Programación Declarativa

Pascual Julián Iranzo

Dep. de Informática. Univ. de Castilla-La Mancha.
Paseo de la Universidad, 4. 13071 Ciudad Real, España.
email: pjulian@inf-cr.uclm.es

Primera versión:	3-Septiembre-2000
Primera revisión:	2-Enero-2002
Segunda revisión:	24-Octubre-2007

Capítulo 1

Introducción.

- En este capítulo estudiaremos las características más notables de los lenguajes de programación convencionales y de los lenguajes de programación declarativos.
- **Objetivos**:

  1. Entender qué son los lenguajes declarativos y qué aportan de nuevo.

  2. Establecer su ámbito de aplicación.

1.1. Computadores y Lenguajes de Programación Convencionales.

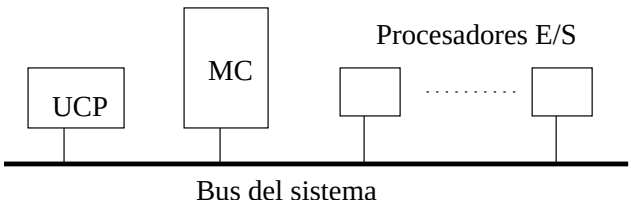
- Los lenguajes convencionales son una abstracción de alto nivel del tipo de máquina para el que se han desarrollado.

- Muchos de sus defectos e imperfecciones provienen de esta estrecha relación con las máquinas que los sustentan.

1.1.1. Organización de los computadores.

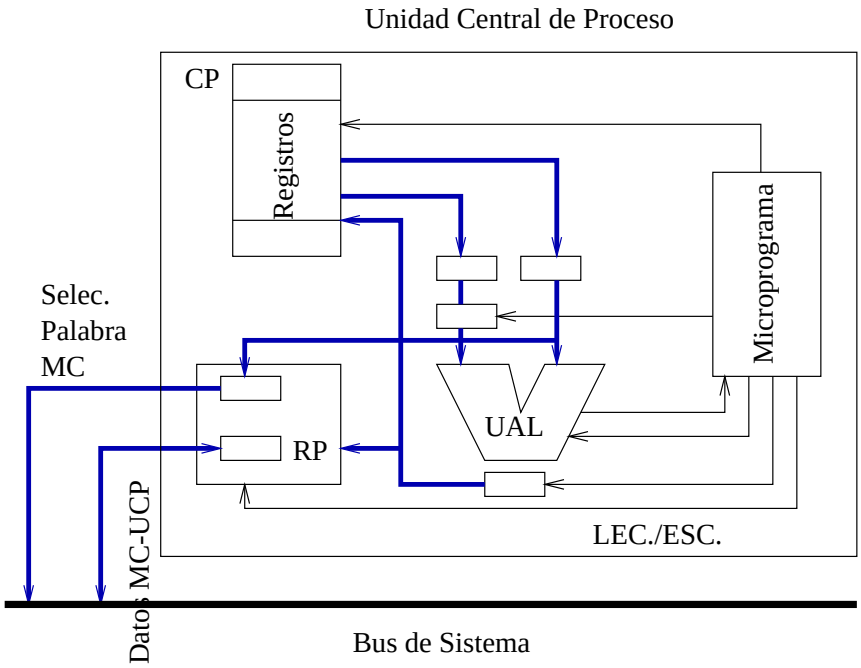
Los computadores que siguen el modelo de organización de von Neumann están constituidos por los siguientes componentes internos:

1. Unidad central de proceso (UCP)
2. Memoria central (MC)
3. Procesadores de dispositivos periféricos
4. El bus del sistema.



Computadores y Lenguajes de Prog. Convencionales.

- **La UCP** está dividida en cuatro componentes funcionales, que están unidos por buses internos y líneas de control:
  - Los registros
  - La unidad de control (UC): rige el comportamiento del computador mediante la interpretación y secuenciamiento de las instrucciones que lee de la MC.
  - La unidad aritmético lógica (UAL): realiza un número muy limitado de operaciones (lógicas, como AND, OR, y NOT; aritméticas como ADD; y desplazamientos).
  - Un reloj
- Existe una correspondencia entre los dos componentes principales de la UCP, la UC y la UAL, y el tipo de informaciones que almacena la MC, instrucciones y datos.
  - La UC trata las instrucciones
  - la ALU los datos.



- **La MC** puede considerarse como un conjunto de celdas (llamadas palabras), cada una de ellas con la capacidad de almacenar una información: dato o instrucción.
  - La propiedad esencial de la memoria que nos interesa es su capacidad de ser direccionada.
  - Las celdas pueden considerarse numeradas y la UC conoce cada celda por su número, llamado dirección.

- **El bus del sistema** es un conjunto de conductores paralelos a través de los cuales transcurren una serie de señales eléctricas que se intercambian entre los componentes del sistema.
  - Ventaja: la organización en torno a un bus único es la menos costosa de las organizaciones.
  - Desventaja: "cuello de botella de von Neumann".
  - Desventaja: gran parte del tráfico no es útil para el cómputo (son direcciones de donde se encuentran las instrucciones y los datos a procesar).

Computadores y Lenguajes de Prog. Convencionales.

El modelo de organización y funcionamiento de los computadores que acabamos de describir conduce a un modelo de cómputo en el que:

1. Los procesadores ejecutan las intrucciones de un modo puramente secuencial (salvo rupturas de secuencia)

   $\Longrightarrow$

   los lenguajes convencionales son dificiles de paralelizar.

2. se realiza una completa separación entre el tipo de informaciones que almacena la MC, dividiendolas en instrucciones y datos

   $\Longrightarrow$

   repercusión sobre el diseño de los lenguajes convencionales.

3. se introduce el concepto de estado de la computación (debido al uso de registros para realizar las operaciones de la UCP y la visión de la MC como un conjunto de palabras).

1.1.2. Características de los lenguajes convencionales

- Los lenguajes convencionales (o imperativos) están inspirados en la arquitectura de von Neumann.

- los distintos recursos expresivos proporcionados por tales lenguajes pueden verse como abstracciones de los componentes de la máquina de von Neumann o de sus operaciones elementales:

  1. variables $\Longleftrightarrow$ celdas de la MC / registros;

  2. registro (o estructura) / arrray $\Longleftrightarrow$ conjunto contiguo de celdas de la MC;

  3. instrucciones de control $\Longleftrightarrow$ intrucciones de salto condicional o incondicional del lenguaje máquina;

  4. instrucción de asignación $\Longleftrightarrow$ intrucciones LOAD+STORE o MOVE del lenguaje máquina.

- La instrucción de asignación resulta ser representativa del cuello de botella de von Neumann y nos obliga a pensar en términos de trasiego de información entre celdas de memoria.

- la instrucción de asignación separa la programación en dos mundos: Expresiones / Instrucciones.

  direcciones $=$ expresión.

Computadores y Lenguajes de Prog. Convencionales.

- Podemos distinguir dos aspectos fundamentales en las tareas de programación:
  - Aspectos *lógicos*: Esto es, *¿Qué debe computarse?*). Esta es la cuestión esencial.
  - Aspectos de *control*:
    - Organización de la secuencia de cómputos en pequeños pasos.
    - Gestión de la memoria durante la computación.

- Kowalski y otros son partidarios de la independencia de los aspectos lógicos y de control.

- Atendiendo a este criterio de independencia podemos elevar algunas críticas sobre el uso de los lenguajes imperativos:
  1. La presencia de instrucciones de control de flujo y las operaciones de gestión de memoria obscurecen el contenido lógico del programa.
  2. La operación de asignación utiliza las variables de modo matemáticamente impuro:

     `x = x+1`.

     Permite la introducción de efectos laterales.
  3. Dejar las cuestiones de control al programador no parece lo más indicado.

Ejemplo 1 *Concatenación de dos listas.*

```
#include <stdio.h>
#include <stdlib.h>
typedef struct nodo
{
 char dato;
 struct nodo *enlace;
 } LISTA;
void mostrar(LISTA *ptr);
void insertar(LISTA **ptr, char elemento);
LISTA *crear_lista();
LISTA *concatenar(LISTA *ptr1, LISTA *ptr2);
void main()
{
 LISTA *l1, *l2, *lis = NULL;
 l1 = crear_lista();
 l2 = crear_lista();
 lis = concatenar(l1, l2);
 printf("\n La nueva lista enlazada es: ");
 mostrar(lis);
}
void mostrar(LISTA *ptr)
{
 while(ptr != NULL)
 {
  printf("%c",ptr->dato);
  ptr = ptr->enlace;
 }
 printf("\n");
}
void insertar(LISTA **ptr, char elemento)
{
 LISTA *p1, *p2;
 p1 = *ptr;
 if(p1 == NULL)
```

```
 {
  p1 = malloc(sizeof(LISTA));
  if (p1 != NULL)
  {
   p1->dato = elemento;
   p1->enlace = NULL;
   *ptr = p1;
  }
 }
 else
 {
  while(p1->enlace != NULL) p1 = p1->enlace;
  p2 = malloc(sizeof(LISTA));
  if(p2 != NULL)
  {
   p2->dato = elemento;
   p2->enlace = NULL;
   p1->enlace = p2;
  }
 }
}
LISTA *crear_lista()
{
 LISTA *lis = NULL;
 char elemento;
 printf("\n Introduzca elementos: ");
 do
 {
  elemento = getchar();
  if(elemento != '\n') insertar(&lis, elemento);
 } while(elemento != '\n');
 return lis;
}LISTA *concatenar(LISTA *ptr1, LISTA *ptr2)
{ LISTA *p1;
 p1 = ptr1;
 while(p1->enlace != NULL) p1 = p1->enlace;
 p1->enlace = ptr2;
 return ptr1;
}
```

Computadores y Lenguajes de Prog. Convencionales.

Este programa ilustra y concreta algunas de las propiedades de los lenguajes convencionales ya mencionadas junto a otras nuevas:

1. Es una secuencia de instrucciones que son **ordenes a la máquina** que operan sobre un estado no explícitamente declarado (*lenguajes imperativos*).

2. Para entender el programa debemos ejecutarlo mentalmente siguiendo el modelo de computación de von Neumann, estudiando como cambian los contenidos de las variables y otras estructuras en la MC.

3. Es necesario gestionar explícitamen la MC: llamada al sistema `malloc()` y empleo de variables de tipo puntero.

4. El programa acusa una falta de generalidad debido a la rigidez del lenguaje a la hora de definir nuevos tipos de datos (sólo concatena listas de caracteres).

5. La lógica y el control están mezclados, lo que dificulta la verificación formal del programa.

Computadores y Lenguajes de Prog. Convencionales.

Hemos puesto de manifiesto los puntos débiles, sin embargo reunen otras muchas ventajas:

1. eficiencia en la ejecución;

2. modularidad;

3. herramientas para la compilación separada;

4. herramientas para la depuración de errores;

5. son los lenguajes preferidos en amplias áreas de aplicación:
   a) computación númerica,
   b) gestión y tratamiento de la información,
   c) programación de sistemas

1.2. Programación Declarativa.

- La *programación declarativa* es un estilo de programación en el que el programador especifica *qué* debe computarse más bien que *cómo* deben realizarse los cómputos.

- "*programa = lógica + control*" (Kowalski)
  ("*algoritmos + estructuras de datos = programas*" (Wirth))

- El componente lógico determina el significado del programa mientras que el componente de control solamente afecta a su eficiencia.

- la tarea de programar consiste en centrar la atención en la *lógica* dejando de lado el *control*, que se asume automático, al sistema.

- La característica fundamental de la programación declarativa es el uso de la lógica como lenguaje de programación:
  - Un *programa* es una teoría formal en una cierta lógica, esto es, un conjunto de fórmulas lógicas que resultan ser la especificación del problema que se pretende resolver, y

  - la *computación* se entiende como una forma de inferencia o deducción en dicha lógica.

Programación Declarativa.

- Los principales requisitos que debe cumplir la lógica empleada son:

  1. disponer de un lenguaje que sea suficientemente expresivo;

  2. disponer de una semántica operacional (un mecanismo de cómputo que permita ejecutar los programas);

  3. disponer de una semántica declarativa que permita dar un significado a los programas de forma independiente a su posible ejecución;

  4. resultados de corrección y completitud.

- Según la clase de lógica que empleemos como fundamento del lenguaje declarativo obtenemos los diferentes estilos de programación declarativa.

Clase de lógica	**Estilo**
Ecuacional	Funcional
Clausal	Relacional
Heterogenea	Tipos
Géneros ordenados	Herencia
Modal	S.B.C.
Temporal	Concurrencia

1.2.1. Programación Lógica.

La *programación lógica* se basa en (fragmentos de) la lógica de predicados: lógica de *cláusulas de Horn* (HCL).

Ejemplo 2 *Consideremos, de nuevo, el problema de la concatenación de dos listas.*

$$\begin{array}{lcl} app([\,], X, X) & \leftarrow & \\ app([X|X_s], Y, [X|Z_s]) & \leftarrow & app(X_s, Y, Z_s) \end{array}$$

- Lectura declarativa:
  - la concatenación de la lista vacía [ ] y otra lista X es la propia lista X.
  - La concatenación de dos listas $[X|X_s]$ e Y es la lista que resulta de añadir el primer elemento X de la lista $[X|X_s]$ a la lista Z_s, que se obtiene al concatenar el resto X_s de la primera lista a la segunda Y.
- Lectura operacional:
  - Para concatenar dos listas $[X|X_s]$ e Y primero es preciso resolver el problema de concatenar el resto X_s de la primera lista, a la segunda Y.
  - La concatenación de la lista vacía [ ] y otra lista X es un problema ya resuelto.

Programación Declarativa.

Hechos que advertimos en el Ejemplo 2:

- No hay ninguna referencia explícita al tipo de representación en memoria de la estructura de datos lista.

  lenguaje declarativo $\Longrightarrow$ *gestión automática de la memoria*

- Mecanismo de cómputo que permite una *búsqueda indeterminista* (*built-in search*) de soluciones $\Longrightarrow$

  1. El programa puede responder a diferentes cuestiones (*objetivos*) sin necesidad de efectuar ningún cambio en el programa,
  2. Permite computar con *datos parcialmente definidos*,
  3. La *relación de entrada/salida* no está fijada de antemano.

El programa del Ejemplo 2 sirve para responder a las preguntas:

- "¿El resultado de concatenar las listas $[2, 4, 6]$ y $[1, 3, 5, 7]$ es la lista $[2, 4, 6, 1, 3, 5, 7]$?"

  $\leftarrow app([2, 4, 6], [1, 3, 5, 7], [2, 4, 6, 1, 3, 5, 7])$

  Respuesta: *verdadero*.

- "¿Cual es el resultado de concatenar las listas $[2, 4, 6]$ y $[1, 3, 5, 7]$?"

  $\leftarrow app([2, 4, 6], [1, 3, 5, 7], Z)$

  Respuesta: $\{Z = [2, 4, 6, 1, 3, 5, 7]\}$ (Z se utiliza con un sentido puramente matemático).

- "¿Qué listas dan como resultado de su concatenación la lista $[2, 4, 6, 1, 3, 5, 7]$?"

  $\leftarrow app(X, Y, [2, 4, 6, 1, 3, 5, 7])$

  Respuestas:

  $\{X = [\,], Y = [2, 4, 6, 1, 3, 5, 7]\}$;
  $\{X = [2], Y = [4, 6, 1, 3, 5, 7]\}$;
  $\{X = [2, 4], Y = [6, 1, 3, 5, 7]\}$;
  $\ldots$

1.2.2. Programación Funcional.

- Los *lenguajes funcionales* se basan en el concepto de *función* (matemática) y su definición mediante ecuaciones (generalmente recursivas), que constituyen el programa.
- La programación funcional se centra en la evaluación de expresiones (funcionales) para obtener un *resultado*.

Ejemplo 3 *Concatenación de dos listas.*

$$
\begin{array}{lll}
data\ [t] & = & [\,] \mid [t : [t]] \\
app & :: & [t] \rightarrow [t] \rightarrow [t]
\end{array}
$$

$$
\begin{array}{lll}
app\ [\,]\ x & = & x \\
app\ (x : x_s)\ y & = & x : (app\ x_s\ y)
\end{array}
$$

Algunas características de la solución anterior:

1. Necesidad de fijar el perfil de la función (dominio y el rango de la función)
   $\Longrightarrow$ idea de *tipo de datos*.
   Los lenguajes funcionales modernos son lenguajes *fuertemente basados en tipos* (*strongly typed*)

2. Se ha declarado la estructura de datos lista
   - "[ ]" y ":" son los *constructores del tipo*,
   - t es una *variable de tipo*
     $\Longrightarrow$ podemos formar listas de diferentes tipos de datos (*Polimorfismo*).

Programación Declarativa.

Otras características de las funciones (matemáticas):

- *Transparencia referencial*: el resultado (*salida*) de aplicar una función sobre sus argumentos viene determinado exclusivamente por el valor de éstos (su *entrada*).
  $\Longrightarrow$
  - NO *efectos laterales* (*side effects*);
  - Permite el estilo de la programación funcional basado en el razonamiento ecuacional (la substitución de iguales por iguales);
  - Cómputos deterministas.

- Capacidad para ser compuestas:
  - La composición de funciones es la técnica por excelencia de la programación funcional;
  - Permite la construcción de programas mediante el empleo de funciones primitivas o previamente definidas por el usuario;
  - La composición de funciones refuerza la modularidad de los programas.

Ejemplo 4 *Función que invierte el orden de los elementos en una lista.*

$$
\begin{array}{lll}
rev & :: & [t] \rightarrow [t] \\
 & & \\
rev\ [\,] & = & [\,] \\
rev\ (x : x_s) & = & app\ (rev\ x_s)\ [x]
\end{array}
$$

Orden superior:

- Una característica de los lenguajes funcionales que va más alla de la mera composición de funciones.

- El empleo de las funciones como "ciudadados de primera clase" dentro del lenguaje, de forma que las funciones:
  1. puedan almacenarse en estructuras de datos;
  2. pasarse como argumento a otras funciones;
  3. devolverse como resultados.

