.

   interseccion(L1, L2, L) :- setof(X, (member(X,L1), member(X,L2)), L), !.
   interseccion(_, _, []).

4. `union(+L1, +L2, L)`, cierto si L es un *conjunto* conteniendo todos los elementos que están en L1 o bien en L2 (o en ambos).

   union(L1, L2, L) :- setof(X, (member(X,L1) ; member(X,L2)), L), !.
   union(_, _, []).

   Tanto en `interseccion` como en `union` la segunda cláusula, protegida por el corte de la primera, es necesaria porque setof falla cuando ningún X cumple el objetivo, es decir cuando la intersección/unión es vacía, y en esos casos los predicados no deben fallar sino devolver la lista vacía.

Ejercicios (Recolección de soluciones)

- *Ejercicio nº 1, apartado 1.1, de la Práctica de* PROLOG *nº 4.*
- *Ejercicio nº 2, apartados 2.1, 2.2 y 2.3, de la Práctica de* PROLOG *nº 4.*

Soluciones propuestas:

Las soluciones comentadas a los ejercicios anteriores pueden consultarse en Práctica de PROLOG nº 4 con soluciones).

BIBLIOGRAFÍA

- L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.
- W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fifth edition, 2003.
- I. Bratko.*Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 2001.
- J. Lloyd. *Foundations of Logic Programming*, (Second Edition). Springer-Verlag, 1987.
- R. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
- U. Nilsson and J. Maluszynski. Logic, Programming and Prolog. John Wiley & Sons Ltd, 1996.
- SWI-Prolog, entorno de programación en Prolog de dominio público.
- comp.lang.prolog. Faq

PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL-3: El lenguaje PROLOG, aspectos avanzados

3. Predicados de orden superior

Grado en Ingeniería Informática

URJC

Ana Pradera

Contenido

1. MOTIVACIÓN Y DEFINICIÓN
2. PREDICADOS DE ORDEN SUPERIOR BÁSICOS
3. PREDICADOS DE ORDEN SUPERIOR CLÁSICOS
   - Operaciones de aplicación (familia map/N)
   - Operaciones de filtrado
   - Operaciones de reducción o plegado
4. NUEVOS PREDICADOS DE ORDEN SUPERIOR

MOTIVACIÓN Y DEFINICIÓN

- Un predicado de orden superior es un predicado tal que al menos uno de sus argumentos, en lugar de ser un término, es un predicado.
- La idea procede de la *Lógica Matemática* y fue introducida en programación por los *lenguajes funcionales*, extendiéndose su uso posteriormente a otros paradigmas.
- Objetivo: producción de código no redundante y reutilizable, por medio de “plantillas” genéricas.
- Beneficios: se consigue un código más conciso, fiable y fácil de mantener.

Ejemplo

```
% todos_nat(+L)
% todos los elementos de L
% son números naturales

todos_nat([]).

todos_nat([C|R]) :-
    natural(C),
    todos_nat(R).

natural(C) :-
    integer(C), C >= 0.
```

```
% sin_variables(+L)
% L no contiene ninguna
% variable.

sin_variables([]).

sin_variables([C|R]) :-
    nonvar(C),
    sin_variables(R).
```

¡Las dos implementaciones anteriores son prácticamente iguales!

Ambas comprueban si todos los elementos de L cumplen algo

- No tiene sentido repetir *n* veces un código que solo se diferencia en qué deben cumplir los elementos de la lista.
- Conviene diseñar un predicado "genérico" capaz de comprobar si todos los elementos de una lista cumplen un cierto objetivo `Obj`:

```
% map(+Obj, +L)
% cierto si todos los elementos de la lista L
% cumplen el objetivo Obj

map(Obj, []).
map(Obj, [C|R]) :-
  <ejecución del objetivo Obj sobre el valor C>,
  map(Obj, R).
```

y definir los anteriores (y otros similares) por medio de él:

```
todos_nat(L) :- map(natural, L).
sin_variables(L) :- map(nonvar, L).
```

- PROLOG hace un uso extenso de predicados de orden superior:
  - El predicado de negación `\+`
  - Los predicados de recolección de soluciones `bagof`, `setof` y `findall`
  - El predicado `map/2` anterior, denominado `maplist/2` en SWI-Prolog (así como otros predicados predefinidos similares, a menudo conocidos como *predicados de orden superior* ***clásicos***, que se verán más adelante)

  son todos ellos predicados de orden superior.