Ejemplo 5 *La función map, que toma como argumento una función f y una lista, y forma la lista que resulta de aplicar f a cada uno de los elementos de la lista.*

$$\begin{array}{lll} map & :: & (t_1 \to t_2) \to [t_1] \to [t_2] \\ \\ map\ f\ [\,] & = & [\,] \\ map\ f\ (x : x_s) & = & (f\ x) : (map\ f\ x_s) \end{array}$$

1.3. Comparación con los Lenguajes Convencionales y Areas de Aplicación.

Realizaremos la comparación estudiando una serie de atributos y criterios que se han utilizado para medir la calidad de los lenguajes de programación:

1. `Diseño del lenguaje y escritura de programas`.

2. `Verificación de programas`.

3. `Mantenimiento`.

4. `Coste y eficiencia`.

Comp. con los Leng. Conv. y Areas de Aplicación.

- **Diseño del lenguaje y escritura de programas**.

1. Sintaxis sencilla.
   El programador desea que la sintaxis de un lenguaje de programación sea sencilla, ya que ésta facilita el aprendizaje y la escritura de aplicaciones.

2. Modularidad y compilación separada.
   Estas características facilitan la estructuración de los programas, la división del trabajo y la depuración de los programas durante la fase de desarrollo.

3. Mecanismos de reutilización del software.
   Para eliminar la repetición de trabajo, un lenguaje debe de suministrar mecanismos para reutilizar el sotware.

   Este es un punto ligado al anterior.

Comp. con los Leng. Conv. y Areas de Aplicación.

4. Facilidades de soporte al método de análisis.

   - Los lenguajes de programación declarativa son lenguajes de especificación
     $\Longrightarrow$
     muy adecuados en las fases de análisis y rápido prototipado.
   - Los lenguajes convencionales requieren del uso de algún tipo de pseudocódigo + técnica de diseño para la confección de un algoritmo, que después es traducido al lenguaje elegido en la fase de implementación. Necesidad de herramientas de ayuda al diseño.

5. Entornos de programación.
   Un buen entorno de programación debe ofrecer editores especiales que faciliten la escritura de programas, depuradores, y en general utilidades de ayuda para la programación que permitan modificar y mantener grandes aplicaciones con multiples versiones.

   Los lenguajes declarativos tienen aquí uno de sus puntos débiles.

Comp. con los Leng. Conv. y Areas de Aplicación.

- **Verificación de programas**.
  La fiabilidad es un criterio central en la medida de la calidad del diseño de un lenguaje. Hay diversas técnicas para la verificación:

  1. Verificación de la corrección.
     Comprobar si el programa se comporta de acuerdo a su significado esperado, i.e. a su semántica.

  2. Terminación.
     Un programa debe terminar bajo cualquier circunstancia concebible.

  3. Depuración de programas.

     - El uso de baterías de pruebas no asegura un programa libre de errores.

     - Si se desea tener un alto grado de certeza, la batería de pruebas debe de ser significativa $\Longrightarrow$
       se alarga el tiempo de desarrollo del programa.

     - Para ayudar en la tarea de la depuración el lenguaje debe suministrar:
       - detección de errores en tiempo de compilación;
       - depuración de errores en tiempo de ejecución;
       - una sintaxis que sea legible.

Comp. con los Leng. Conv. y Areas de Aplicación.

- **Mantenimiento**.
  El mantenimiento incluye la reparación de los errores descubiertos después de que el programa se ha puesto en uso y la implementación de cambios necesarios para satisfacer nuevas necesidades. Para realizar estas tareas, el lenguaje debería poseer:

  1. Facilidad de uso y lectura.

     - Para que no sea muy complicado llegar a comprender el texto del programa y encontrar los errores.

     - También, si hay que extender el programa, un programador que no conozca la aplicación (o incluso el lenguaje) se beneficiará de estas propiedades.

  2. Modularidad y compilación separada.

     - Para que los errores estén localizados y simplificar las modificaciones.

     - Nuevamente, la posibilidad de realizar compilación separada ahorra gran cantidad de tiempo de desarrollo.

Comp. con los Leng. Conv. y Areas de Aplicación.

- **Coste y eficiencia**.
  Existen muchas factores para los que se puede aplicar un criterio de coste, sin embargo nos centraremos en dos de ellos:

  1. Coste de ejecución.
     El coste de ejecución es de primaria importancia en aplicaciones que se ejecutan repetidamente.
     - Uno de los puntos débiles de los lenguajes declarativos es que suelen ser ineficientes.

     - **Causa**: La dificultad de implementar las operaciones de unificación y emparejamiento y los mecanismos de búsqueda de soluciones en máquinas de arquitecturas convencionales.

  2. Coste de desarrollo.
     Comprende los costes asociados a las fases de análisis, programación y verificación de un programa. Es un criterio importante, ya que el coste de la hora de programador es muy alto frente al de la hora de UCP.

     Los costes de desarrollo son muy bajos cuando se utilizan lenguajes declarativos: la relación entre el número de líneas escritas en Prolog y el de escritas en C o Pascal es 1/10 en aplicaciones que desarrollan sistemas expertos y de 1/8 para aplicaciones que desarrollan compiladores.

Comp. con los Leng. Conv. y Areas de Aplicación.

La discusión anterior revela que un lenguaje de programación nunca es bueno para todas las tareas. Cada lenguaje tiene su dominio de aplicación.

Aplicación	**Principales lenguajes**	**Otros lenguajes**
Gestión	COBOL, SQL hojas de cálculo	C, 4GLs
Científica	FORTRAN, C, C++	BASIC, Pascal
Sistemas	C, C++	Pascal, Ada, BASIC, Modula
IA	Lisp, Prolog	
Edición	TeX, Postscript, procesadores de texto	
Procesos	Shells de UNIX, TCL, PERL	Marvel
Nuevos paradigmas	ML, Haskell, Smalltalk	Eiffel, Curry λ-Prolog

Comp. con los Leng. Conv. y Areas de Aplicación.

A pesar de ser un área de trabajo relativamente nueva la programación declarativa ha encontrado una gran variedad de aplicaciones:

- Representación del Conocimiento, Resolución de Problemas, Desarrollo de Sistemas de Producción, Sistemas Expertos y Procesamiento del lenguaje natural.

- Metaprogramación.

- Prototipado de aplicaciones, Bases de Datos Deductivas, Servidores y buceadores de información inteligentes.

- Quimica y biología molecular.

- Diseño de sistemas VLSI.

Más generalmente, la programación declarativa se ha aplicado en todos los campos de la computación simbólica (*lenguajes de computación simbólica*).

Capítulo 2

Preliminares y Fundamentos.

El objetivo de este capítulo es poner de manifiesto las relaciones existentes entre la lógica y los lenguajes de programación cuando se les considera como sistemas formales.

2.1. Sistema Formal.

- Según Carnap, el estudio de un sistema abarca tanto la componente sintáctica como la semántica, así como las relaciones entre ambas.

- Con esta perspectiva la definición un sistema formal sería algo más que la especificación de *lenguaje formal* y un *cálculo deductivo*.

- Esta visión de sistema formal es muy adecuada para el estudio de los lenguajes de programación.

Sistema Formal.

Definición 2.1.1 (Sistema formal, S) *Es un modelo de razonamiento matemático.Componentes:*

1. Sintaxis.

a*)* Lenguaje Formal*: Conjunto de símbolos y fórmulas sintácticamente correctas. Se necesita:*

- *Un vocabulario o alfabeto.*
- *Reglas que establezcan qué cadenas de signos son* fórmulas bien formadas *(fbf).*
- *Un conjunto de las definiciones utilizadas.*

b*)* Cálculo Deductivo*: Conjunto de reglas que permiten obtener fbf nuevas. Está constituido por:*

- *Un conjunto (posiblemente vacío) de axiomas.*
- *Un conjunto finito de reglas de inferencia.*
- *Un conjunto de reglas de construcción de una deducción en S.*

2. Semántica.

Estudia la adscripción de significado a los símbolos y fórmulas de los lenguajes formales; Se define el concepto de interpretación *y una noción de* verdad *en una interpretación.*

Sistema Formal.

Algunas de las propiedades de los sistemas formales:

- `Consistencia`.
  Un Sistema Formal es consistente cuando de él no se desprenden contradicciones, esto es, es imposible demostrar una fórmula y la negación de esa fórmula.

- `Corrección`.
  Un Sistema Formal es correcto si toda fórmula demostrable sin premisas, es válida: `Todo lo demostrable es cierto`.

- `Completitud`.
  Un sistema formal es completo si toda fórmula que es válida, segn la noción de verdad, es demostrable si premisas en el sistema: `Todo lo que es cierto se puede demostar`.

- `Decidibilidad`.
  Un sistema formal es decidible si existe un procedimiento finito para comprobar si una fórmula es o no válida.

2.2. Lógica de Predicados.

- El conocimiento de la lógica de predicados se considera un presupuesto para este curso.

- Introducimos ligeros cambios de notación:
  - emplemos letras mayúsculas para los símbolos de variable y minúsculas para símbolos de función y relación.

  - se emplean los símbolos "$\forall$" y "$\exists$" en lugar de símbolos "$\bigwedge$" y "$\bigvee$".

- La lógica de predicados es susceptible de caracterizarse como un sistema formal con las siguientes propiedades:
  - Consistente.

  - Correcta y completa.

  - Indecidible.
    - Sin embargo, si la fórmula realmente es válida, hay procedimientos de prueba que pueden verificarlo.

    - Estos procedimientos, en general, no terminan si la fórmula no es válida
      $\Longrightarrow$ la lógica de predicados es *semidecidible*.
      (e.g., Procedimiento de Herbrand — 1930).

2.3. Semántica, Sistemas Formales y Lenguajes de Programación.

- Un lenguaje de programación puede considerarse que es un lenguaje formal.

- Desde este punto de vista su definición consta de dos partes:

  1. La definición sintáctica.
     Se ocupa de fijar las reglas que describen las estructuras sintácticas (correctas) que constituyen los programas: las instruccciones, las declaraciones, las expresiones aritméticas, etc.

  2. la definición semántica.
     Se ocupa de asignar un significado a cada una de esas construcciones sintácticas del lenguaje.

- La teoría de autómatas y lenguajes formales constituye la base para la definición sintáctica de los lenguajes de programación.

- No existe una clase de definición semántica universalmente aceptada.

2.3.1. Necesidad de una definición semántica formal.

La definición sintáctica de un lenguaje de programación mediante una gramática formal (e.g., la notación BNF — *Bakus-Naur Form*), si bien facilita su análisis sintáctico, no es suficiente para la comprensión de sus estructuras sintácticas.

Ejemplo 6 *Definición sintáctica de programa para el lenguaje Pascal:*

$\langle Program \rangle$ ::= $\langle Program\_head \rangle \langle Block \rangle$‘**.**’
$\langle Program\_head \rangle$ ::= ‘**program**’ $\langle Identifier \rangle$[‘**(**’$\langle Identifier\_list \rangle$‘**)**’]‘;’
$\langle Identifier\_list \rangle$::= $\langle Identifier \rangle${‘**,**’$\langle Identifier \rangle$}

Estas reglas nos permiten escribir con corrección la cabecera de un programa, pero cabe preguntarse:

- ¿Los identificadores a qué hacen referencia?,

- ¿son nombres de ficheros de entrada/salida o qué son?.

Un desconocedor del lenguaje Pascal difícilmente responderá a esta pregunta con la sóla información suministrada por la descripción sintáctica.

Semántica informal.

- Se define la sintaxis de una construccción y se pasa a dar el significado de la misma indicando sus características de funcionamiento mediante el empleo de un lenguaje natural (español, inglés, etc.).

- La definición semántica mediante el lenguaje natural encierra ambigüedades y no suele ser lo suficientemente precisa como para captar todos los matices del significado de una construcción.

- Puede conducir a errores de programación y a, lo que es aún peor, errores de implementación cuando se utiliza como base para descripción del lenguaje.

Ejemplo 7 *En el libro de Kernighan y Ritchie, que describe el lenguaje C, puede leerse respecto a las expresiones de asignación:*

> *Existen varios operadores de asignación; todos se agrupan de derecha a izquierda.*
>
> expresión-asignación:
>
> expresión-condicional
> expresión-unaria operador-asignación expresión-asignación
>
> operador-asignación:
>
> `= *= /= %= += ...`
>
> *Todos requieren de un valor–l como operando izquierdo y éste debe ser modificable ... El tipo de una expresión de asignación es el de su operando izquierdo, y el valor es el almacenado en el operando izquierdo después de que ha tenido lugar la asignación.*
>
> *En la asignación simple con* `=`*, el valor de la expresión reemplaza al del objeto al que se hace referencia con el valor–l. Uno de los siguientes hechos debe ser verdadero: ambos operandos tienen tipo aritmético, en tal caso, el operando de la derecha es convertido al tipo del operando de la izquierda por la asignación; o ...*

Semántica, Sist. Formales y Leng. de Programación.

- ¿Qué quieren decir los autores cuando se refieren a que "Todos requieren de un valor–l como operando izquierdo"?

- ¿basta con esta descripción para contestar a la pregunta de en qué contextos podría emplearse una asignación?

- Supongamos que en un programa aparece la instrucción: `x = 2,45 + 3,67`.
  - La simple definición sintáctica de la instrucción de asignación no nos indica si la variable `x` ha sido declarada o, si lo fue, si se hizo con tipo real.

  - no podríamos decir a simple vista que resultado se obtendría al evaluar esa expresión.

    `x` toma el valor 6, si `x` se refiere a enteros;
    `x` toma el valor 6,12, si `x` se refiere a reales;

  - Esto muestra que es necesaria información sobre el entorno en el que se está utilizando una instrucción para comprender todo su significado.

Semántica por traducción.

1. Consiste en definir el significado de un lenguaje por medio de la traducción de sus construcciones sintácticas a un lenguaje más conocido.

2. Si disponemos de una máquina `M` cuyo conjunto de instrucciones y funcionamiento es conocido, puede hacerse empleando una técnica de *interpretación* o de *compilación*.

3. Un *compilador* que transforme programas escritos en nuestro lenguaje de programación (*lenguaje fuente*) en programas del lenguaje de la máquina `M` (*lenguaje objeto*) sería una definición semántica del lenguaje.

4. El significado del programa (fuente) es el programa obtenido tras la compilación.

5. Problemas:

   a) Un grado de *complejidad* que no está presente en el propio lenguaje. Esto conduce a una *falta de abstracción* en la descripción semántica que se pierde en los detalles.
   $\Longrightarrow$
   un programador puede verse sometido a un nivel de detalle innecesario.

   b) La *falta de portabilidad*. En este caso la definición semántica depende de la máquina `M` escogida.

   c) Una de las aplicaciones de las definiciones semánticas es servir de ayuda a la construcción de traductores de lenguajes
   $\Longrightarrow$ subvertir uno de sus objetivos.

La discusión precedente sugiere la necesidad de dotar a los lenguajes de programación de una *semántica formal*.

Semántica, Sist. Formales y Leng. de Programación.

Semántica formal.

- `Objetivo`: Asegurar que las definiciones de las construcciones del lenguaje sean claras y evitar ambigüedades en su interpretación.

- Las semánticas formales se definen mediante:
  - Una descripción de la clase de objetos que pueden ser manipulados por los programas del lenguaje (universo de discurso):
    - estados de una máquina (abstracta),
    - funciones, valores, formas normales,
    - asertos,

    u otra clase de objetos que deberán ser especificados formalmente.

  - Reglas que describen las formas en que las expresiones del lenguaje pueden combinarse para dar la salida.

- Son un modelo matemático que nos permite comprender y razonar sobre el comportamiento de los programas, facilitando:
  - el análisis y verificación de programas.

  - criterios para comprobar la equivalencia entre programas.

2.3.2. Semánticas formales de los Lenguajes de Programación.

Ya que se requiere estudiar las características de los lenguajes de programación con distintos niveles de detalle y perspectivas, se han propuesto diversas aproximaciones a la semántica formal:

- **Semántica operacional.**
  - Es el enfoque más antiguo (con este estilo se definió la semántica de ALGOL'60).
  - Prorciona una definición del lenguaje en términos de una máquina abstracta (de estados).
  - Debemos identificar una noción de estado abstracto que contenga la información esencial para describir el proceso de ejecución de un programa la máquina abstracta.
  - La semántica se define indicando cuál es el efecto de las construcciones sintácticas del lenguaje sobre el estado de la máquina abstracta.
  - Equivale a la definición de un intérprete (abstracto) para el lenguaje, que lo hace ejecutable.
  - A menudo, la especificación operacional de un lenguaje se lleva a cabo mediante el uso de *sistemas de transición*. (e.g., sistemas de transición de Plotkin).

- **Semántica axiomática.**
  - Es el enfoque típico de los trabajos sobre verificación fomal de programas imperativos.
  - Con este estilo se definió la semántica del lenguaje Pascal.
  - El significado de un programa es la relación existente entre las propiedades que verifica su entrada (*precondiciones*) y las propiedades que verifica su salida (*postcondiciones*).
  - Las precondiciones y postcondiciones son fórmulas de algún sistema lógico que especifican el significado del programa.
  - Las fórmulas pueden ser enunciados sobre el estado de la computación, pero no necesariamente debe ser así.

- **Semántica axiomática.**
  - `Notación`: $\{\mathcal{A}\}I\{\mathcal{B}\}$,

    Si el enunciado $\mathcal{A}$ es verdadero antes de la ejecución de la instrucción `I` entonces, el enunciado $\mathcal{B}$ es verdadero después de su ejecución .

  - $\{X \geq 0\}$ `FACT` $\{Y = X!\}$.

  - La semántica axiomática pretende introducir este tipo de construcciones en las pruebas de la lógica de predicados.

Regla	Antecedente	Consecuente
1. Consec_1	$\{\mathcal{A}\}I\{\mathcal{B}\}$, $\{\mathcal{B}\} \vdash \mathcal{C}$	$\{\mathcal{A}\}I\{\mathcal{C}\}$
2. Consec_2	$\{\mathcal{C}\} \vdash \mathcal{A}$, $\{\mathcal{A}\}I\{\mathcal{B}\}$	$\{\mathcal{C}\}I\{\mathcal{B}\}$
3. Compos.	$\{\mathcal{A}\}I_1\{\mathcal{B}\}$, $\{\mathcal{B}\}I_2\{\mathcal{C}\}$	$\{\mathcal{A}\}I_1; I_2\{\mathcal{C}\}$
4. Asign.	$X = \exp$	$\{\mathcal{A}(\exp)\}X = \exp\{\mathcal{A}(X)\}$
5. if	$\{\mathcal{A} \wedge B\}S_1\{\mathcal{C}\}$, $\{\mathcal{A} \wedge \neg B\}S_2\{\mathcal{C}\}$	$\{\mathcal{A}\}$`if` B `then` S_1 `else` $S_2\{\mathcal{C}\}$
6. while	$\{\mathcal{A} \wedge B\}S_1\{\mathcal{C}\}$	$\{\mathcal{A}\}$`while` B `do` $S\{\mathcal{A} \wedge \neg B\}$

Semántica, Sist. Formales y Leng. de Programación.

Semántica declarativa.

El significado de cada construcción sintáctica se define en términos de elementos y estructuras de un dominio matemático conocido.

- `Semántica teoría de modelos`.
  - Basada en la teoría de modelos de la lógica.
  - Con este estilo se ha definido la semántica de lenguajes de programación lógica como Prolog.
- `Semántica algebraica`.
  - Basada en la teoría de álgebras libres o $\mathcal{F}$-álgebras.
  - Con este estilo se ha definido la semántica de lenguajes ecuacionales de primer orden, una clase especial de lenguajes funcionales.
  - También de lenguajes de especificación de tipos abstractos de datos: OBJ y AXIS.

Semántica, Sist. Formales y Leng. de Programación.

- `Semántica de punto fijo`.
  - Basada en la teoría de funciones recursivas.
  - Se utiliza como enlace para demostrar la equivalencia entre diferentes caracterizaciones semánticas de un mismo lenguaje.
  - El significado del programa se define como el (menor) punto fijo de un operador de transformación (continuo).
  - Con este enfoque se ha definido la semántica de lenguajes lógicos.

- `Semántica denotacional`.
  - Basada en la teoría de dominios de Scott.
  - Las construcciones sintácticas de un lenguaje denotan valores de un determinado *dominio* matemático.
  - El significado del programa es la función que denota.
  - Se centra en "lo esencial": el resultado final obtenido. No se fija en los estados intermedios de la computación $\Longrightarrow$ mayor abstracción que la sem. operacional.

- `Semántica denotacional.`
  - Con este estilo se ha definido la semántica de la mayoría de los lenguajes funcionales (e.g. Haskell y ML) así como la del lenguaje imperativo ADA.
  - Técnicamente, es la más compleja de las caracterizaciones semánticas.
  - Es muy rica, ya que permite dar cuenta de:
    - computaciones que no terminan
    - orden superior
  - Para su definición se requiere describir:
    1. por cada construcción sintáctica un dominio sintáctico;
    2. los dominios semánticos;
    3. las funciones de evaluación semántica (dominio sintáctico $\longrightarrow$ dominio semántico);
    4. las ecuaciones semánticas.

2.3.3. Lenguajes de Programación Declarativa.

- Los lenguajes de programación declarativa son los que mejor se adaptan al concepto de sistema formal.

- Es posible definir un lenguaje de programación declarativa como un sistema constituido por tres componentes:

  1. La sintaxis.

     - las construcciones del lenguaje son fbf's de algún sistema lógico;

     - Los programas son conjuntos de fbf's (teorías) en esa lógica.

  2. La semántica operacional.

     - Una forma eficiente de deducción en esa lógica;

     - Se corresponde con el cálculo deductivo de los sistemas formales tradicionales $\Longrightarrow$ tiene un carácter sintáctico.

  3. La semántica declarativa.

- Debe existir una relación de corrección y completitud, de forma que las definiciones semánticas del lenguaje coincidan.

Capítulo 3

De la demostración automática a la Programación Lógica.

- La programación lógica surgió de las investigaciones realizadas (circa 1960) en el área de la demostración automática de teoremas.

- **Objetivo**: Presentar algunos de los tópicos de la demostración automática de teoremas que sirven de fundamento a la programación lógica.

3.1. Razonamiento Automático.

- El **razonamiento automático** es un campo de la inteligencia artificial (IA).

- Estudia como implementar programas que permitan verificar argumentos mediante el encadenamiento de pasos de inferencia, que siguen un sistema de reglas de inferencia (no necesariamente basadas en la lógica clásica).

- Enfasis en el razonamiento como proceso deductivo $\Longrightarrow$ **deducción automática**.

Razonamiento Automático.

- Si se centra en la obtención de algoritmos que permitan encontrar pruebas de teoremas matemáticos (expresados como fbf de la lógica de predicados) $\Longrightarrow$
  demostración automática de teoremas (DAT).

- Las técnicas desarrolladas en el área de la DAT son de especial interés para la programación lógica.

3.1.1. Aproximaciones a la demostración automática de teoremas.

Las aproximaciones a la DAT pueden clasificarse atendiendo a dos características esenciales:

1. La forma en la que se simula el razonamiento.

2. La cantidad de información intermedia retenida durante el proceso de la demostración.

Razonamiento Automático.

- La forma en la que se simula el razonamiento.
  Desde el inicio han existido dos orientaciones al problema de simular el razonamiento.

  1. Desarrollo de técnicas que simulan la forma de razocinio humano.
     - Trabajos pioneros: (circa 1950)
       Gelernter (geometría plana);
       Newell, Shaw y Simon (lógica de proposiciones).

     - Basados en métodos heurísticos, intentaban modelar la forma de raciocinio humana [ver Nil87].

     - Este tipo de técnicas fueron abandonadas debido a su fracaso a la hora de afrontar problemas más complejos.

     - **Razón**: Estructura del cerebro humano distinta a la de los computadores.
       - Los humanos hacemos uso intensivo dela regla de inferencia de instanciación.
       - Poder de deducción relativamente pequeño.
       - Conduce a la rápida aparición del problema de la explosión combinatoria.

2. Desarrollar técnicas que se adapten a su automatización en un computador.
   - Gilmore, Prawirtz, Davis, y Putman (circa 1960 – basados en el teorema de Herbrand).
   - Wos, Robinson y Carson (1964 – primer demostrador automático basado en el *principio de resolución* de J. A. Robinson, también tributario de los trabajos de Herbrand)
   - El principio de resolución permite la obtención de conclusiones generales a partir de premisas generales, aplicando lo menos posible la regla de instanciación.

1. La cantidad de información intermedia retenida.
   Son posibles tres alternativas:

   a) Almacenar toda la información derivada intermedia.
   - Problema: crecimiento del **espacio de búsqueda**.
   - Ejemplo: estrategia de resolución por saturación de niveles.

   b) No almacenar o minimizar la información intermedia retenida.
   - Problema: pérdida de potencia deductiva $\Longrightarrow$ poco adecuados para resolver problemas de complejidad mediana y grande.
   - Ejemplo: estrategia de resolución SLD.

   c) Retener sólo información que cumpla ciertos requisitos.
   - Ejemplos:
     - estrategia de resolución semántica;
     - métodos para el tratamiento de la información intermedia [Wos et al.].

3.1.2. Un demostrador automático genérico.

- Características (desde la perpectiva de Wos et al.):
  - Obtención de resultados intermedios (lemas) que serán los puntos de partida de una deducción lógica.
  - Uso de una representación especial de las fórmulas denominada **forma clausal**.
  - Eliminación información innecesaria o redundante: (*demodulación* + *subsumción*) $\Longrightarrow$ descartar expresiones (generalizandolas);
  - Empleo de reglas de inferencia alejadas del razonamiento humano: *Resolución* y *paramodulación*.
  - Pruebas por contradicción.

Razonamiento Automático.

1. Para utilizar un demostrador automático hay que expresar las fórmulas en forma clausal.

   Por ejemplo, si queremos probar el teorema:

   `Si` $\mathcal{A}$ `entonces` $\mathcal{B}$

   - los conocimientos relativos al campo del que se extrae el teorema (teoría – hipótesis de partida),
   - la condición $\mathcal{A}$ (*hipótesis especial*),
   - la negación de la conclusión $\mathcal{B}$.

2. Acciones típicas que lleva a cabo un demostrador de teoremas, son:

   a) Aplicación de las reglas de inferencia (restringida y dirigida por estrategias) para obtener conclusiones.

   b) Paso de las conclusiones a una forma canónica (*demodulación*).

   c) Análisis de las conclusiones.

   d) Si la conclusión ya se conoce (*test de subsumción*) entonces rechazarla.

   e) Si la conclusión obtenida entra en contradicción con alguna de las informaciones mantenidas entonces concluir con éxito, sino ir a 1.

3.1.3. Límites de la demostración automática de teoremas.

1. Limites teóricos:

   - La lógica de predicados es indecidible.

     No existe un procedimiento general de decisión que nos permita saber si una fórmula $\mathcal{A}$ es, o no, un teorema de $\mathcal{L}$.

   - El objetivo de la DAT es irrealizable en términos generales.

   - Sin embargo, existen procedimientos de semidecisión.

2. Limites prácticos: la complejidad computacional de los problemas objeto de estudio y los algoritmos utilizados en su solución.

   - Crecimiento exponencial del espacio de búsqueda (**explosión combinatoria**)
     $\Longrightarrow$ necesidad creciente de memoria.

   - Alto coste temporal de los algoritmos utilizados en el proceso de la demostración: unificación, subsumción, podas en el espacio de búsqueda, etc.

Razonamiento Automático.

A pesar de las limitaciones señaladas la demostración automática de teoremas ha alcanzado un grado de madurez considerable en los últimos veinte años que le ha llevado a superar con éxito muchos de sus objetivos iniciales.

- `Ejemplo`: El problema de Robbins que había intrigado a los matemáticos desde hacía más de 60 años.

Concluimos que, en el estudio de la DAT, los aspectos esenciales son:

1. el estudio de los procedimientos de prueba por refutación;

2. la definición de alguna forma de representación especial para las fórmulas;

3. el tipo de reglas de inferencia utilizadas y

4. las estrategias de búsqueda.

En los próximos apartados profundizaremos en cada uno de estos puntos.

3.2. Procedimiento de Prueba por Refutación.

- Los procedimientos de prueba basados en el teorema de Herbrand y en el principio de resolución de Robinson son procedimientos de *prueba por refutación*

- Un procedimiento de prueba por refutación consiste en lo siguiente:

  1. suponemos la falsedad de la conclusión (negamos lo que queremos probar);

  2. a partir de esta suposición tratamos de obtener una contradicción;

  3. si llegamos a una contradicción entonces, se rechazar el supuesto en vista del resultado y

  4. como consecuencia, se afirma la conclusión deseada.

Procedimiento de Prueba por Refutación.

- Justificación:
  - Un procedimiento de prueba trata de contestar a la pregunta

    ¿$\Gamma \models \mathcal{A}$?, donde $\mathcal{A}$ es una fbf de $\mathcal{L}$ y Γ un conjunto de fbf's de $\mathcal{L}$.

  - Alternativamente,

$$\begin{aligned}\Gamma \models \mathcal{A} &\Leftrightarrow \Gamma \cup \{\neg\mathcal{A}\} \text{ es insatisfacible} \\ &\Leftrightarrow \Gamma \cup \{\neg\mathcal{A}\} \text{ es inconsistente} \\ &\Leftrightarrow \Gamma \cup \{\neg\mathcal{A}\} \vdash \square\end{aligned}$$

- Punto de interés:
  - Nuestra pregunta original puede transformarse en la pregunta

    ¿La fórmula $(\mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_n \wedge \neg\mathcal{A})$ es insatisfacible?, donde $\mathcal{A}_1, \ldots, \mathcal{A}_n$ y $\mathcal{A}$ son fbf's de $\mathcal{L}$.

    supuesto que $\Gamma = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$.

  - Alternativamente, la inconsistencia de la fbf

    $(\mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_n \wedge \neg\mathcal{A})$.

  - Cobra sentido buscar formas equivalentes que la hagan más sencilla de manipular: **formas estándares**.

3.3. Formas Normales.

- El proceso obtención de una forma normal consiste en sustituir partes de la fórmula original por fórmulas lógicamente equivalentes.

- **Objetivo**: Obtener una fórmula que emplea un número restringido de conectivas y/o cuantificadores y tiene el mismo comportamiento que la fórmula original.

- Fórmula lógicamente equivalente en la lógica de predicados.
  - **Definición 3.3.1** *Dos fbf's* $\mathcal{A}$ *y* $\mathcal{B}$ *son* lógicamente equivalentes*, denotado* $\mathcal{A} \Leftrightarrow \mathcal{B}$*, sii para toda interpretación* $\mathcal{I}$*,* $\{\mathcal{A}\} \models \mathcal{B}$ *y* $\{\mathcal{B}\} \models \mathcal{A}$*.*

  - Intuitivamente: dos fbf son lógicamente equivalentes si una vez interpretadas y valuadas toman el mismo valor de verdad en cualquier circunstancia.

  - Son fórmulas identicas desde una perspectiva semántica.

Formas Normales.

Tabla 3.1. Fórmulas equivalentes de la lógica de proposiciones.

1	$(\mathcal{A} \leftrightarrow \mathcal{B}) \Leftrightarrow (\mathcal{A} \rightarrow \mathcal{B}) \wedge (\mathcal{B} \rightarrow \mathcal{A})$	
2	$(\mathcal{A} \rightarrow \mathcal{B}) \Leftrightarrow (\neg\mathcal{A} \vee \mathcal{B})$	
3	(a) $(\mathcal{A} \vee \mathcal{B}) \Leftrightarrow (\mathcal{B} \vee \mathcal{A})$	(b) $(\mathcal{A} \wedge \mathcal{B}) \Leftrightarrow (\mathcal{B} \wedge \mathcal{A})$
4	(a) $(\mathcal{A} \vee \mathcal{B}) \vee \mathcal{C} \Leftrightarrow \mathcal{A} \vee (\mathcal{B} \vee \mathcal{C})$	(b) $(\mathcal{A} \wedge \mathcal{B}) \wedge \mathcal{C} \Leftrightarrow \mathcal{A} \wedge (\mathcal{B} \wedge \mathcal{C})$
5	(a) $\mathcal{A} \vee (\mathcal{B} \wedge \mathcal{C}) \Leftrightarrow (\mathcal{A} \vee \mathcal{B}) \wedge (\mathcal{A} \vee \mathcal{C})$	(b) $\mathcal{A} \wedge (\mathcal{B} \vee \mathcal{C}) \Leftrightarrow (\mathcal{A} \wedge \mathcal{B}) \vee (\mathcal{A} \wedge \mathcal{C})$
6	(a) $(\mathcal{A} \vee \Box) \Leftrightarrow \mathcal{A}$	(b) $(\mathcal{A} \wedge \Diamond) \Leftrightarrow \mathcal{A}$
7	(a) $(\mathcal{A} \vee \Diamond) \Leftrightarrow \Diamond$	(b) $(\mathcal{A} \wedge \Box) \Leftrightarrow \Box$
8	(a) $(\mathcal{A} \vee \neg\mathcal{A}) \Leftrightarrow \Diamond$	(b) $(\mathcal{A} \wedge \neg\mathcal{A}) \Leftrightarrow \Box$
9	$\neg(\neg\mathcal{A}) \Leftrightarrow \mathcal{A}$	
10	(a) $\neg(\mathcal{A} \vee \mathcal{B}) \Leftrightarrow (\neg\mathcal{A} \wedge \neg\mathcal{B})$	(b) $\neg(\mathcal{A} \wedge \mathcal{B}) \Leftrightarrow (\neg\mathcal{A} \vee \neg\mathcal{B})$
11	(a) $(\mathcal{A} \vee \mathcal{A}) \Leftrightarrow \mathcal{A}$	(b) $(\mathcal{A} \wedge \mathcal{A}) \Leftrightarrow \mathcal{A}$

1. Al emplear las metavariables $\mathcal{A}$, $\mathcal{B}$ y $\mathcal{C}$, las relaciones de equivalencia se establecen entre esquemas de fórmulas.

2. El metasímbolo $\Box$ representa una fórmula insatisfacible (una *contradicción*).

3. El metasímbolo $\Diamond$ representa una fórmula lógicamente válida (una em tautogía).

4. Estas equivalencias son bien conocidas: (9) es la ley de la doble negación; (10.a) y (10.b) son las leyes de De Morgan etc.

   (Ver Apuntes de Lógica).

Tabla 3.2. Fórmulas equivalentes de la lógica de predicados.

12.a	$(QX)\mathcal{A}(X) \vee \mathcal{B} \Leftrightarrow (QX)(\mathcal{A}(X) \vee \mathcal{B})$
12.b	$(QX)\mathcal{A}(X) \wedge \mathcal{B} \Leftrightarrow (QX)(\mathcal{A}(X) \wedge \mathcal{B})$
13	(a) $\neg(\forall X)\mathcal{A}(X) \Leftrightarrow (\exists X)\neg\mathcal{A}(X)$ (b) $\neg(\exists X)\mathcal{A}(X) \Leftrightarrow (\forall X)\neg\mathcal{A}(X)$
14.a	$(\forall X)\mathcal{A}(X) \wedge (\forall X)\mathcal{C}(X) \Leftrightarrow (\forall X)(\mathcal{A}(X) \wedge \mathcal{C}(X))$
14.b	$(\exists X)\mathcal{A}(X) \vee (\exists X)\mathcal{C}(X) \Leftrightarrow (\exists X)(\mathcal{A}(X) \vee \mathcal{C}(X))$
15.a	$(Q_1X)\mathcal{A}(X) \vee (Q_2X)\mathcal{C}(X) \Leftrightarrow (Q_1X)(Q_2Z)(\mathcal{A}(X) \vee \mathcal{C}(Z))$
15.b	$(Q_3X)\mathcal{A}(X) \wedge (Q_4X)\mathcal{C}(X) \Leftrightarrow (Q_3X)(Q_4Z)(\mathcal{A}(X) \wedge \mathcal{C}(Z))$

1. La variable Z que se muestra en las equivalencias (15) es una variable que no aparece en la fórmula $\mathcal{A}(X)$.

2. En las equivalencias (15),

   - Si $Q_1 \equiv Q_2 \equiv \exists$ y $Q_3 \equiv Q_4 \equiv \forall$ entonces, no es necesario renombrar las apariciones de la variable X en $(Q_2X)\mathcal{C}(X)$ o en $(Q_4X)\mathcal{C}(X)$.