- Prolog facilita además herramientas (*predicados de orden superior* ***básicos***) que permiten a los programadores diseñar fácilmente sus propios predicados de orden superior.

PREDICADOS DE ORDEN SUPERIOR BÁSICOS

Familia de predicados `call/N`, con `N = 1,2,3,...`

- Los valores de `N` admitidos dependen del intérprete.
- Para `N=1`: `call(+Obj)` intenta ejecutar `Obj`, produciendo un error en caso de que no sea un objetivo ejecutable.
- Para `N>1`: `call(+Obj, +Extra1, +Extra2, ...)` añade los valores `Extra1, Extra2, ...`, en el orden dado, detrás de los argumentos propios de `Obj`, si este los tuviera, e intenta ejecutar el objetivo resultante, produciendo un error en caso de que no sea ejecutable.

Ejemplo (Usos de `call/N`)

Suponga disponible el predicado `gusta/2`:

```
?- call(gusta(X, prolog)). % N=1, Obj=gusta(X,prolog)
?- call(gusta(X), prolog). % N=2, Obj=gusta(X)
?- call(gusta, X, prolog). % N=3, Obj=gusta

% las tres consultas anteriores son equivalentes,
% todas ellas ejecutan "gusta(X, prolog)".

?- call(gusta, X).
% intenta ejecutar "gusta(X)"
ERROR: procedure `gusta(A)' does not exist

?- call(gusta(X), prolog, scala).
% intenta ejecutar "gusta(X,prolog,scala)"
ERROR: procedure `gusta(A,B,C)' does not exist
```

Utilidad de los predicados `call/N`

Los predicados `call/N` se usan fundamentalmente tomando `N>1` para construir y resolver objetivos en tiempo de ejecución y así poder:

- Implementar predicados de orden superior clásicos como los de *aplicación, filtrado y reducción* procedentes de la programación funcional y discutidos a continuación.
- Implementar nuevos predicados de orden superior cuando se detecte la necesidad de utilizar códigos muy similares entre sí que solo se diferencian en algunos detalles parametrizables. Este uso se ilustra al final de esta presentación con varios ejemplos.

Observación

La combinación "`Exe =.. [Obj,C], call(Exe)`" usada anteriormente por motivos técnicos (y que puede aparecer por ello en algunos ejercicios de examen) es equivalente a `call(Obj, C)`.

PREDICADOS DE ORDEN SUPERIOR CLÁSICOS SOBRE LISTAS

Operaciones de aplicación (familia map/N)

- Para `N=2` se obtiene el predicado discutido previamente, `map(+Obj,?L)`, cierto si el objetivo `Obj` se puede aplicar con éxito sobre **todos** los elementos de la lista `L`.
- Para `N=3` se obtiene el predicado `map(+Obj,?L1,?L2)`, cierto si el objetivo `Obj` se puede aplicar con éxito sobre **todas las parejas** de elementos de las listas `L1` y `L2` situados en la misma posición (los dos primeros, los dos segundos, etc).
- Ambos están disponibles en SWI-Prolog bajo el nombre de `maplist` (`maplist/2` y `maplist/3`), a pesar de lo cual es interesante estudiar su implementación (ver a continuación).

Implementación de map/2 (maplist/2 en SWI-Prolog)

```
% map(+Obj, ?L)
% cierto si Obj se puede aplicar con éxito sobre
% TODOS los elementos de la lista L

map(_, []).

map(Obj, [C|R]) :-
        call(Obj, C),  % uso de call/2
        map(Obj, R).
```

El parámetro `Obj` anterior puede llevar o no argumentos propios, pero tiene que ser ejecutable una vez añadido, *en último lugar*, un argumento más (el elemento `C` que se va tomando de la lista `L`).

Ejemplo (Usos de map/2)

```
?- map(integer, [1,-4]).
true   % tras ejecutar integer(1) e integer(-4)

?- map(var, [1,X]).
false  % tras ejecutar var(1), que falla

?- map(>, [1,0]).   % ejecuta >(1): error
ERROR: procedure ` >(A)' does not exist

?- map(>(2), [1,0]).
true   % tras ejecutar >(2,1) y >(2,0)

?- map(>(2), [1,3]).
false. % tras ejecutar >(2,1) y >(2,3), que falla

?- map(>(2), [a,2]). % tras ejecutar >(2,a), error
ERROR: Arithmetic: `a/0' is not a function
```

Observación (Árboles de Resolución con `call`)

Los Árboles de Resolución en los que aparece el predicado `call/N` se construyen igual que los de los predicados convencionales, con la única diferencia de que, como se ha explicado antes, `call(+Obj, +Extra1, +Extra2, ...)` simplemente añade los valores `Extra1, Extra2, ...`, en el orden dado, detrás de los argumentos propios del objetivo `Obj`, si este los tuviera, y ejecuta el objetivo resultante (produciendo un error en caso de que no sea ejecutable).

El ejemplo a continuación incluye un Árbol de Resolución de este estilo, el correspondiente a la consulta `?- map(>(2), [1,3]).`

Ejemplo (Árbol de Resolución con `call`)

$$?- \underline{map(>(2),[1,3])}.$$

$$\Big| \{Obj/>(2), C/2, R/[3]\}$$

$$?- \underline{call(>(2),1)}, map(>(2),[3]).$$

$$\Big| \{\}$$

$$?- \underline{>(2,1)}, map(>(2),[3]).$$

$$\Big| \{\}$$

$$?- \underline{map(>(2),[3])}.$$

$$\Big| \{Obj4/>(2), C4/3, R4/[]\}$$

$$?- \underline{call(>(2),3)}, map(>(2),[\,]).$$

$$\Big| \{\}$$

$$?- \underline{>(2,3)}, map(>(2),[\,]).$$

$$\boxed{\text{fallo}}$$

Implementación de map/3 (maplist/3 en SWI-Prolog)

```
% map(+Obj, ?L1, ?L2)
% cierto si Obj se puede aplicar con éxito sobre
% TODAS las parejas de elementos de las listas
% L1 y L2 situados en la misma posición.

map(_,[], []).

map(Obj, [C1|R1], [C2|R2]) :-
        call(Obj, C1, C2),    % uso de call/3
        map(Obj, R1, R2).
```

El parámetro `Obj` anterior puede llevar o no argumentos propios, pero tiene que ser ejecutable una vez añadidos, *en último lugar*, dos argumentos más (los elementos `C1` y `C2` que se van tomando de las listas `L1` y `L2`, respectivamente).

Ejemplo (Usos de map/3, 1/4)

Suponga disponibles los siguientes predicados:

```
% cuad(+X, ?Y)
% cierto si Y es el cuadrado de X

    cuad(X,Y) :-
        Y is X*X.
    % X debe ser de entrada debido al "is"

% trad(?X, ?Y)
% cierto si Y es el nombre del dígito X

    trad(0, cero).
    trad(1, uno).
        ...
    trad(9, nueve).
```

Ejemplo (Usos de map/3, 2/4)

```
?- map(cuad, [2,3], [4,9]).
true  % ejecuta cuad(2,4) y cuad(3,9)

?- map(cuad, [2,3], [4,8]).
false % ejecuta cuad(2,4) y cuad(3,8), que falla

?- map(cuad, [2,3], [4,9,16]).
false % map falla con listas de tamaño distinto

?- map(cuad, [1,2], L).
L = [1,4]
% ejecuta cuad(1,E1), cuad(2,E2) siendo L=[E1,E2]
```

Ejemplo (Usos de map/3, 3/4)

```
?- map(cuad, L, [4,9]).
% ejecuta cuad(E1,4) siendo L=[E1,E2] pero da error:
% el primer argumento de cuad debe ser de entrada
ERROR: Arguments are not sufficiently instantiated

? map(integer, [1,2], L).
% intenta ejecutar integer(1,E1), siendo L=[E1,E2].
ERROR: procedure `integer(A,B)' does not exist

?- map(trad, [1,3], L).
% ejecuta trad(1,E1), trad(3,E2), siendo L=[E1,E2].
L = [uno,tres]

?- map(trad, L, [1,3]).
 % ejecuta trad(E1,1), que falla, siendo L=[E1,E2]
false.
```