   - En este caso se puede usar directamente las equivalencias 14.

3.3.1. Formas normales en la lógica proposicional.

- La idea detrás de las formas normales en la lógica proposicional es el deseo de representar cualquier fórmula mediante conjuntos adecuados de conectivas.

- Un *conjunto adecuado* de conectivas: $\{\neg, \vee, \wedge\}$.

- **Definición 3.3.2 (forma normal disyuntiva)**
  Una fbf $\mathcal{A}$ está en forma normal disyuntiva *si y sólo si $\mathcal{A} \equiv \mathcal{A}_1 \vee \mathcal{A}_2 \vee \ldots \vee \mathcal{A}_n$, con $n \geq 1$, y cada $\mathcal{A}_i$ es una conjunción de literales.*

- **Definición 3.3.3 (forma normal conjuntiva)**
  Una fbf $\mathcal{A}$ está en forma normal conjuntiva *si y sólo si $\mathcal{A} \equiv \mathcal{A}_1 \wedge \mathcal{A}_2 \wedge \ldots \wedge \mathcal{A}_n$, con $n \geq 1$, y cada $\mathcal{A}_i$ es una disyunción de literales.*

Formas Normales.

- **Algoritmo 1**

  Entrada: *una fbf $\mathcal{A}$ de $\mathcal{L}$.*

  Salida: *una fórmula en forma normal conjuntiva (o disyuntiva) $\mathcal{B}$.*

  Comienzo

  P1. *Usar las reglas (1) y (2) de la Tabla 3.1 para eliminar las conectivas "$\leftrightarrow$" y "$\rightarrow$" de la fbf $\mathcal{A}$.*

  P2. *Emplear las reglas (9), (10.a) y (10.b) de la Tabla 3.1 para llevar la negación a los átomos.*

  P3. *Emplear las reglas distributivas (5.a) y (5.b) y el resto de las reglas de la Tabla 3.1 hasta obtener una forma normal conjuntiva (o disyuntiva) $\mathcal{B}$.*

  Devolver $\mathcal{B}$.

  Fin

- **Ejemplo 8** *Obtener una forma normal conjuntiva de la fórmula $\mathcal{A} \equiv (p \vee \neg q) \rightarrow r$.*

$$\begin{array}{lll} \mathcal{A} & \Leftrightarrow & \neg(p \vee \neg q) \vee r \qquad (2) \\ & \Leftrightarrow & (\neg p \wedge \neg(\neg q)) \vee r \qquad (10.a) \\ & \Leftrightarrow & (\neg p \wedge q) \vee r \qquad (9) \\ & \Leftrightarrow & r \vee (\neg p \wedge q) \qquad (3.a) \\ & \Leftrightarrow & (r \vee \neg p) \wedge (r \vee q) \qquad (5.a) \end{array}$$

Nótese que la fórmula $(\neg p \wedge q) \vee r$ (tercer paso) es una forma normal disyuntiva de $\mathcal{A}$.

3.3.2. Formas normales en la lógica de predicados.

- Cualquier fbf la lógica de predicados puede transformarse en una fórmula equivalente denominada *forma normal prenexa*.

- En una forma normal prenexa todos los cuantificadores deben encontrarse al principio de la fórmula.

- **Definición 3.3.4 (forma normal prenexa)**
  Una fbf $\mathcal{A}$ está en forma normal prenexa *sii*

  $$\mathcal{A} \equiv (Q_1 X_1) \ldots (Q_n X_n)\mathcal{M},$$

  donde $X_1, \ldots, X_n$ son variables diferentes, para cada $1 \leq i \leq n$, $Q_i \equiv \forall$ o bien $Q_i \equiv \exists$ y $\mathcal{M}$ es una fórmula que no contiene cuantificadores.

- $(Q_1 X_1) \ldots (Q_1 X_n)$ es el *prefijo* de $\mathcal{A}$.

- $\mathcal{M}$ es la *matríz* de $\mathcal{A}$.

- Una fórmula cerrada en forma prenexa $(\forall X_1) \ldots (\forall X_n)\mathcal{M}$ se denomina *fórmula universal*.

- Una fórmula cerrada en forma prenexa $(\exists X_1) \ldots (\exists X_n)\mathcal{M}$ se denomina *fórmula existencial*.

Formas Normales.

- **Algoritmo 2**

  Entrada: *una fbf $\mathcal{A}$ de $\mathcal{L}$.*

  Salida: *una fórmula en forma normal prenexa $\mathcal{B}$.*

  Comienzo

  P1. *Usar las reglas (1) y (2) de la Tabla 3.1 para eliminar las conectivas "$\leftrightarrow$" y "$\rightarrow$" de la fbf $\mathcal{A}$.*

  P2. *Emplear repetidamente las reglas (9), (10.a) y (10.b) de la Tabla 3.1 y la regla (13) de la Tabla 3.2 para llevar la negación a los átomos.*

  P3. *Renombrar las variables ligadas, si es necesario.*

  P4. *Emplear las reglas (12), (14) y (15) de la Tabla 3.2 para mover los cuantificadores a la izquierda y obtener una forma normal prenexa $\mathcal{B}$.*

  Devolver $\mathcal{B}$.

  Fin

- **Ejemplo 9** *Transformar la fórmula $\mathcal{A} \equiv (\forall X)p(X) \rightarrow (\exists X)q(X)$ a una forma normal prenexa.*

$$\begin{array}{rcll} \mathcal{A} & \Leftrightarrow & \neg(\forall X)p(X) \vee (\exists X)q(X) & (2) \\ & \Leftrightarrow & (\exists X)\neg p(X) \vee (\exists X)q(X) & (13.a) \\ & \Leftrightarrow & (\exists X)(\neg p(X) \vee q(X)) & (14.b) \end{array}$$

3.4. La Forma Clausal.

3.4.1. Forma normal de Skolem.

- **Algoritmo 3**

Entrada: *una fbf* $\mathcal{A}$ *de* $\mathcal{L}$.

Salida: *una fórmula en forma normal de Skolem* $\mathcal{B}$.

Comienzo

P1. *Usar el Algoritmo 2 para obtener una FNP* $(Q_1X_1)\ldots(Q_nX_n)\mathcal{M}$ *de* $\mathcal{A}$.

P2. *Usar el Algoritmo 1 para obtener la FNC de* $\mathcal{M}$.

P3. *Eliminar los cuantificadores existenciales. Aplicar los siguientes pasos: Supongamos que* $Q_r \equiv \exists$ *en el prefijo* $(Q_1X_1)\ldots(Q_nX_n)$, *con* $1 \leq r \leq n$.

1. *Si a* Q_r *no le precede ningún cuantificador universal, substituir cada aparición de* X_r *por una constante* c, *que sea diferente de cualquier otra constante que aparezca en* $\mathcal{M}$. *Borrar* (Q_rX_r) *del prefijo.*
2. *Si* $Q_{s_1}\ldots Q_{s_m}$ *son todos los cuantificadores universales que preceden a* Q_r, $1 \leq s_1 < s_2 < \ldots < s_m < r$, *elegir un nuevo símbolo de función* m*-ario,* f, *que no aparezca en* $\mathcal{M}$ *y reemplazar todas las apariciones de* X_r *en* $\mathcal{M}$ *por* $f(X_{s_1}, X_{s_2}, \ldots, X_{s_m})$. *Borrar* (Q_rX_r) *del prefijo.*

Dev. $\mathcal{B}$.

Fin

La Forma Clausal.

- En la fórmula $\mathcal{B}$ todos los cuantificadores son universales:

  $$(\forall X_{j_1}) \ldots (\forall X_{j_n}) \mathcal{M}(X_{j_1}, \ldots, X_{j_n})$$

  $\{X_{j_1}, \ldots, X_{j_n}\} \subseteq \{X_1, \ldots, X_n\}$.

- Las acciones del P3 reciben el nombre de *skolemización*.
  - Las constantes c y las funciones f, empleadas en la sustitución de variables existenciales se denominan *constantes de Skolem* y *funciones de Skolem*.

- **Ejemplo 10**

  $$\mathcal{A} \equiv (\exists X_1)(\forall X_2)(\forall X_3)(\exists X_4)(\forall X_5)(\exists X_6) p(X_1, X_2, X_3, X_4, X_5, X_6)$$

  - *Skolemización:*

  $$\begin{array}{rcll} \mathcal{A} & \Leftrightarrow & (\forall X_2)(\forall X_3)(\exists X_4)(\forall X_5)(\exists X_6) p(a, X_2, X_3, X_4, X_5, X_6) & P3\ (a) \\ & \Leftrightarrow & (\forall X_2)(\forall X_3)(\forall X_5)(\exists X_6) p(a, X_2, X_3, f(X_2, X_3), X_5, X_6) & P3\ (b) \\ & \Leftrightarrow & (\forall X_2)(\forall X_3)(\forall X_5) p(a, X_2, X_3, f(X_2, X_3), X_5, g(X_2, X_3, X_5)) & P3\ (b) \end{array}$$

  - *Forma normal de Skolem para $\mathcal{A}$:*

    $$(\forall X_2)(\forall X_3)(\forall X_5) p(a, X_2, X_3, f(X_2, X_3), X_5, g(X_2, X_3, X_5)).$$

- **Ejemplo 11**

  $$\mathcal{A} \equiv (\forall X_1)(\exists X_2)(\exists X_3)[(\neg p(X_1) \wedge q(X_2)) \vee r(X_2, X_3)]$$

  Forma normal de Skolem para $\mathcal{A}$:

  $$\begin{array}{rcll} \mathcal{A} & \Leftrightarrow & (\forall X_1)(\exists X_2)(\exists X_3)[(\neg p(X_1) \vee r(X_2, X_3)) \wedge (q(X_2) \vee r(X_2, X_3))] & P2,\ 5(a) \\ & \Leftrightarrow & (\forall X_1)(\exists X_3)[(\neg p(X_1) \vee r(f(X_1), X_3)) \wedge (q(f(X_1)) \vee r(f(X_1), X_3))] & P3\ (b) \\ & \Leftrightarrow & (\forall X_1)[(\neg p(X_1) \vee r(f(X_1), g(X_1))) \wedge (q(f(X_1)) \vee r(f(X_1), g(X_1)))] & P3\ (b) \end{array}$$

La Forma Clausal.

- Necesidad de introducir funciones de Skolem.

  Ejemplo 12 *La sentencia "todo el mundo tiene madre" puede formalizase mediante la fórmula:*

  $$(\forall X)(\exists Y)esMadre(Y, X),$$

  - *Posición ingenua: de sustituir la variable ligada Y por la constante "eva" (un cierto individuo que cumpliese la propiedad):*

    $$(\forall X)esMadre(eva, X).$$

  - *Esta es una formalización no se corresponde con el sentido de la sentencia original.*

  - *Para eliminar el cuantificador existencial conservando el sentido del enunciado original necesitamos introducir una función $madre(X)$:*

    $$(\forall X)esMadre(madre(X), X),$$

- En lo que sigue, emplearemos el término "*forma normal*" o "*forma estándar*" para referirnos a la forma normal de Skolem.

3.4.2. Cláusulas.

- **Definición 3.4.1 (cláusula)**
  Una cláusula *es una disyunción finita de cero o más literales.*

- **Cláusula unitaria**: si está compuesta por un literal.

- **Ejemplo 13** *En el Ejemplo 11 las disyunciones*

  $$\neg p(X_1) \vee r(f(X_1) \vee g(X_1))$$

  y

  $$q(f(X_1)) \vee r(f(X_1) \vee g(X_1))$$

  son cláusulas.

- Sea $\mathcal{A} \equiv (\forall X_1) \ldots (\forall X_n)\mathcal{M}$ una fórmula en forma estándar. Podemos expresarla como:

  $$\begin{array}{l}(\forall X_1) \ldots (\forall X_n)(\mathcal{M}_{11} \vee \mathcal{M}_{12} \vee \ldots) \wedge \\ (\forall X_1) \ldots (\forall X_n)(\mathcal{M}_{21} \vee \mathcal{M}_{22} \vee \ldots) \wedge \\ \vdots \\ (\forall X_1) \ldots (\forall X_n)(\mathcal{M}_{(n-1)1} \vee \mathcal{M}_{(n-1)2} \vee \ldots) \wedge \\ (\forall X_1) \ldots (\forall X_n)(\mathcal{M}_{n1} \vee \mathcal{M}_{n2} \vee \ldots)\end{array}$$

Cláusulas.

- Entonces, $\mathcal{A}$ pueda ser representada, de una manera más simple, mediante un conjunto de cláusulas Δ.

$$\Delta = \{(\mathcal{M}_{11} \vee \mathcal{M}_{12} \vee \ldots), (\mathcal{M}_{21} \vee \mathcal{M}_{22} \vee \ldots), \ldots, (\mathcal{M}_{n1} \vee \mathcal{M}_{n2} \vee \ldots)\}$$

- **Ejemplo 14** *forma normal del Ejemplo 11 como conjunto de cláusulas.*
  $\{(\neg p(X_1) \vee r(f(X_1), g(X_1))), (q(f(X_1))) \vee r(f(X_1), g(X_1)))\}$

- En un conjunto de cláusulas (que representan una forma normal) las variables deben suponerse cuantificadas universalmente.

- Las cláusulas son fórmulas cerradas y sus variables pueden renombrarse.

- En ocasiones conviene representar una cláusula como un conjunto de literales:

$$(L_1 \vee L_2 \vee \ldots) \Longrightarrow \{L_1, L_2, \ldots\}$$

- **Ejemplo 15** *Las cláusulas del Ejemplo 11 se representan mediante los conjuntos*
  $\{\neg p(X_1), r(f(X_1), g(X_1))\}$ *y* $\{q(f(X_1)), r(f(X_1), g(X_1))\}$.

- Una contradicción, $\Box \equiv (L \wedge \neg L)$, se representa mediante el conjunto vacío, $\{\}$ (**cláusula vacía**).

3.4.3. El papel de la forma clausal en los procedimientos de prueba.

- La eliminación de los cuantificadores existenciales de una fórmula para pasar a su forma normal no afecta a su inconsistencia.

- **Teorema 3.4.2** *Sea un conjunto de cláusulas Δ, que representa una forma estándar de una fórmula $\mathcal{A}$. La fórmula $\mathcal{A}$ es insatisfacible sii Δ es insatisfacible.*

- Si un conjunto de cláusulas Δ representa una forma normal estándar de una fórmula $(\neg\mathcal{A})$, la formula $\mathcal{A}$ será válida si y sólo si el conjunto Δ es insatisfacible.

- La equivalencia entre una fórmula $\mathcal{A}$ y su forma normal representada por un conjunto de cláusulas Δ, sólo se mantiene cuando $\mathcal{A}$ es insatisfacible.

  `Ejemplo`: Sea $\mathcal{A} \equiv (\exists X)p(X)$.

  - Su forma normal es la fórmula $\mathcal{B} \equiv p(a)$;

  - $\mathcal{A}$ no es equivalente a $\mathcal{B}$, ya que:
    - Dada la interpretación $\mathcal{I} = \langle \mathcal{D}, \mathcal{J} \rangle$: $\mathcal{D} = \{1, 2\}$, $\mathcal{J}(a) = 1$, $\mathcal{J}(p) = \overline{p}$, con $\overline{p}(1) = \mathtt{F}$ y $\overline{p}(2) = \mathtt{V}$.

    - Claramente, $\mathcal{A}$ es satisfecha por $\mathcal{I}$, pero $\mathcal{B}$ es falsa en $\mathcal{I}$, por lo que $\mathcal{A} \not\Leftrightarrow \mathcal{B}$.

- El Teorema 3.4.2 nos dice que basta con:

  1. utilizar como entrada a un procedimiento de prueba por refutación un conjunto de cláusulas Δ;

  2. probar la insatisfacibilidad de Δ, que representa la forma estándar de una fórmula (original).

3.5. Teorema de Herbrand.

- Una fórmula es insatisfacible si y sólo si es falsa para toda interpretación sobre cualquier dominio.

- **Problema**: Para probar la insatisfacibilidad de una fórmula cualquiera hay que considerar todas las interpretaciones posibles.

- Afortunadamente, para comprobar la insatisfacibilidad de un conjunto de cláusulas basta investigar las interpretaciones sobre un dominio concreto: el **universo de Herbrand**.

3.5.1. Universo de Herbrand e interpretaciones de Herbrand.

- Un conjunto de fórmulas Γ generan un lenguaje de primer orden cuyo alfabeto está constituido por los símbolos de constante, variable, función y relación que aparecen en dichas fórmulas.

- Si no hay constantes en ese alfabeto, se añadirá una constante artificial "a".

- **Ejemplo 16** *Sea* $\mathcal{A} \equiv (\exists Y)(\forall X)p(g(X), f(Y))$. *El lenguaje de primer orden generado por* $\mathcal{A}$ *tiene por alfabeto:*

$$\begin{array}{lll} \mathcal{C} & = & \{a\} \\ \mathcal{V} & = & \{X, Y\} \end{array} \qquad \begin{array}{lll} \mathcal{F} & = & \{f^1, g^1\} \\ \mathcal{P} & = & \{p^2\} \end{array}$$

- **Definición 3.5.1 (Universo de Herbrand)**
Dado un lenguaje de primer orden $\mathcal{L}$*, el universo de Herbrand* $\mathcal{U}_{\mathcal{L}}$ *de* $\mathcal{L}$ *es el conjunto de todos los términos básicos de* $\mathcal{L}$

- Notación: $\mathcal{U}_{\mathcal{L}}(\Gamma)$, Si queremos hacer expícito el conjunto de fórmulas Γ.

Teorema de Herbrand.

- **Algoritmo 4**

  Entrada: *Un conjunto de fórmulas* Γ. *Un valor límite* n.

  Salida: *El conjunto de términos básicos de nivel* n *de* Γ, $\mathcal{U}_n(\Gamma)$.

  Comienzo

  1. *Construir el conjunto* $\mathcal{U}_0(\Gamma)$ *compuesto por todos los símbolos de constante que aparecen en* Γ. *Si en* Γ *no aparecen símbolos de constante entonces,* $\mathcal{U}_0(\Gamma) = \{a\}$.
  2. `Para` $i = 1$ `hasta` n `hacer`
     1. *Construir el conjunto* $\mathcal{T}_i(\Gamma)$ *compuesto por todos los términos básicos,* $f^k(t_1, \ldots, t_n)$, *que pueden formarse combinando todos los símbolos de función* f^k *que aparecen en* Γ *con los términos* t_j, $j = 1, \ldots, k$, *tales que* $t_j \in \mathcal{U}_{i-1}(\Gamma)$;
     2. $\mathcal{U}_i(\Gamma) = \mathcal{T}_i(\Gamma) \cup \mathcal{U}_{i-1}(\Gamma)$.

  Dev. $\mathcal{U}_n(\Gamma)$.

  Fin

- El universo de herbrand del conjunto Γ es $\mathcal{U}_{\mathcal{L}}(\Gamma) = \mathcal{U}_{\infty}(\Gamma)$.

Teorema de Herbrand.

- **Ejemplo 17** *Para la fbf $\mathcal{A}$ del Ejemplo 16 tenemos que:*

$$\begin{aligned}
\mathcal{U}_0(\mathcal{A}) &= \{a\} \\
\mathcal{U}_1(\mathcal{A}) &= \{f(a), g(a)\} \cup \mathcal{U}_0(\mathcal{A}) \\
&= \{a, f(a), g(a)\} \\
\mathcal{U}_2(\mathcal{A}) &= \{f(a), f(f(a)), f(g(a)), g(a), g(f(a)), g(g(a))\} \cup \mathcal{U}_1 \\
&= \{a, f(a), f(f(a)), f(g(a)), g(a), g(f(a)), g(g(a))\} \\
&\vdots
\end{aligned}$$

- **Ejemplo 18** $\Delta = \{p(a), \neg p(X) \vee p(f(X))\}$.

  - *Lenguaje de primer orden generado por Δ (alfabeto):*

$$\begin{aligned}
\mathcal{C} &= \{a\} \qquad & \mathcal{F} &= \{f^1\} \\
\mathcal{V} &= \{X\} \qquad & \mathcal{P} &= \{p^1\}
\end{aligned}$$

  - *Universo de Herbrand para el conjunto Δ:*

$$\begin{aligned}
\mathcal{U}_0(\Delta) &= \{a\} \\
\mathcal{U}_1(\Delta) &= \{f(a)\} \cup \mathcal{U}_0(\Delta) = \{a, f(a)\} \\
\mathcal{U}_2(\Delta) &= \{f(a), f(f(a))\} \cup \mathcal{U}_1(\Delta) = \{a, f(a), f(f(a))\} \\
&\vdots \\
\mathcal{U}_{\mathcal{L}}(\Delta) &= \{a, f(a), f(f(a)), f(f(f(a))), f(f(f(f(a)))), \ldots
\end{aligned}$$

Teorema de Herbrand.

- **Definición 3.5.2 (Base de Herbrand)**
  Dado un lenguaje de primer orden $\mathcal{L}$, la base de Herbrand *$\mathcal{B}_{\mathcal{L}}$ de $\mathcal{L}$ es el conjunto de todos los átomos básicos de $\mathcal{L}$.*

- Notación: $\mathcal{B}_{\mathcal{L}}(\Gamma)$, Si queremos hacer expícito el conjunto de fórmulas Γ.

- **Ejemplo 19** $\Delta = \{p(a), \neg p(X) \vee q(f(X))\}$.
  - *El lenguaje de primer orden generado por Δ (alfabeto):*

    $$\begin{array}{llll} \mathcal{C} & = & \{a\} & \mathcal{F} = \{f^1\} \\ \mathcal{V} & = & \{X\} & \mathcal{P} = \{p^1, q^1\} \end{array}$$

  - *Universo de Herbrand,*

    $$\mathcal{U}_{\mathcal{L}}(\Delta) = \{a, f(a), f(f(a)), f(f(f(a))), f(f(f(f(a)))), \ldots\}$$

  - *Base de Herbrand es:*

    $$\begin{array}{rl} \mathcal{B}_{\mathcal{L}}(\Delta) = & \{p(a), p(f(a)), p(f(f(a))), p(f(f(f(a)))), \ldots \\ & q(a), q(f(a)), q(f(f(a))), q(f(f(f(a)))), \ldots\} \end{array}$$

Teorema de Herbrand.

- **Definición 3.5.3 (Interpretación de Herbrand)**
  Una interpretación de Herbrand *de* $\mathcal{L}$ *(*H-interpretación*) es una interpretación* $\mathcal{I} = (\mathcal{D}_\mathcal{I}, \mathcal{J})$ *de* $\mathcal{L}$ *tal que:*

  1. $\mathcal{D}_\mathcal{I}$ *es el universo de Herbrand* $\mathcal{U}_\mathcal{L}$.
  2. $\mathcal{J}$ *es la aplicación que asigna:*
     - *A cada símbolo de constante* a_i *de* $\mathcal{L}$ *él mismo; i.e.,* $\mathcal{J}(a_i) = a_i$.
     - *A cada functor* ${f_i}^n$ *n-ario de* $\mathcal{L}$ *una función* $\mathcal{J}({f_i}^n) = \overline{f_i}^n$,
       $$\overline{f_i}^n : \mathcal{U}_\mathcal{L}^n \longrightarrow \mathcal{U}_\mathcal{L}$$
       $$(t_1, \ldots, t_n) \hookrightarrow {f_i}^n(t_1, \ldots, t_n).$$
     - *Si* ${r_i}^n$ *es un símbolo de relación n-ario de* $\mathcal{L}$ *entonces, se le asigna*
       $$\mathcal{J}({r_i}^n) = \overline{r_i}^n \subseteq \mathcal{U}_\mathcal{L}^n.$$

- No hay restricción respecto de las relaciones de $\mathcal{U}_\mathcal{L}^n$ asignadas a cada uno de los símbolos de relación. $\Longrightarrow$ hay varias interpretaciones de herbrand para $\mathcal{L}$.

Teorema de Herbrand.

- Una H-interpretación queda unívocamente determinada dando la interpretación de sus símbolos de relación.

- Hay una correspondencia uno a uno entre las distintas H-interpretaciones y los subconjuntos de la base de Herbrand, $\mathcal{B}_{\mathcal{L}}$:
  $\{{r_i}^n(t_1, \ldots, t_n) \mid {r_i}^n$ es un símbolo de relación n-ario de $\mathcal{L}$ y $(t_1, \ldots, t_n) \in \overline{r_i}^n\}$.

- Una H-interpretación puede entenderse como un subconjunto $\mathcal{B}_{\mathcal{L}}$ que representa el conjunto de los átomos básicos que se evalúan al valor de verdad V (los átomos que no están en el conjunto, se evalúan al valor de verdad F).

Teorema de Herbrand.

- **Ejemplo 20** $\Delta = \{p(a), \neg p(X) \vee q(X)\}$.
  - *Lenguaje generado por* Δ *(alfabeto):*

    $\mathcal{C} = \{a\} \qquad \mathcal{F} = \emptyset$
    $\mathcal{V} = \{X\} \qquad \mathcal{P} = \{p^1, q^1\}$

  - *Universo de Herbrand para el conjunto* Δ*:*

    $\mathcal{U}_{\mathcal{L}}(\Delta) = \{a\}$

  - *Base de Herbrand:*

    $\mathcal{B}_{\mathcal{L}}(\Delta) = \{p(a), q(a)\}$

  - *Son posibles las siguientes H-interpretaciones para* Δ*:*

    $\mathcal{I}_1 = \{p(a), q(a)\}$
    $\mathcal{I}_2 = \{p(a)\}$ *i.e., (*$q(a)$ *es falso)*
    $\mathcal{I}_3 = \{q(a)\}$ *i.e., (*$p(a)$ *es falso)*
    $\mathcal{I}_4 = \emptyset$

- Las anteriores H-interpretaciones se corresponden con la siguiente tabla de verdad:

	$p(a)$	$q(a)$
$\mathcal{I}_1$	V	V
$\mathcal{I}_2$	V	F
$\mathcal{I}_3$	F	V
$\mathcal{I}_4$	F	F

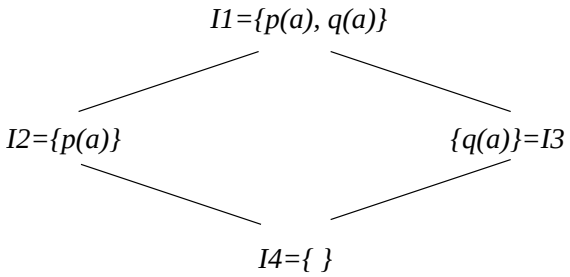
Teorema de Herbrand.

- El conjunto de H-interpretaciones de $\mathcal{L}$ es $\wp(\mathcal{B}_{\mathcal{L}})$.

- El número de H-interpretaciones de $\mathcal{L}$ es $2^{|\mathcal{B}_{\mathcal{L}}|}$, donde $|\mathcal{B}_{\mathcal{L}}|$ es el cardinal del conjunto $\mathcal{B}_{\mathcal{L}}$.

- El conjunto $\wp(\mathcal{B}_{\mathcal{L}})$ es un retículo completo cuyo máximo es el conjunto $\mathcal{B}_{\mathcal{L}}$ y cuyo mínimo es el conjunto vacío $\emptyset$.

- El orden entre los elementos de un retículo pueden ilustrarse mediante un diagrama de Hasse.



- Propiedad importante de las H-interpretaciones: basta explorar las H-interpretaciones de una fórmula para probar su insatisfacibilidad

  Teorema 3.5.4 *Sea Δ un conjunto de cláusulas. Δ es insatisfacible sii Δ es falsa bajo todas las H-interpretaciones de Δ.*

Teorema de Herbrand.

- Otras propiedades interesantes de las H-interpretaciones en relación con la satisfacibilidad:

  Definición 3.5.5 *Sea Δ un conjunto de cláusulas. Una* instancia básica *de un a cláusula $\mathcal{C}$ de Δ es la cláusula obtenida substituyendo cada una de las variables de $\mathcal{C}$ por elementos de $\mathcal{U}_{\mathcal{L}}(\Delta)$.*

  Proposición 3.5.6 *Sea Δ un conjunto de cláusulas e $\mathcal{I}$ una H-interpretación para Δ.*

  1. *Sea $\mathcal{C}'$ una instancia básica de una cláusula $\mathcal{C}$ de Δ, $\mathcal{C}'$ es verdadera en $\mathcal{I}$, sii*
     a) *existe un literal positivo L'_1 de $\mathcal{C}'$ tal que $L'_1 \in \mathcal{I}$ o*

     b) *existe un literal negativo L'_2 de $\mathcal{C}'$ tal que $\neg L'_2 \notin \mathcal{I}$.*
  2. *Sea $\mathcal{C}$ una cláusula de Δ, $\mathcal{C}$ es verdadera en $\mathcal{I}$ sii para toda instancia básica $\mathcal{C}'$ de $\mathcal{C}$ se cumple que $\mathcal{C}'$ es verdadera en $\mathcal{I}$.*
  3. *Sea $\mathcal{C}$ una cláusula de Δ, $\mathcal{C}$ es falsa en $\mathcal{I}$ sii existe al menos una instancia básica $\mathcal{C}'$ de $\mathcal{C}$ que no es verdadera en $\mathcal{I}$.*

3.5.2. Modelos de Herbrand.

- **Definición 3.5.7 (modelo de Herbrand)**
  Un modelo de Herbrand *para un conjunto de fórmulas* Γ *es una H-interpretación* $\mathcal{I}$ *que es modelo de* Γ.

- **Ejemplo 21** $\Delta = \{p(b), \neg p(X) \vee p(f(X)), \neg p(X) \vee q(b)\}$.
  - *Lenguaje generado por* Δ *(alfabeto):*

    $$\begin{array}{llll} \mathcal{C} & = & \{b\} & \mathcal{F} = \{f^1\} \\ \mathcal{V} & = & \{X\} & \mathcal{P} = \{p^1, q^1\} \end{array}$$

  - *Universo de Herbrand para* Δ*:*

    $$\mathcal{U}_{\mathcal{L}}(\Delta) = \{b, f(b), f(f(b)), f(f(f(b))), \ldots\}$$

  - *Base de Herbrand:*

    $$\begin{array}{rcl} \mathcal{B}_{\mathcal{L}}(\Delta) & = & \{p(b), p(f(b)), p(f(f(b))), p(f(f(f(b)))), \ldots \\ & & \quad q(b), q(f(b)), q(f(f(b))), q(f(f(f(b)))), \ldots\} \end{array}$$

  - *Algunas H-interpretaciones para* Δ *son:*

    $$\begin{array}{rcl} \mathcal{I}_1 & = & \{p(b), q(b)\} \\ \mathcal{I}_2 & = & \{p(b), p(f(b)), p(f(f(b))), p(f(f(f(b)))), \ldots\} \\ \mathcal{I}_3 & = & \{q(b), p(b), p(f(b)), p(f(f(b))), p(f(f(f(b)))), \ldots\} \end{array}$$

Teorema de Herbrand.

- *Haciendo uso de la Proposición 3.5.6 puede comprobarse que:*
  - $\mathcal{I}_1$ *no es modelo de* Δ.
  - $\mathcal{I}_2$ *no es modelo de* Δ.
  - $\mathcal{I}_3$ *es modelo de* Δ.

- **Observación 3.5.8**
  Sea un conjunto C *y* $S = \{S_1, S_2, \ldots, S_n\} \subseteq \wp(C)$. *Entonces* $S_1 \cap S_2 \cap \ldots \cap S_n$ *(respecto* $S_1 \cup S_2 \cup \ldots \cup S_n$*) es el ínfimo (supremo) del conjunto* S.

- Es posible la construcción de un modelo mínimo mediante la intersección de modelos.

3.5.3. El teorema de Herbrand y las dificultades para su implementación.

- El Teorema 3.5.4 y la Proposición 3.5.6 sugieren un **método semántico para probar la insatisfacibilidad** de un conjunto de cáusulas Δ:

  Generar todas las posibles H-interpretaciones e ir comprobando si cada una de éstas hace falsa alguna instancia básica $\mathcal{C}_i'$ de alguna $\mathcal{C}_i \in \Delta$.

- Problema: el número de H-interpretaciones puede ser infinito.

- Solución: Herbrand (1930)

  Para probar la insatisfacibilidad de un conjunto de cláusulas Δ basta con generar un conjunto finito de H-interpretaciones parciales que hacen falsa alguna instancia básica $\mathcal{C}_i'$ de alguna $\mathcal{C}_i \in \Delta$.

- Una H-interpretación parcial es una estructura en la que sólo se han asignado valores de verdad a una parte de los átomos de la base de Herbrand.

Teorema de Herbrand.

- **Ejemplo 22** $\Delta = \{p(X), \neg p(f(a))\}$.
  - *Lenguaje generado por* Δ *(alfabeto):*

    $\mathcal{C} = \{a\}$ $\quad$ $\mathcal{F} = \{f^1\}$
    $\mathcal{V} = \{X\}$ $\quad$ $\mathcal{P} = \{p^1\}$

  - *Universo de Herbrand:*

    $\mathcal{U}_{\mathcal{L}}(\Delta) = \{a, f(a), f(f(a)), f(f(f(a))), f(f(f(f(a)))), \ldots\}$

  - *Base de Herbrand:*

    $\mathcal{B}_{\mathcal{L}}(\Delta) = \{p(a), p(f(a)), p(f(f(a))), p(f(f(f(a)))), \ldots\}$

  - *Interpretaciones parciales para* Δ*:*
    - *Definimos la H-interpretación parcial* $\mathcal{I}'_1$ *como aquella en la que* $p(a) \notin \mathcal{I}'_1$.
    - *Definimos la H-interpretación parcial* $\mathcal{I}'_2$ *como aquella en la que* $p(a) \in \mathcal{I}'_2$ *y* $p(fa) \in \mathcal{I}'_2$.
    - *Definimos la H-interpretación parcial* $\mathcal{I}'_3$ *como aquella en la que* $p(a) \in \mathcal{I}'_3$ *pero* $p(fa) \notin \mathcal{I}'_3$.

  - *No importa el valor de verdad que tomen el resto de los átomos de* $\mathcal{B}_{\mathcal{L}}(\Delta)$ *en estas interpretaciones.*

Teorema de Herbrand.

- *Este conjunto de interpretaciones parciales es* **completo** *en el sentido de que cualquier posible H-interpretación está en* $\mathcal{I}'_1$, $\mathcal{I}'_2$ *o* $\mathcal{I}'_3$.

- *Un conjunto de interpretaciones parciales completo es suficiente para probar la insatisfacibilidad de* Δ*:*

  Cada una de las H-interpretaciones parciales hace falsa una instancia básica de una cláusula de $\Delta \Longrightarrow \Delta$ *es insatisfacible.*

- *El resultado de agrupar las instancias básicas que son falsas en cada una de las H-interpretaciones parciales es el conjunto* $\Delta' = \{p(a), p(f(a)), \neg p(f(a))\}$, *que es insatisfacible.*

- EJERCICIO: Extraer un conjunto finito de instancias básicas de cardinalidad mínima para Δ.

- **Teorema 3.5.9 (Teorema de Herbrand)** *Un conjunto de cláusulas* Δ *es insatisfacible sii existe un conjunto de instancias básicas de* Δ *insatisfacible.*

Teorema de Herbrand.

- Procedimiento de prueba por refutación basado el teorema de Herbrand (Gilmore – 1960).

Algoritmo 5

Entrada: *Un conjunto de cláusulas* Δ.