Ejemplo (Usos de map/3, 4/4)

```
?- map(cuad, [1,2,3], L), map(trad, L, NL).
L = [1, 4, 9], NL = [uno, cuatro, nueve]

?- map(>, [3,2], [0,1]).
% ejecuta >(3,0) y >(2,1), ambos ciertos
true
```

Siendo `plus(?I1,?I1,?I3)` el predicado predefinido cierto si `I3=I2+I1` y al menos dos de sus argumentos tienen valor:

```
?- map(plus(1), [1,2], L).
% ejecuta plus(1,1,E1) y plus(1,2,E2), con L=[E1,E2]
L = [2, 3]  % suma 1 a todos los elementos de [1,2]

?- map(plus(1), L, [1,2]).
% ejecuta plus(1,E1,1) y plus(1,E2,2), con L=[E1,E2]
L = [0, 1] % resta 1 a todos los elementos de [1,2]
```

Observación (1 de 2)

Conviene destacar que a diferencia de lo que ocurre con los predicados de recolección, los predicados `map/maplist` (o cualquier otro predicado de orden superior, como los discutidos más adelante, involucrando el uso de `call/N` con `N>1`), NO aceptan objetivos negados ni compuestos, por lo que estos deben implementarse aparte. Por ejemplo, suponga que necesita averiguar si una lista contiene exclusivamente números enteros y constantes:

```
?- map((integer;atom), [1,2,a]).  % o maplist
ERROR procedure `;(A,B,C)' does not exist
```

La solución pasa por implementar aparte el objetivo adecuado:

```
intORatom(X) :- integer(X) ; atom(X).

?- map(intORatom, [1,2,a]).
true
```

Observación (2 de 2)

Suponga ahora que necesita averiguar si dos listas dadas tienen mismo tamaño, ambas contienen solo números naturales positivos y cada elemento de la primera es divisible por el correspondiente elemento de la segunda. Una posible solución:

```
positivo(X) :-                  |    divi(X,Y) :-
    integer(X), X > 0.          |      positivo(X),
                                |      positivo(Y),
                                |      X mod Y =:= 0.
```

```
?- map(divi, [4,2,8], [2,1,4]).
% ejecuta divi(4,2), divi(2,1) y divi(8,4)
true

?- map(divi, [4,4], [3,2]).
% ejecuta divi(4,3), que falla.
false
```

Operaciones de filtrado

- `filter(+Obj,+L,?NL)` (`include` en SWI-Prolog)

  cierto si `NL` es la lista conteniendo aquellos elementos de la lista de entrada `L` sobre los que se puede aplicar con éxito el objetivo `Obj` (`NL=[]` si no hay ninguno).

- `excluye(+Obj,+L,?NL)` (`exclude` en SWI-Prolog)

  cierto si `NL` es la lista resultante tras excluir de la lista de entrada `L` aquellos elementos sobre los que se puede aplicar con éxito el objetivo `Obj` (`NL=[]` si se excluyen todos).

En ambos casos el parámetro `Obj` puede llevar o no argumentos propios, pero tiene que ser ejecutable una vez añadido, *en último lugar*, un argumento más (el elemento `C` que se va tomando de la lista de entrada `L`).

Implementación de filter (include en SWI-Prolog)

```
% implementación recursiva
    filter(_, [], []).

    filter(Obj, [C|R], NL) :-
          call(Obj, C),              % uso de call/2
          !,
          NL = [C|NR],
          filter(Obj, R, NR).

    filter(Obj, [_|R], NL) :-
          filter(Obj, R, NL).

% implementación basada en "findall"
  filter_v2(Obj, L, NL) :-
     findall(X, (member(X,L), call(Obj,X)), NL).
```