Salida: *Un conjunto de cláusulas básicas* Δ' *insatisfacible.*

Comienzo

Inicialización: $i = 0$.

Repetir

1. *Generar un conjunto de cáusulas básicas* Δ'_i *a partir de* Δ*:*
   sustituyendo las variables de las cláusulas de Δ *por términos básicos de* $\mathcal{U}_i(\Delta)$.
2. *Comprobar la insatisfacibilidad de* Δ'_i*:*
   utilizando el método de la multiplicación de la lógica de proposiciones.
3. `Si` Δ'_i *no es insatisfacible* `entonces` $i = i + 1$.

Hasta que Δ'_i *sea insatisfacible.*

Devolver $\Delta' = \Delta'_i$.

Fin

- Si Δ no es insatisfacible el Algoritmo 5 no terminará.

Teorema de Herbrand.

- El **método de la multiplicación**:

  1. considerar la conjunción de las cláusulas que formán Δ'_i y "multiplicarla" hasta alcanzar una forma normal disyuntiva.

  2. Cada conjunción que contiene un par de literales complementarios es eliminada.

  3. Repetir el proceso hasta que Δ'_i se transforme en la cláusula vacía $\Box$. Entonces, Δ'_i es insatisfacible.

- **Ejemplo 23** $\Delta = \{p(X), \neg p(f(a))\}$.

  - *Aplicamos el Algoritmo 5 y después de dos iteraciones se prueba que* Δ *es insatisfacible:*

    1. $\mathcal{U}_0(\Delta) = \{a\}$; $\Delta'_0 = \{p(a), \neg p(f(a))\}$. *Se aplica el método de la multiplicación:*
    $$\Delta'_0 = p(a) \wedge \neg p(f(a)) \neq \Box.$$
    2. $\mathcal{U}_1(\Delta) = \{a, f(a)\}$; $\Delta'_1 = \{p(a), p(f(a)), \neg p(f(a))\}$. *Se aplica el método de la multiplicación:*
    $$\Delta'_1 = p(a) \wedge p(f(a)) \wedge \neg p(f(a)) = p(a) \wedge \Box = \Box.$$

  - *El Algoritmo 5 termina devolviendo:*
  $$\Delta' = \{p(a), p(f(a)), \neg p(f(a))\}.$$

- **Ejemplo 24** $\Delta = \{p(a), \neg p(X) \vee q(f(X)), \neg q(f(a))\}$.
  - *El Algoritmo 5 prueba la insatisfacibilidad de* Δ *en la primera iteración:*

    $\mathcal{U}_0(\Delta) = \{a\}; \Delta'_0 = \{p(a), (\neg p(a) \vee q(f(a))), \neg q(f(a))\}$.

    Aplicamos el método de la multiplicación:

$$\begin{aligned}
\Delta'_0 &= p(a) \wedge (\neg p(a) \vee q(f(a))) \wedge \neg q(f(a)) \\
&= [(p(a) \wedge \neg p(a)) \vee (p(a) \wedge q(f(a)))] \wedge \neg q(f(a)) \\
&= (p(a) \wedge \neg p(a) \wedge \neg q(f(a))) \vee \\
&\quad (p(a) \wedge q(f(a)) \wedge \neg q(f(a))) \\
&= (\square \wedge \neg q(f(a))) \vee (p(a) \wedge \square) \\
&= \square \vee \square \\
&= \square
\end{aligned}$$

  - *El Algoritmo 5 termina devolviendo:*

    $\Delta' = \{p(a), (\neg p(a) \vee q(f(a))), \neg q(f(a))\}$.

Teorema de Herbrand.

- **Críticas** al Algoritmo 5:
  - Los ejemplos anteriores se han escogido para facilitar la aplición del Algoritmo 5, lo que da lugar a una engañosa apariencia de simplicidad.
  - El método de la multiplicación es muy ineficiente y conduce al conocido problema de la *explosión combinatoria*.

    Un conjunto con diez cláusulas básicas de dos literales cada una, da lugar a 2^{10} conjunciones de diez literales.
  - Las mejoras introducidas por Davis y Putnam no eliminaban los problemas.

3.6. El Principio de Resolución de Robinson.

- Objetivo: Evitar las ineficiencias de las implementaciones directas del teorema de Herbrand.
  - Generación sistemática de conjuntos de cláusulas básicas.
  - Comprobación de su insatisfacibilidad.
- Se aleja de los métodos basados en el teorema de Herbrand al permitir deducir consecuencias universales de enunciados universales $\Longrightarrow$ reduce las instanciaciones.
- Se aplica a fórmulas en forma clausal.
- Junto con el procedimiento de unificación, constituye un sistema de inferencia completo.
- Es más adecuado para la mecanización que los sistemas de inferencia tradicionales.

3.6.1. El Principio de Resolución en la lógica de proposiciones.

- El Principio de resolución de Robinson para la lógica de proposiciones:

$$\frac{\mathcal{A} \in \mathcal{C}_1, \neg\mathcal{A} \in \mathcal{C}_2}{(\mathcal{C}_1 \setminus \{\mathcal{A}\}) \cup (\mathcal{C}_2 \setminus \{\neg\mathcal{A}\})}$$

  - $\mathcal{C}_1$ y $\mathcal{C}_2$ se denominan *cláusulas padres*.
  - $\mathcal{A}$ y $\neg\mathcal{A}$ son los *literales resueltos*.
  - A la cláusula obtenida se le llama *resolvente* de $\mathcal{C}_1$ y $\mathcal{C}_2$.

- **Ejemplo 25**

$$\begin{array}{rcl} \mathcal{C}_1 & : & (\boxed{p} \vee r) \\ \mathcal{C}_2 & : & (\boxed{\neg p} \vee q) \\ \hline & & (r \vee q) \end{array}$$

- **Ejemplo 26**

$$\begin{array}{rcl} \mathcal{C}_1 & : & (\neg p \vee \boxed{q} \vee r) \\ \mathcal{C}_2 & : & (\boxed{\neg q} \vee s) \\ \hline & & (\neg p \vee r \vee s) \end{array}$$

- **Ejemplo 27** $\mathcal{C}_1 \equiv (\neg p \vee q)$ *y* $\mathcal{C}_2 \equiv (\neg p \vee r)$.
  - *No existe un literal en* $\mathcal{C}_1$ *que sea complementario a otro en* $\mathcal{C}_2 \Longrightarrow$ *no existe resolvente de* $\mathcal{C}_1$ *y* $\mathcal{C}_2$.

El Principio de Resolución de Robinson.

- El principio de resolución es una regla de inferencia muy potente que incorpora otras reglas de inferencia de la lógica.

Cláusulas padres	**Resolvente**	**Observación**
$\mathcal{C}_1: \ p \vee q$ $\mathcal{C}_2: \ \neg p$	$\mathcal{C}: \ q$	Equivale a la regla de inferencia del *silogismo disyuntivo*.
$\mathcal{C}_1: \ \neg p \vee q$ $\mathcal{C}_2: \ p$	$\mathcal{C}: \ q$	Ya que, $(\neg p \vee q) \Leftrightarrow (p \rightarrow q)$ lo anterior equivale a la regla de inferencia de *modus ponens*: Si $(p \rightarrow q)$ y p entonces q.
$\mathcal{C}_1: \ \neg p \vee q$ $\mathcal{C}_2: \ \neg q$	$\mathcal{C}: \ \neg p$	Equivale a la regla de inferencia de *modus tollens*: Si $(p \rightarrow q)$ y $\neg q$ entonces $\neg p$.
$\mathcal{C}_1: \ \neg p \vee q$ $\mathcal{C}_2: \ \neg q \vee r$	$\mathcal{C}: \ \neg p \vee r$	Equivale a la regla de inferencia del *silogismo hipotético*: Si $p \rightarrow q$ y $q \rightarrow r$ entonces $p \rightarrow r$.
$\mathcal{C}_1: \ p \vee q$ $\mathcal{C}_2: \ \neg p \vee q$	$\mathcal{C}: \ q$	La cláusula $q \vee q$ es lógicamente equivale a q. Este resolvente se denomina *fusión*.
$\mathcal{C}_1: \ p \vee q$ $\mathcal{C}_2: \ \neg p \vee \neg q$	$\mathcal{C}: \ p \vee \neg p$ $\mathcal{C}: \ q \vee \neg q$	Son posibles dos resolventes; ambos son tautologías.
$\mathcal{C}_1: \ p$ $\mathcal{C}_2: \ \neg p$	$\mathcal{C}: \ \square$	El resolvente de dos cláusulas unitarias es la cláusula vacía.

El Principio de Resolución de Robinson.

- **Definición 3.6.1 (Deducción)**
  Dado un conjunto de cláusulas Δ, *una* deducción *(*resolución*) de* $\mathcal{C}$ *a partir de* Δ *es un secuencia finita* $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$ *de cláusulas tal que:*

  - $\mathcal{C}_i$ *es una cláusula de* Δ *o*
  - *un resolvente de cláusulas que preceden a* $\mathcal{C}_i$, *y*
  - $\mathcal{C}_k = \mathcal{C}$.

  Una deducción de $\square$ *a partir de* Δ *se denomina* refutación *o* prueba *de* Δ.

- **Propiedades**:
  - **Teorema 3.6.2** *Dadas dos cláusulas* $\mathcal{C}_1$ *y* $\mathcal{C}_2$, *un resolvente* C *de* $\mathcal{C}_1$ *y* $\mathcal{C}_2$ *es una consecuencia lógica de* $\mathcal{C}_1$ *y* $\mathcal{C}_2$.
  - **Teorema 3.6.3** *Sea* Δ *un conjunto de cláusulas.* Δ *es insatisfacible si y sólo si existe una deducción de la cláusula vacía,* $\square$, *a partir de* Δ.

- **Ejemplo 28** $\Delta = \{(\neg p \vee q), \neg q, p\}$,
  - *Es posible la siguiente deducción:*

    (1) $\neg p \vee q$
    (2) $\neg q$
    (3) p

  - *generamos los siguientes resolventes*

    (4) $\neg p$ *de* (1) *y* (2)
    (5) $\square$ *de* (3) *y* (4)

  - *Por tanto,* Δ *es insatisfacible.*

El Principio de Resolución de Robinson.

- **Ejemplo 29** $\Delta = \{(p \vee q), (\neg p \vee q), (p \vee \neg q), (\neg p \vee \neg q)\}$,
  - *Es posible la siguiente deducción:*

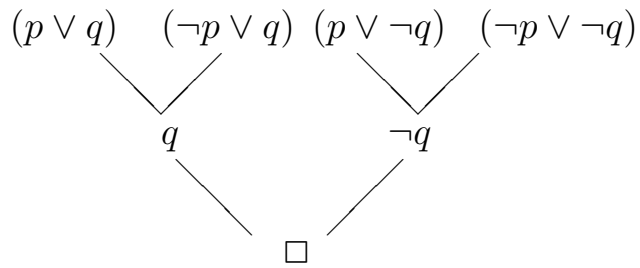
    (1) $p \vee q$
    (2) $\neg p \vee q$
    (3) $p \vee \neg q$
    (4) $\neg p \vee \neg q$

  - *generamos los siguientes resolventes*

    (5) q *de* (1) *y* (2)
    (6) $\neg q$ *de* (3) *y* (4)
    (7) $\square$ *de* (5) *y* (6)

  - *Por tanto,* Δ *es insatisfacible.*

- Arbol de deducción para la refutación del Ejemplo 29.



3.6.2. Sustituciones.

- **Definición 3.6.4 (Sustitución)**

$$\begin{array}{rccl} \sigma: & \mathcal{V} & \longrightarrow & \mathcal{T} \\ & X & \hookrightarrow & \sigma(X) \end{array}$$

- Es habitual representar las sustituciones como conjuntos finitos de la forma

$$\{X_1/t_1, X_2/t_2, \ldots, X_n/t_n\}$$

- para cada elemento X_i/t_i de la sustitución, t_i es un término diferente de X_i.

- $\{X_1, X_2, \ldots, X_n\}$ es el *dominio* ($\mathcal{D}om(\sigma)$).

- $\{t_1, t_2, \ldots, t_n\}$ es el *rango* ($\mathcal{R}an(\sigma)$).

- Con esta notación, la *sustitución identidad*, *id*, se representa mediante el conjunto vacío de elementos: $\{\}$ (*sustitución vacía*).

El Principio de Resolución de Robinson.

- Una sustitución donde los términos t_i son básicos se denomina *sustitución básica*.

- **Ejemplo 30** *Ejemplos de sustituciones:*
  - $\theta_1 = \{X/f(Z), Z/Y\}$;
  - $\theta_2 = \{X/a, Y/g(Y), Z/f(g(b))\}$.

- **Definición 3.6.5 (instancia)**
  - *La aplicación de una sustitución*
    $$\sigma = \{X_1/t_1, X_2/t_2, \ldots, X_n/t_n\}$$
    a una expresión $\mathcal{E}$, denotado $\sigma(\mathcal{E})$, se obtiene reemplazando **simultaneamente** *cada ocurrencia de X_i en la expresión $\mathcal{E}$ por el correspondiente término t_i.*
  - *Se dice que $\sigma(\mathcal{E})$ es una* instancia *de $\mathcal{E}$.*

- **Definición 3.6.6 ((pre)orden de máxima generalidad)**

  Sean $\mathcal{E}_1$ *y* $\mathcal{E}_2$ *expresiones de* $\mathcal{L}$. $\mathcal{E}_1$ *es* más general *que* $\mathcal{E}_2$, *denotado* $\mathcal{E}_1 \leq \mathcal{E}_2$, *sii*

  - *existe una sustitución* σ *tal que* $\sigma(\mathcal{E}_1) = \mathcal{E}_2$.

- **Ejemplo 31**
  $\mathcal{E} \equiv p(X, Y, f(b))$ *y* $\theta = \{Y/X, X/b\}$.

  - $\theta(\mathcal{E}) = p(b, X, f(b))$.

  - *El término* $p(b, X, f(b))$ *es una instancia del término* $p(X, Y, f(b))$.

  - $p(X, Y, f(b)) \leq p(b, X, f(b))$.

- En programación lógica es habitual denotar la aplicación de una sustitución σ a una expresión $\mathcal{E}$ mediante $\mathcal{E}\sigma$ en lugar de por $\sigma(\mathcal{E})$.

- **Definición 3.6.7 (Composición de Sustituciones)**

  Dadas dos sustituciones σ y θ, la composición de σ y θ es

  - *la aplicación $\sigma \circ \theta$ tal que $(\sigma \circ \theta)(\mathcal{E}) = \sigma(\theta(\mathcal{E}))$.*

- Propiedades de la composición de sustituciones:

  - (Asociativa) para toda sustitución ρ, σ y θ, se cumple que $(\rho \circ \sigma) \circ \theta = \rho \circ (\sigma \circ \theta)$.

  - (Existencia de elemento neutro) para toda sustitución θ, se cumple que $id \circ \theta = \theta \circ id = \theta$.

El Principio de Resolución de Robinson.

- Composición sustituciones y notación conjuntista.

Algoritmo 6

Entrada: *Dos sustituciones* $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ *y* $\sigma = \{Y_1/s_1, \ldots, Y_m/s_m\}$.

Salida: *La sustitución compuesta* $\sigma \circ \theta$.

Comienzo

Computar:

1. *La sustitución* $\theta_1 = \{X_1/\sigma(t_1), \ldots, X_n/\sigma(t_n)\}$.
2. *La sustitución* θ_2: *eliminando los elementos* $X_i/\sigma(t_i)$, *con* $X_i \equiv \sigma(t_i)$, *que aparecen en* θ_1.
3. *La sustitución* σ_2: *eliminando los elementos* Y_i/s_i, *con* $Y_i \in \mathcal{D}om(\theta)$, *que aparecen en* σ.

Devolver $\sigma \circ \theta = \theta_2 \cup \sigma_2$.

Fin

- **Ejemplo 32**
  $\theta = \{X/f(Y), Y/Z\}$ *y* $\sigma = \{X/a, Y/b,\ Z/Y\}$
  Aplicando el Algoritmo 6 obtenemos:

  (1) $\theta_1 = \{X/\sigma(f(Y)), Y/\sigma(Z)\} = \{X/f(b), Y/Y\}$
  (2) $\theta_2 = \{X/f(b)\}$
  (3) $\sigma_2 = \{Z/Y\}$

  Así pues,

  $$\sigma \circ \theta = \theta_2 \cup \sigma_2 = \{X/f(b), Z/Y\}.$$

  Dado el término $t \equiv h(X, Y, Z)$,

  $$(\sigma \circ \theta)(t) = h(f(b), Y, Y)$$

- EJERCICIO: Comprobar que este resultado coincide con el obtenido al aplicar la Definición 3.6.7.

- Si $\mathcal{V}ar(\sigma) \cap \mathcal{V}ar(\theta) = \emptyset$ entonces $\sigma \circ \theta = \sigma \cup \theta$.

El Principio de Resolución de Robinson.

- **Definición 3.6.8 (Sustitución Idempotente)**
  Una sustitución σ *se dice* idempotente *sii* $\sigma \circ \sigma = \sigma$.

- **Ejemplo 33**
  $\theta_1 = \{X/f(Z), Z/Y\}$ *y* $\theta_2 = \{X/a, Y/g(Y), Z/f(g(b))\}$ *no son idempotentes:*
  - $\theta_1 \circ \theta_1 = \{X/f(Y), Z/Y\}$
  - $\theta_2 \circ \theta_2 = \{X/a, Y/g(g(Y)), Z/f(g(b))\}$.

- EJERCICIO: Una substitución σ es idempotente si $\mathcal{D}om(\sigma) \cap \mathcal{V}ar(\mathcal{R}an(\sigma)) = \emptyset$.

- **Importante**: El principio de resolución solamente computa sustituciones idempotentes.

- **Definición 3.6.9 (Renombramiento)**
  Una sustitución ρ *se denomina* sustitución de renombramiento *o, simplemente,* renombramiento*, si*
  - *existe la sustitución inversa* ρ^{-1} *tal que* $\rho \circ \rho^{-1} = \rho^{-1} \circ \rho = id$.

El Principio de Resolución de Robinson.

- **Definición 3.6.10 (Variante)**
  Dadas dos expresiones $\mathcal{E}_1$ y $\mathcal{E}_2$, decimos que son variantes *si*
  - *existen dos renombramientos σ y θ, tales que*
    $$\mathcal{E}_1 = \sigma(\mathcal{E}_2) \text{ y } \mathcal{E}_2 = \theta(\mathcal{E}_1).$$

- **Ejemplo 34**
  $t_1 \equiv p(f(X_1, X_2), g(X_3), a)$ es una variante de $t_2 \equiv p(f(X_2, X_1), g(X_4), a)$:
  - *existen las sustituciones*
    $$\sigma = \{X_1/X_2, X_2/X_1, X_4/X_3, X_3/X_4\}$$
    y
    $$\theta = \{X_1/X_2, X_2/X_1, X_3/X_4, X_4/X_3\}$$
    tales que
    $$t_1 = \sigma(t_2) \text{ y } t_2 = \theta(t_1).$$

- **Definición 3.6.11 ((pre)orden de máxima generalidad)**

  Dadas dos sustituciones σ y θ. Decimos que σ es más general *que θ, denotado $\sigma \leq \theta$, sii*
  - *existe una sustitución λ tal que $\theta = \lambda \circ \sigma$.*

- **Ejemplo 35**
  $\sigma = \{X/a\}$ *y* $\theta = \{X/a, Y/b\}$.
  - *Existe $\lambda = \{Y/b\}$ tal que $\theta = \lambda \circ \sigma \Longrightarrow \sigma \leq \theta$.*

- **Definición 3.6.12** *Dadas dos sustituciones σ y θ. Decimos que σ es* equivalente *a θ, denotado $\sigma \sim \theta$, sii*

  $\sigma \leq \theta$ *y* $\theta \leq \sigma$.

- EJERCICIO: $\sigma \sim \theta$ sii existe una sustitución de renombramiento ρ tal que $\theta = \rho \circ \sigma$.

- EJERCICIO: dos sustituciones son variantes una de otra sii son equivalentes.

3.6.3. Unificación.

- **Definición 3.6.13 (Unificador)**

  - *Una sustitución θ es un* unificador *del conjunto de expresiones*

    $\{\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_k\}$

    sii

    $\theta(\mathcal{E}_1) = \theta(\mathcal{E}_2) = \ldots = \theta(\mathcal{E}_k)$.

  - *El conjunto $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_k$ es* unificable *si existe un unificador para él.*

- **Ejemplo 36**
  $\{X/a\}$ es un unificador del conjunto $\{p(X, b), p(a, b)\}$.

El Principio de Resolución de Robinson.

- **Definición 3.6.14 (Unificador Más General (u.m.g.))**

  Un unificador σ de un conjunto de expresiones es un unificador más general *(o de máxima generalidad) sii*

  - *para cualquier unificador θ del conjunto, $\sigma \leq \theta$.*

- **Ejemplo 37**
  $E = \{p(f(Y), Z), p(X, a)\}$.

  - $\{X/f(a), Z/a, Y/a\}$ *es un unificador del conjunto E.*

  - $\{X/f(Y), Z/a, W/a\}$ *es un unificador del conjunto E.*

  - $\{X/f(Y), Z/a\}$ *es un unificador del conjunto E.*

  Sólo la última es un u.m.g. de E.

El Principio de Resolución de Robinson.

- **Algoritmo de unificación sintáctica** de Martelli y Montanari.
  - la unificación de dos expresiones $\mathcal{E}_1 \equiv F(t_1, \ldots, t_n)$ y $\mathcal{E}_2 \equiv F(s_1, \ldots, s_n)$ es una secuencia de transformaciones

    $$\langle G, id\rangle \Rightarrow \langle G1, \theta_1\rangle \Rightarrow \langle G2, \theta_2\rangle \Rightarrow \ldots \Rightarrow \langle G_n, \theta_n\rangle.$$

  - $\langle G, id\rangle$ es el *estado inicial*, donde $G = \{t_1 = s_1, \ldots, t_n = s_n\}$ es un conjunto de ecuaciones e id es la sustitución identidad.

  - Si $\langle G_n, \theta_n\rangle \equiv \langle \emptyset, \theta\rangle$, diremos que $\mathcal{E}_1$ y $\mathcal{E}_2$ son unificables con u.m.g. θ (denotado, $mgu(\mathcal{E}_1, \mathcal{E}_2)$).

  - Si $\langle G_n, \theta_n\rangle \equiv \langle Fallo, \theta\rangle$ diremos que $\mathcal{E}_1$ y $\mathcal{E}_2$ no son unificables y termina con un *estado de fallo*.

- El estado $\langle \emptyset, \theta\rangle$ simboliza el éxito en el proceso de unificación (el que el conjunto inicial G se ha reducido al conjunto vacío).

- El estado $\langle Fallo, \theta\rangle$ simboliza el fracaso.

- **Definición 3.6.15 (Reglas-MM y relación "$\Rightarrow$")**
  La relación de unificación "$\Rightarrow$" es la mínima relación definida por el conjunto de reglas de transición:

  1. `Descomposición en términos`*:*
  $$\frac{\langle \{P(t_1,\ldots,t_n) = P(s_1,\ldots,s_n)\} \cup E, \theta\rangle}{\langle \{t_1 = s_1,\ldots,t_n = s_n\} \cup E, \theta\rangle}$$
  donde P es tanto un símbolo de función como de relación.

  2. `Eliminación de ecuaciones triviales`*:*
  $$\frac{\langle \{X = X\} \cup E, \theta\rangle}{\langle E, \theta\rangle}$$

  3. `Intercambio`*:*
  $$\frac{\langle \{t = X\} \cup E, \theta\rangle}{\langle \{X = t\} \cup E, \theta\rangle}$$
  si t no es una variable.

  4. `Eliminación de variable`*:*
  $$\frac{\langle \{X = t\} \cup E, \theta\rangle}{\langle \{X/t\}(E), \{X/t\} \circ \theta\rangle}$$
  si la variable X no aparece en t.

  5. `Regla de fallo`*:*
  $$\frac{\langle \{P(t_1,\ldots,t_n) = Q(s_1,\ldots,s_n)\} \cup E, \theta\rangle}{\langle Fallo, \theta\rangle}$$

  6. `Ocurrencia de variable` *(*occur check*):*
  $$\frac{\langle \{X = t\} \cup E, \theta\rangle}{\langle Fallo, \theta\rangle}$$
  si la variable X aparece en t.

- **Ejemplo 38** *Para hallar el u.m.g. de las expresiones* $\mathcal{E}_1 \equiv p(a, X, f(g(Y)))$ *y* $\mathcal{E}_2 \equiv p(Z, f(Z), f(U))$*, procedemos como sigue:*

  1. *Construimos el estado o configuración inicial:*

  $$\langle \{a = Z, X = f(Z), f(g(Y)) = f(U)\}, id \rangle$$

  2. *Aplicamos las reglas MM hasta alcanzar una configuración de éxito o de fallo:*

  $$\begin{array}{ll} \langle \{a = Z, X = f(Z), f(g(Y)) = f(U)\}, id \rangle & \Rightarrow_3 \\ \langle \{Z = a, X = f(Z), f(g(Y)) = f(U)\}, id \rangle & \Rightarrow_4 \\ \langle \{X = f(a), f(g(Y)) = f(U)\}, \{Z/a\} \rangle & \Rightarrow_4 \\ \langle \{f(g(Y)) = f(U)\}, \{Z/a, X/f(a)\} \rangle & \Rightarrow_1 \\ \langle \{g(Y) = u\}, \{Z/a, X/f(a)\} \rangle & \Rightarrow_3 \\ \langle \{U = g(Y)\}, \{Z/a, X/f(a)\} \rangle & \Rightarrow_4 \\ \langle \emptyset, \{Z/a, X/f(a), U/g(Y)\} \rangle & \end{array}$$

  $mgu(\mathcal{E}_1, \mathcal{E}_2) = \{Z/a, X/f(a), U/g(Y)\}$.

- **Teorema 3.6.16 (Teorema de Unificación)** *Dadas dos expresiones* $\mathcal{E}_1$ *y* $\mathcal{E}_2$ *unificables, la secuencia transformaciones a partir del estado inicial*

  $$\langle G, id \rangle \Rightarrow \langle G1, \theta_1 \rangle \Rightarrow \langle G2, \theta_2 \rangle \Rightarrow \ldots \Rightarrow \langle \emptyset, \theta \rangle$$

  termina obteniendo el u.m.g. θ *de las expresiones* $\mathcal{E}_1$ *y* $\mathcal{E}_2$*, que es único salvo renombramientos de variables.*

3.6.4. El Principio de Resolución en la lógica de predicados.

- **Definición 3.6.17 (Factor de una Cláusula)**

  - *Si dos o más literales (con el mismo signo) de una cláusula $\mathcal{C}$ tienen un unificador de máxima generalidad σ, entonces se dice que $\sigma(\mathcal{C})$ es un* factor *de $\mathcal{C}$.*

  - *Si $\sigma(\mathcal{C})$ es una cláusula unitaria, entonces es un* factor unitario *de $\mathcal{C}$.*

- **Ejemplo 39**
  Sea $\mathcal{C} \equiv p(X) \vee p(f(Y)) \vee \neg q(X)$.

  - $\sigma = mgu(p(X), p(f(Y))) = \{X/f(Y)\}$.

  - $\sigma(\mathcal{C}) = p(f(Y)) \vee \neg q(f(Y))$ *es un factor de* $\mathcal{C}$.

- **Definición 3.6.18 (Resolvente Binario)**

$$\frac{L_1 \in \mathcal{C}_1, L_2 \in \mathcal{C}_2, \sigma = mgu(L_1, \neg L_2) \not\equiv fallo}{(\sigma(\mathcal{C}_1) \setminus \sigma(L_1)) \cup (\sigma(\mathcal{C}_2) \setminus \sigma(L_2))}$$

  - $\mathcal{C}_1$ *y* $\mathcal{C}_2$ *son las cláusulas padres.*

  - L_1 *y* L_2 *son los literales resueltos.*

  - $(\sigma(\mathcal{C}_1) \setminus \sigma(L_1)) \cup (\sigma(\mathcal{C}_2) \setminus \sigma(L_2))$ *es el resolvente binario de* $\mathcal{C}_1$ *y* $\mathcal{C}_2$.

- **Ejemplo 40**
  $\mathcal{C}_1 \equiv p(X) \vee q(X)$ *y* $\mathcal{C}_2 \equiv \neg p(a) \vee r(X)$.
  - *Como* X *aparece tanto en* $\mathcal{C}_1$ *como en* $\mathcal{C}_2$*, renombramos la variable de* $\mathcal{C}_2$ *y hacemos* $\mathcal{C}_2 \equiv \neg p(a) \vee r(Y)$.
  - $L_1 \equiv p(X)$ *y* $L_2 \equiv \neg p(a)$, $\sigma = mgu(L_1, \neg L_2) = \{X/a\}$.
  - *Así, pues:*
    $$\begin{aligned}&(\sigma(\mathcal{C}_1) \setminus \sigma(L_1)) \cup (\sigma(\mathcal{C}_2) \setminus \sigma(L_2))\\ &= (\{p(a), q(a)\} \setminus \{p(a)\}) \cup (\{\neg p(a), r(Y)\} \setminus \{\neg p(a)\})\\ &= \{q(a)\} \cup \{r(Y)\}\\ &= \{q(a), r(Y)\}\end{aligned}$$
  - $q(a) \vee r(Y)$ *es un resolvente binario de* $\mathcal{C}_1$ *y* $\mathcal{C}_2$.
  - $p(X)$ *y* $\neg p(a)$ *son los literales resueltos.*

- **Definición 3.6.19 (Resolvente)**
  Un resolvente *de cláusulas (padres)* $\mathcal{C}_1$ *y* $\mathcal{C}_2$ *es uno de los siguientes resolventes binarios:*
  1. *Un resolvente binario de* $\mathcal{C}_1$ *y* $\mathcal{C}_2$,
  2. *Un resolvente binario de* $\mathcal{C}_1$ *y un factor de* $\mathcal{C}_2$,
  3. *Un resolvente binario de un factor de* $\mathcal{C}_1$ *y* $\mathcal{C}_2$,
  4. *Un resolvente binario de un factor de* $\mathcal{C}_1$ *y un factor de* $\mathcal{C}_2$.

- **Ejemplo 41** *Sea*

  $\mathcal{C}_1 \equiv p(X) \vee p(f(Y)) \vee r(g(Y))$ *y* $\mathcal{C}_2 \equiv \neg p(f(g(a))) \vee q(b)$.

  - *Un factor de* $\mathcal{C}_1$ *es la cláusula*

    $\mathcal{C}_1' \equiv p(f(Y)) \vee r(g(Y))$.

  - *Un resolvente binario de* $\mathcal{C}_1'$ *y* $\mathcal{C}_2$ *es* $r(g(g(a))) \vee q(b)$.

  - *Por tanto* $r(g(g(a))) \vee q(b)$ *es un resolvente de* $\mathcal{C}_1$ *y* $\mathcal{C}_2$

- A la hora de aplicar un paso de resolución, un hecho importante es la existencia de un algoritmo de unificación que permite obtener el u.m.g. o de avisar de su inexistencia.

- La unificación de dos cláusulas puede verse como un método que calcula la particularización más adecuada que permite que pueda emplearse la regla de resolución.

El Principio de Resolución de Robinson.

- Existen paralelos entre la aplicación del principio de resolución y de la regla de eliminación del generalizador del sistema $\mathcal{K}_{\mathcal{L}}$.
  - Por ejemplo:

$$\begin{array}{lll} & \vdots & \\ (k) & (\forall X)(\mathcal{A}(X) \rightarrow \mathcal{B}(X)) & \\ (k+1) & \mathcal{A}(t) & \\ (k+2) & (\mathcal{A}(t) \rightarrow \mathcal{B}(t)) & \texttt{(EG)}\, k,\, \{X/t\} \\ (k+3) & \mathcal{B}(t) & \texttt{(MP)}\, k+1, k+2 \end{array}$$

  - Se han unificado las expresiones $\mathcal{A}(X)$ y $\mathcal{A}(t)$ para poder aplicar *modus ponens*.

- **Teorema 3.6.20** *Sea Δ un conjunto de cláusulas.*
  1. *(*`Corrección`*) Si existe $\Delta \vdash \square$ entonces, Δ es insatisfacible.*
  2. *(*`Completitud`*) Si Δ es insatisfacible entonces, existe $\Delta \vdash \square$.*

- **Observación 3.6.21**

  De la discusión mantenida en el Apartado 3.2 y el Teorema 3.6.20, se sigue que probar que $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n\} \models \mathcal{B}$, es equivalente a:
  1. *obtener el conjunto de cláusulas que representa la formula $(\mathcal{A}_1 \wedge \mathcal{A}_2 \wedge \ldots \wedge \mathcal{A}_n \wedge \neg\mathcal{B})$;*
  2. *obtener la cláusula vacía a partir de ese conjunto de cláusulas.*

- **Ejemplo 42**
  $\Delta = \{\neg p(f(X)) \vee s(Y), \neg s(X) \vee r(X,b), p(X) \vee s(X)\}$.
  Para probar que $(\exists X)(\exists Y) r(X,Y)$ *es una consecuencia lógica de* Δ*:*

  - *Negamos la conclusión y pasamos a forma clausal:* $\neg r(X,Y)$

  - *obtenemos la siguiente prueba aplicando resolución:*

(1)	$\neg p(f(X)) \vee s(Y)$	
(2)	$\neg s(X) \vee r(X,b)$	
(3)	$p(X) \vee s(X)$	
(4)	$\neg r(X,Y)$	*negación de la conclusión*
(5)	$s(f(X)) \vee s(Y)$	*de* (1) *y* (3)
(6)	$r(f(X),b)$	*de* (2) *y factor* $s(f(X))$ *de* (5)
(7)	$\square$	*de* (4) *y* (6)

  - *En la resolución de* (1) *y* (3)*:*
    $mgu(p(f(X)), p(X_1)) = \{X_1/f(X)\}$.

  - *En la resolución de* (2) *y el factor de* (5)*:*
    $mgu(s(X_2), s(f(X))) = \{X_2/f(X)\}$.

  - *En la resolución de* (4) *y* (6)*:*
    $mgu(r(X,Y), r(f(X_3),b)) = \{X/f(X_3), Y/b\}$.

- **Ejemplo 43**

  $\Delta = \{\neg s(X,Y) \vee \neg p(Y) \vee r(f(X)), \neg r(Z), s(a,b), p(b)\}$.

  - *Para probar que el conjunto de cláusulas Δ es insatisfacible procedemos del siguiente modo:*

    (1) $\neg s(X,Y) \vee \neg p(Y) \vee r(f(X))$
    (2) $\neg r(Z)$
    (3) $s(a,b)$
    (4) $p(b)$
    (5) $\neg s(X,Y) \vee \neg p(Y)$ *de* (1) *y* (2)
    (6) $\neg p(b)$ *de* (3) *y* (5)
    (7) $\square$ *de* (4) *y* (6)

  - *En la resolución de* (1) *y* (2)*: el*

    $mgu(r(f(X)), r(Z)) = \{Z/f(X)\}$.

  - *En la resolución de* (3) *y* (5)*:*

    $mgu(s(a,b), s(X,Y)) = \{X/a, Y/b\}$.

  - *En la resolución de* (4) *y* (6)*:*

    $mgu(p(b), p(b)) = id$.

- En caso de que un conjunto de cláusulas sea insatisfacible el Teorema 3.6.20 no nos dice como encontrar la cláusula vacía.

- Un esquema que permite sistematizar la búsqueda de la cláusula vacía.