Ejemplo (Usos de filter)

```
?- filter(var, [1, X, Hola, 2, adios], [X, Hola]).
% ejecuta var(1), ..., var(adios)
true % último arg. de entrada, para comprobación

?- filter(positivo, [1, X, Hola, -2, adios], NL).
% ejecuta positivo(1), ..., positivo(adios)
NL = [1]  % positivo: ver última observación

?- filter(>(3), [1,2,3], L).
% ejecuta >(3,1), ..., >(3,3)
L = [1, 2]

?- filter(>(3), [1,b,3], L).
% ejecuta >(3,1), >(3,b): error
ERROR: Arithmetic: `b/0' is not a function
```

Implementación de excluye (exclude en SWI-Prolog)

```
excluye(_, [], []).     % implementación recursiva
excluye(Obj, [C|R], NR) :-
   call(Obj, C),        % uso de call/2
   !,
   excluye(Obj, R, NR).
excluye(Obj, [C|R], [C|NR]) :-
    excluye(Obj, R, NR).

% implementación basada en "filter" y "findall"
excluye_v2(Obj, L, NL) :-
  filter(Obj, L, LF),  % include en SWI-Prolog
  findall(X, (member(X,L), \+ member(X,LF)), NL).

% implementación basada en "findall" y "\+ call"
 excluye_v3(Obj, L, NL) :-
  findall(X, (member(X,L), \+ call(Obj,X)), NL).
```

Ejemplo (Usos de excluye)

```
?- excluye(var, [1, X, Hola, 2, adios], L).
% ejecuta var(1), ..., var(adios)
L = [1, 2, adios]

?- excluye(>(3), [1,2,3], L).
% ejecuta >(3,1), ..., >(3,3)
L = [3]

?- excluye(>(3), [1,b,3], L).
% ejecuta >(3,1), >(3,b): error
ERROR: Arithmetic: `b/0' is not a function

?- excluye(par, [1,b,2], L).
% siendo par(X) :- integer(X), X>=0, X mod 2 =:= 0.
L = [1,b]
```

Observación

Note que las implementaciones recursivas discutidas más arriba para las operaciones de aplicación y filtrado presentan recursión de cola módulo cons, recursión en la cual lo único que queda por hacer después de la llamada recursiva es añadir un elemento en cabeza. Recuerde (ver tema de listas) que PROLOG es capaz de optimizar este tipo de recursión, por lo que se trata de implementaciones eficientes.

Ejercicios (Uso de predicados de orden superior clásicos)

- *Ejercicio nº 1, apartado 1.2, de la Práctica de* PROLOG *nº 4.*
- *Ejercicio nº 2, apartado 2.4, de la Práctica de* PROLOG *nº 4.*
- *Dibuje el Árbol de Resolución correspondiente a la consulta* `?- map(cuad, [2,3], L)` *con las implementaciones de* `map/3` *y* `cuad/2` *mencionadas* ▸ *previamente*.

Soluciones propuestas:

Las soluciones propuestas, comentadas, a los ejercicios de esta práctica están disponibles en Práctica de PROLOG nº 4 con soluciones.

$$? - \underline{map(cuadrado, [2,3], L)}.$$

$$\Big| \quad \{Obj/cuadrado, C1/2, R1/[3], L/[C2|R2]\}$$

$$? - \underline{call(cuadrado, 2, C2)}, map(cuadrado, [3], R2).$$

$$\Big| \quad \{\}$$

$$? - \underline{cuadrado(2, C2)}, map(cuadrado, [3], R2).$$

$$\Big| \quad \{X/2, Y/C2\}$$

$$? - \underline{C2 \text{ is } 2 * 2}, map(cuadrado, [3], R2).$$

$$\Big| \quad \{C2/4\}$$

$$? - \underline{map(cuadrado, [3], R2)}.$$

$$\Big| \; \{Obj_5/cuadrado, C1_5/3, R1_5/[], R2/[C2_5|R2_5]\}$$

$$? - \underline{call(cuadrado, 3, C2_5)}, map(cuadrado, [\,], R2_5).$$

$$\Big| \; \{\}$$

$$? - \underline{cuadrado(3, C2_5)}, map(cuadrado, [\,], R2_5).$$

$$\Big| \; \{X_7/3, Y_7/C2_5\}$$

$$? - \underline{C2_5 \text{ is } 3 * 3}, map(cuadrado, [\,], R2_5).$$

$$\Big| \; \{C2_5/9\}$$

$$? - \underline{map(cuadrado, [\,], R2_5)}.$$

$$\Big| \; \{R2_5/[\,]\}$$

$$?-$$

$$\boxed{\text{éxito}}$$

$$L = [C2|R2] = [4|R2] = [4|[C2_5|R2_5]] = [4|[9|R2_5]] = [4|[9|[\,]]] = [4, 9]$$

Operaciones de reducción o plegado

- Permiten obtener *un único valor* a partir de una o varias listas, combinando sucesivamente los elementos de la(s) lista(s) mediante una cierta operación, en una cierta dirección, y partiendo, opcionalmente, de un valor inicial dado.

- Existen distintas operaciones de este tipo, que se diferencian básicamente en cuántas listas combinan, si usan o no valor inicial y en cómo y en qué orden operan sus elementos para obtener el valor de salida.

- Existen también distintas terminologías para referirse a estas operaciones, no siempre consistentes entre sí.

Operaciones de reducción o plegado

`pliegai(+Obj, +L, +VIni, -VFin)`
`(plegado por la izquierda)`

Obtiene un valor final de salida, `VFin`, combinando de izquierda a derecha los elementos de la lista `L` mediante `Obj` y partiendo del valor inicial `VIni`:

- Si `L=[]`, `VFin = VIni`.
- Si `L=[C|R]`:
  1. Combina `VIni` con `C` mediante `Obj`, dando lugar a `V1`, es decir, ejectuta `Obj(VIni, C, V1)`.
  2. Pliega por la izquierda (recursivamente) el resto de la lista, `R`, con valor inicial el `V1` obtenido en el paso anterior.

Observación

- Desde un punto de vista iterativo, siendo `L=[x1,...,xn]`, el predicado `pliegai(+Obj, +L, +VIni, -VFin)` hace lo siguiente:

```
Obj(VIni,x1,V1)
Obj(V1,x2,V2)
...
Obj(V(n-1),xn,VFin).
```

- El parámetro `Obj` puede llevar o no argumentos propios, pero tiene que ser ejecutable una vez añadidos, *en último lugar y en el orden dado a continuación*, tres argumentos más: el valor calculado en el paso anterior (`VIni` la primera vez), el elemento de la lista que se combina con él y el valor de salida.

Implementaciones de pliegai y de foldl (SWI-Prolog)

```
% pliegai(+Obj, +L, +VIni, -VFin)

  pliegai(_, [], VIni, VIni).

  pliegai(Obj, [C|R], VIni, VFin) :-
        call(Obj, VIni, C, V1),      %% Aquí difieren
        pliegai(Obj, R, V1, VFin).

% foldl(+Obj, +L, +VIni, -VFin) SWI-Prolog

  foldl(_, [], VIni, VIni).

  foldl(Obj, [C|R], VIni, VFin) :-
        call(Obj, C, VIni, V1),     %% Aquí difieren
        foldl(Obj, R, V1, VFin).
```

Ejemplo (Usos de pliegai y de foldl, 1/2)

```
?- pliegai(plus, [1,2,3,4], 0, S).  % o foldl
S = 10  % suma los elementos de una lista

?- pliegai(prod, [1,2,3,4], 1, S).  % o foldl
S = 24 % multiplica los elementos de una lista

longitud(L, Long) :-
    pliegai(suma1, L, 0, Long).     % o foldl
suma1(Ac, _Elemento, NAc) :-
    NAc is Ac + 1.

?- longitud([a,b,c], X).
X = 3 % calcula la longitud de una lista
```

Ejemplo (Usos de pliegai y de foldl, 2/2)

```
?- pliegai(append, [[1,2],[3]], [], S).
S = [1, 2, 3] % concatena las listas de una lista

?- foldl(append, [[1,2],[3]], [], S).
S = [3,1,2] % concatena las listas en orden inverso
```

Los resultados difieren porque la operación `append`, a diferencia de la suma o el producto de los ejemplos anteriores, no es conmutativa.

Ejercicios (Uso de predicados de orden superior clásicos)

Utilice los predicados de orden superior sobre listas que considere oportunos para implementar los siguientes predicados:

1. `imprimelista(+L)`, *cuyo efecto es escribir los elementos de la lista* `L` *en el fichero de escritura actual, uno por línea.*
2. `todospares(+L)`, *cierto si* `L` *es una lista compuesta exclusivamente por números naturales pares.*
3. `doblalista(+L1, ?L2)`, *cierto si* `L1` *es una lista de números enteros y* `L2` *contiene los mismos números pero multiplicados por dos.*

Soluciones propuestas:

1. *imprimelista(L) :- maplist(imprimeelemento, L).*
   imprimeelemento(E) :- write(E), nl.

2. *todospares(L) :- maplist(espar, L).*
   espar(N) :- integer(N), N >= 0, N mod 2 =:= 0.

3. *doblalista(L1, L2) :- maplist(integer, L1), maplist(dobla, L1, L2).*
   *dobla(I1, I2) :- I2 is I1*2.*

Ejercicios (Uso de predicados de orden superior clásicos)

Utilice los predicados de orden superior sobre listas que considere oportunos para implementar los siguientes predicados:

1. `traducerestos(+L1, ?L2)`, *cierto si* `L1` *es una lista de números naturales y* `L2` *contiene los restos obtenidos al hacer la división entera de esos números por diez, escritos en castellano. Por ejemplo, la consulta* `?- traducerestos([1,13,20], X)` *debe devolver* `X=[uno, tres, cero].`
2. `opuestos(+L1, ?L2)`, *cierto si* `L1` *y* `L2` *son listas de números reales tales que, en cada posición,* `L2` *contiene el número opuesto de* `L1`. *Por ejemplo,* `?- opuestos([1,-2,3.5,-1.5],L)` *debe devolver* `L = [-1, 2, -3.5, 1.5].`

Soluciones propuestas:

1.
```
traducerestos(L1, L2) :-
maplist(procesa, L1, L2).

procesa(E, TE) :- integer(E), E >= 0, R is E mod 10, traduce(R, TE).

traduce(0, cero).
traduce(1, uno).
traduce(2, dos).
traduce(3, tres).
traduce(4, cuatro).
traduce(5, cinco).
traduce(6, seis).
traduce(7, siete).
traduce(8, ocho).
traduce(9, nueve).
```

2.
```
opuestos(L1, L2) :-
maplist(number, L1),
maplist(opuesto, L1, L2).

opuesto(X, Y) :- Y is -X.
```

Ejercicios (Uso de predicados de orden superior clásicos)

Utilice los predicados de orden superior sobre listas que considere oportunos para implementar los siguientes predicados:

1. `multiplosN(+L,+N,?LMN)`, *cierto si* `L` *es una lista de números naturales,* `N` *es un natural positivo y* `LMN` *es la lista de los elementos de* `L` *que son múltiplos de* `N`. *Por ejemplo, la consulta* `?- multiplosN([1,2,3,4,5,6],2,L)` *debe devolver* `L = [2,4,6].`
2. `separaMultiplosN(+L,+N,?LMN, ?R)`, *cierto si* `L` *es una lista de números naturales,* `N` *es un natural positivo,* `LMN` *es la lista de los elementos de* `L` *que son múltiplos de* `N` *y* `R` *es una lista con los que no lo son. Por ejemplo, la consulta* `?- separaMultiplosN([1,2,3,4,5,6],2,LSi, LNo)` *debe devolver* `LSi = [2,4,6]` *y* `LNo = [1,3,5].`

Soluciones propuestas:

1.
```
multiplosN(L, N, LMN) :-
maplist(natural, [N|L]),
N > 0,
include(multiplo(N), L, LMN).

multiplo(N, E):- E mod N =:= 0.
```

2.
```
separaMultiplosN(L, N, LMN, R) :-
multiplosN(L, N, LMN),
exclude(multiplo(N), L, R).
```

Ejercicios (Uso de predicados de orden superior clásicos)

Ejercicio nº 3 de la Práctica de PROLOG nº 4.

Las soluciones comentadas a los ejercicios de esta práctica están disponibles en Práctica de PROLOG nº 4 con soluciones.

IMPLEMENTACIÓN DE NUEVOS PREDICADOS DE ORDEN SUPERIOR

- Los predicados de orden superior básicos `call/N` no solo permiten implementar los predicados de orden superior clásicos (de aplicación, filtrado y plegado) estudiados más arriba, sino que sirven también para implementar fácilmente nuevos predicados de orden superior cuya necesidad pueda surgir en el desarrollo de aplicaciones.
- La técnica para implementar nuevos predicados de orden superior es la misma que la descrita más arriba para los clásicos: se implementan igual que cualquier otro predicado en PROLOG, con la diferencia de que algunos de sus argumentos pueden ser otros predicados, desconocidos en tiempo de compilación, que se invocan mediante `call` y que se concretan y ejecutan en el momento de usar el predicado.

Ejemplo

`cuantos(+Obj,+L,?N)`, cierto si `N` es el número de elementos de la lista `L` sobre los que se aplica con éxito el objetivo `Obj`. Por ejemplo,
`?- cuantos(integer,[1,X,3],C).` daría `C = 2` y
`?- cuantos(var,[1,X,3],C).` daría `C = 1`.

```
% mediante filtrado
cuantos_f(Obj, L, N) :-
  include(Obj, L, NL),
  length(NL, N).
```

```
% implementación recursiva
cuantos(_, [], 0).
cuantos(Obj, [C|R], N) :-
    call(Obj,C),
    !,
    cuantos(Obj, R, NR),
    N is NR + 1.
cuantos(Obj, [_|R], N) :-
    cuantos(Obj, R, N).
```

Ejemplo

`map_pares(+Obj,?L1,?L2)`, cierto si el objetivo `Obj` se puede aplicar con éxito sobre todas las parejas de elementos de las listas `L1` y `L2` situados en posiciones pares (los segundos, los cuartos, etc). Por ejemplo, `?- map_pares(cuadrado,[1,2,3,4], L).` daría `L = [1,4,3,16].`

```
map_pares(_, [], []).

map_pares(_, [C], [C]).

map_pares(Obj, [C1,C2|R], [C1,NC2|NR]) :-
                call(Obj, C2, NC2),
                map_pares(Obj, R, NR).
```

Ejercicios (Recolección, orden superior)

1. *Proponga una implementación con recursión de cola para el predicado* `cuantos(+Obj,+L,?N)`*, cierto si* `N` *es el número de elementos de la lista* `L` *sobre los que se aplica con éxito el objetivo* `Obj` *(implementado en el penúltimo ejemplo con recursión no final).*

2. *Estudie la resolución en* PROLOG *del problema del coloreado de mapas (aquí puede encontrar el código).*

3. *Haga los ejercicios 4 y 5 de la Práctica de* PROLOG *nº 4. Las soluciones comentadas a los ejercicios de esta práctica están disponibles en Práctica de* PROLOG *nº 4 con soluciones.*

Soluciones propuestas:

```
% cuantos(+Obj,+L,?N)
% cierto si N es el número de elementos de la lista L
% sobre los que se aplica con éxito el objetivo Obj
% Implementación con recursión de cola
```

```
% añadir parámetro de acumulación con valor caso base
cuantosrc(Obj, L, N) :- cuantosrc(Obj, L, 0, N).


% caso base: se devuelve lo acumulado
cuantosrc(_, [], Ac, Ac).


% caso recursivo cuando C cumple Obj: se suma 1 al acumulador
cuantosrc(Obj, [C|R], Ac, N) :-
call(Obj, C),
!,
NAc is Ac + 1,
cuantosrc(Obj, R, NAc, N).


% caso recursivo cuando C NO cumple Obj: se mantiene el acumulador
cuantosrc(Obj, [_|R], Ac, N) :- cuantosrc(Obj, R, Ac, N).
```

BIBLIOGRAFÍA

- L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.
- W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fifth edition, 2003.
- I. Bratko.*Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 2001.
- J. Lloyd. *Foundations of Logic Programming*, (Second Edition). Springer-Verlag, 1987.
- R. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
- U. Nilsson and J. Maluszynski. Logic, Programming and Prolog. John Wiley & Sons Ltd, 1996.
- SWI-Prolog, entorno de programación en Prolog de dominio público.
- comp.lang.prolog. Faq