Algoritmo 7 (Algoritmo de Resolución Genérico)

Entrada: *Un conjunto de cláusulas* Δ.
Salida: *Una condición de* **éxito**.
Comienzo
Inicialización: $CLAUSULAS = \Delta$.
Repetir
1. *Seleccionar dos cláusulas distintas,* $\mathcal{C}_i$ *y* $\mathcal{C}_j$, *del conjunto* $CLAUSULAS$, *que sean resolubles;*
2. *Hallar el resolvente* $\mathcal{R}_{ij}$ *de* $\mathcal{C}_i$ *y* $\mathcal{C}_j$*;*
3. $CLAUSULAS = CLAUSULAS \cup \{\mathcal{R}_{ij}\}$*;*

Hasta que $\square$ *aparezca en* $CLAUSULAS$.
Devolver **éxito**.
Fin

- Este algoritmo genera nuevas cláusulas a partir de las antiguas sin ninguna restricción.

- **Inconveniente**: Una aplicación sin restricciones del principio de resolución puede generar cláusulas que son redundantes o irrelevantes para los objetivos de la prueba.

Estrategias de Resolución.

- El Algoritmo 7 posee dos grados de libertad:
  1. ¿qué cláusulas seleccionar para realizar la resolución?.
  2. ¿qué literal $L_i \in \mathcal{C}_i$ y qué literal $L_j \in \mathcal{C}_j$ elegimos para realizar el paso de resolución que conduce al resolvente $\mathcal{R}_{ij}$?.

- Fijar estos grados de libertad da lugar a diferentes estrategias de resolución.

- Una **estrategia de resolución** es una instancia del Algoritmo 7.

- **Objetivo**: lograr eficiencia en el número de resoluciones empleadas y en la cantidad de almacenamiento empleado

- Es importante que una estrategia continue siendo completa. Esto es, que sea capaz de encontrar la cláusula vacía, $\square$, siempre que Δ sea insatisfacible.

- Estrategias y árboles de búsqueda.
  - Una estrategia de resolución puede verse como un mecanismo que restringe (disminuye) el árbol de búsqueda.

3.7.1. Estrategia de resolución por saturación de niveles.

- La *estrategia de resolución por saturación de niveles* esencialmente es una estrategia de búsqueda a ciegas.

- También se denomina estrategia a lo ancho (Nilsson).

- Construye una colección $\Delta^0, \Delta^1, \ldots$ de conjuntos de cláusulas, donde

$$\begin{aligned}\Delta^0 &= \Delta \\ \Delta^n &= \{\text{resolventes de } \mathcal{C}_i \text{ y } \mathcal{C}_j \mid \mathcal{C}_i \in (\Delta^0, \ldots, \Delta^{n-1}) \wedge \\ &\qquad \mathcal{C}_j \in \Delta^{n-1} \wedge \\ &\qquad \mathcal{C}_i \prec \mathcal{C}_j\}\end{aligned}$$

- En el Algoritmo 7, en cada iteración n:
  - Los pasos (1) y (2) se substituyen por la tarea de computar el conjunto Δ^n.
  - El paso 3 se substituyen por:

$$CLAUSULAS = CLAUSULAS \cup \Delta^n.$$

- La estrategia de resolución por saturación de niveles es completa pero sumamente ineficiente.

3.7.2. Estrategia de borrado.

- Se basa en la estrategia de resolución por saturación de niveles.
  - Si $\mathcal{R}_{ij} \in \Delta^n$ es una tautología o es una cláusula subsumida por una cláusula en $CLAUSULAS$, entonces se elimina del conjunto Δ^n.

- **Definición 3.7.1**

  - *Una cláusula* $\mathcal{C}$ subsume *a una cláusula* $\mathcal{C}'$ *sii existe una substitución* σ *tal que* $\sigma(\mathcal{C}) \subseteq \mathcal{C}'$.

  - $\mathcal{C}'$ *es la cláusula* subsumida.

- La estrategia de borrado es completa.

3.7.3. Estrategia de resolución semántica.

- Intenta limitar el número de resoluciones dividiendo en dos subconjuntos disjuntos el conjunto $CLAUSULAS$:
  - Idea: no puedan resolverse entre sí las cláusulas que pertenecen a un mismo subconjunto.

- La estrategia de resolución semántica se comporta de siguiente modo:
  1. Divide el conjunto $CLAUSULAS$ en dos conjunto utilizando una interpretación $\mathcal{I}$:
     $$\Delta_1 = \{\mathcal{C} \in CLAUSULAS \mid \mathcal{I} \text{ es modelo de } \mathcal{C}\}$$
     $$\Delta_2 = \{\mathcal{C} \in CLAUSULAS \mid \mathcal{I} \text{ no es modelo de } \mathcal{C}\}$$
  2. Halla el conjunto de todos los resolventes que pueden formarse a partir de las cláusulas de Δ_1 y Δ_2:
     $$\Delta' = \{\mathcal{R}_{ij} \mid \mathcal{R}_{ij} \text{ es un resolvente de } \mathcal{C}_i \in \Delta_1 \text{ y } \mathcal{C}_j \in \Delta_2\}$$
  3. $CLAUSULAS = CLAUSULAS \cup \Delta'$

- La estrategia de resolución semántica es completa.

Estrategias de Resolución.

- **Estrategia del conjunto de soporte**: es un caso particular de la estrategia de resolución semántica.

- $\Delta' \subseteq \Delta$ se denomina *conjunto de soporte* de Δ si $\Delta \setminus \Delta'$ es satisfacible.

- La estrategia del conjunto de soporte impide el cómputo de resolventes cuyas cláusulas padres pertenezcan, ambas, al subconjunto $\Delta \setminus \Delta'$.

- Justificación: del conjunto satisfacible $\Delta \setminus \Delta'$ no puede extraerse la cláusula vacía.

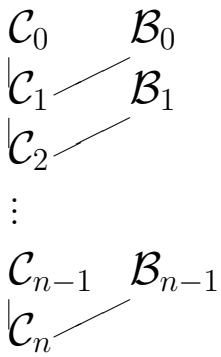
- La estrategia del conjunto de soporte es completa.

3.7.4. Estrategia de resolución lineal.

- **Definición 3.7.2 (Deducción Lineal)**
  Dado un conjunto de cláusulas Δ *y una cláusula* $\mathcal{C}_0$ *de* Δ, *una* deducción lineal *de* $\mathcal{C}_n$ *a partir de* Δ *con* cláusula en cabeza $\mathcal{C}_0$ *es una deducción de la forma mostrada en la Figura, donde:*

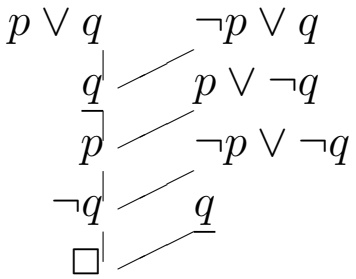
  1. *Para* $i = 0, 1, \ldots, n-1$, $\mathcal{C}_{i+1}$ *es un resolvente de* $\mathcal{C}_i$ *(llamada* cláusula central*) y* $\mathcal{B}_i$ *(llamada* cláusula lateral*), y*

  2. *Cada* $\mathcal{B}_i$ *pertenece a* Δ, *o es una cláusula central* $\mathcal{C}_j$ *para algún* j, *con* $j < i$.



Estrategias de Resolución.

- **Ejemplo 44**
  $\Delta = \{(p \vee q), (\neg p \vee q), (p \vee \neg q), (\neg p \vee \neg q)\}$.
  - *La Figura muestra una deducción lineal de* $\square$ *a partir de* Δ *con cláusula en cabeza* $(p \vee q)$.
  - *Se observan cuatro cláusulas laterales:*
    - *Tres de ellas pertenecen a* Δ.
    - *La última, "q", es un resolvente.*

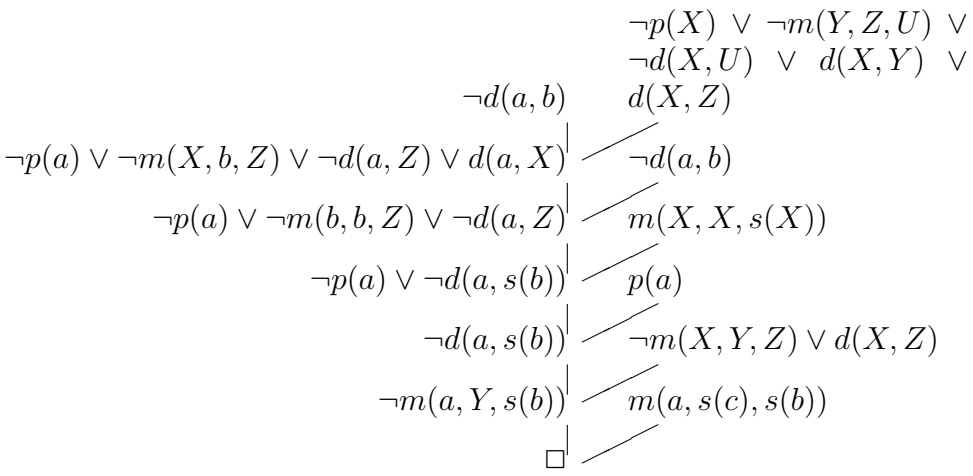


Estrategias de Resolución.

- **Ejemplo 45** *Consideremos el conjunto de cláusulas*

$$\begin{aligned}\Delta = \{&m(a, s(c),\ s(b)),\ p(a),\ m(X, X, s(X)),\\ &(\neg m(X, Y, Z) \vee m(Y, X, Z)), (\neg m(X, Y, Z) \vee d(X, Z)),\\ &(\neg p(X) \vee \neg m(Y, Z, U) \vee \neg d(X, U) \vee d(X, Y) \vee d(X, Z)),\\ &\neg d(a, b)\}.\end{aligned}$$

La Figura es una deducción lineal de $\square$ *a partir de* Δ.



Estrategias de Resolución.

- La estrategia de resolución lineal es completa

- **Estrategia de resolución lineal de entrada.** (*input resolution*)
  - Una de las cláusulas padres seleccionada tiene que pertenecer al conjunto de cláusulas inicial Δ.

  - La estrategia de resolución lineal de entrada no es completa.

Capítulo 4

Programación Lógica.

DAT y Programación Lógica (PL).

- Similitudes.
  1. El empleo de un lenguaje basado en la forma cláusal.
  2. El tipo de regla de inferencia empleado: el principio de resolución.
  3. Compartir ciertos aspectos del empleo de estrategias (estrategia del conjunto de soporte).
  4. El empleo de pruebas por contradicción.

DAT y Programación Lógica (PL).

- Diferencias.
  1. La PL sólo usa cláusulas de Horn. En la DAT no se impone ninguna restricción sobre el tipo de cláusulas.
  2. En la PL no hay retención de información intermedia, mientras que para la DAT ésto es esencial para conseguir buenos resultados.
     (No retener información intermedia permite implementación eficiente)
  3. La PL hace un uso muy limitado de las estrategias.
  4. La PL destierra el uso de la igualdad, mientras que en la implementación eficiente el tratamiento de la igualdad es uno de los puntos claves.
  5. Ambos campos poseen objetivos diferentes.
     La DAT $\Longrightarrow$ solución de problemas sin algoritmo efectivo conocido.
     La PL $\Longrightarrow$ lenguaje de programación.

- **Objetivo.** Caracterizar un lenguaje de programación lógica como sistema formal.

4.1. Sintaxis.

4.1.1. Notación para las cláusulas.

- Una cláusula no vacía puede escribirse de la siguiente forma:

  $$(\forall X_1), \ldots, (\forall X_s)(\mathcal{M}_1 \vee \ldots \vee \mathcal{M}_m \vee \neg\mathcal{N}_1 \vee \ldots \vee \neg\mathcal{N}_n)$$

  donde $\mathcal{M}_i$, $i = 1, \ldots, m$, y $\mathcal{N}_j$, $j = 1, \ldots, n$, representan átomos.

- Mediante una sencilla manipulación se obtiene:

  $$(\forall X_1), \ldots, (\forall X_s)(\mathcal{M}_1 \vee \ldots \vee \mathcal{M}_m) \leftarrow (\mathcal{N}_1 \wedge \ldots \wedge \mathcal{N}_n)$$

  donde el símbolo "$\leftarrow$" se denomina *condicional recíproco*.

- *Visión operacional* (*procedimental*):

  Para resolver el problema $\mathcal{M}_1$ o el $\mathcal{M}_2$ o ... o el $\mathcal{M}_m$, hay que resolver los problemas $\mathcal{N}_1$ y $\mathcal{N}_2$ y ... y $\mathcal{N}_n$.

  Aquí, $\mathcal{N}_1, \ldots, \mathcal{N}_n$ se consideran como procedimientos.

- *Visión declarativa*:

  si se cumple $\mathcal{N}_1$ y $\mathcal{N}_2$ y ... y $\mathcal{N}_n$ entonces debe cumplirse $\mathcal{M}_1$ o $\mathcal{M}_2$ o ... o $\mathcal{M}_m$.

Sintaxis.

- **Definición 4.1.1 (Cláusula General)** *Una* cláusula general *es una fórmula de la forma*

$$\mathcal{A}_1 \vee \ldots \vee \mathcal{A}_m \leftarrow L_1 \wedge \ldots \wedge L_n$$

donde

  - $\mathcal{A}_1 \vee \ldots \vee \mathcal{A}_m$ *es la* cabeza *(o* conclusión *o* consecuente*) de la cláusula general y*
  - $L_1 \wedge \ldots \wedge L_n$ *es el* cuerpo *(o* condición *o* antecedente*).*
  - *Cada* $\mathcal{A}_i$, $i = 1, \ldots, m$, *es un átomo y cada* L_j, $j = 1, \ldots, n$, *es un literal (átomo positivo o negativo).*
  - *Todas las variables se suponen universalmente cuantizadas.*
  - *Si* $m > 1$ *la cláusula general se dice que es* indefinida*.*
  - *Si* $m = 0$ *la cláusula general se dice que es un*objetivo*.*

- Nos referiremos a la notación que hemos introducido en este apartado como *notación clausal*.

Sintaxis.

- Clasificación de los diferentes tipos de cláusulas.

Forma	(m, n)	**Denominación**
$\mathcal{A} \leftarrow$	$(m = 1, n = 0)$	Cláusula unitaria; aserto; hecho.
$\mathcal{A}_1 \vee \ldots \vee \mathcal{A}_m \leftarrow$	$(m \geq 1, n = 0)$	Cláusula positiva; aserto indefinido.
$\mathcal{A} \leftarrow \mathcal{B}_1 \wedge \ldots \wedge \mathcal{B}_n$	$(m = 1, n \geq 0)$	Cláusula definida; cláusula de Horn.
$\mathcal{A} \leftarrow L_1 \wedge \ldots \wedge L_n$	$(m = 1, n \geq 0)$	Cláusula normal.
$\mathcal{A}_1 \vee \ldots \vee \mathcal{A}_m \leftarrow \mathcal{B}_1 \wedge \ldots \wedge \mathcal{B}_n$	$(m \geq 1, n \geq 0)$	Cláusula disyuntiva.
$\mathcal{A}_1 \vee \ldots \vee \mathcal{A}_m \leftarrow L_1 \wedge \ldots \wedge L_n$	$(m \geq 1, n \geq 0)$	Cláusula general.
$\leftarrow \mathcal{B}_1 \wedge \ldots \wedge \mathcal{B}_n$	$(m = 0, n \geq 1)$	Objetivo definido.
$\leftarrow L_1 \wedge \ldots \wedge L_n$	$(m = 0, n \geq 1)$	Objetivo normal.
$\leftarrow$	$(m = 0, n = 0)$	Cláusula vacía $\Box$.

Los $\mathcal{A}_i$'s y los $\mathcal{B}_j$'s representan átomos,
mientras que los L_k's representan literales.

4.1.2. Cláusulas de Horn y Programas Definidos.

- La programación logica deja de lado las cláusulas generales para ocuparse de las cláusulas de Horn (y de las cláusulas normales).

- **Definición 4.1.2 (Cláusula de Horn)** *Una* cláusula de Horn *(o* definida*) es una disyunción de literales de los cuales uno, como mucho, es positivo.*

- **Ejemplo 46** *Diferentes tipos de Cláusulas de Horn:*

$$\begin{array}{lll} hermanas(X,Y) \leftarrow & mujer(X) \wedge mujer(Y) \wedge & \\ & padres(X,P,M) \wedge padres(Y,P,M) & \textit{regla.} \\ mujer(ana) \leftarrow & & \textit{hecho.} \\ \leftarrow hermanas(X,Ana) & & \textit{objetivo.} \end{array}$$

- Algunos hechos interesantes:
  1. Las cláusulas de Horn tienen el mismo poder de cómputo que las máquinas de Turing o el λ-cálculo.
  2. Muchos de métodos de resolución de problemas pueden expresarse mediante cláusulas de Horn [Kowalski 79].

Sintaxis.

- **Definición 4.1.3 (Programa Definido)** *Un* programa definido *es un conjunto de cláusulas definidas* Π. *La* definición de un símbolo de predicado p *que aparece en* Π *es el subconjunto de cláusulas de* Π *en cuyas cabezas aparece el predicado* p.

- **Ejemplo 47**

$$\begin{aligned}
&\mathcal{C}_1: \ p(X,Y) \leftarrow q(X,Y)\\
&\mathcal{C}_2: \ p(X,Y) \leftarrow q(X,Z) \wedge p(Z,Y)\\
&\mathcal{C}_3: \ q(a,b)\\
&\mathcal{C}_4: \ q(b,c)
\end{aligned}$$

  - *Las cláusulas* $\mathcal{C}_1$ *y* $\mathcal{C}_2$ *definen el predicado* p.

  - *Las cláusulas* $\mathcal{C}_3$ *y* $\mathcal{C}_4$ *definen el predicado* q.

4.2. Semántica Operacional.

- Resolución SLD [Kowalski 74]: Es un refinamiento del procedimiento de refutación por resolución de Robinson.

- "SLD" $\Longrightarrow$ "resolución **L**ineal con función de **S**elección para programas **D**efinidos".

- Otras denominaciones:
  - Resolución LUSH $\Longrightarrow$ "*Linear resolution with Unrestricted Selection fuction for Horn clauses*".
  - Inferencia analítica (*top-down inference*) [Kowalski 79].

4.2.1. Procedimientos de prueba y objetivos definidos.

- La programación lógica intenta comprobar

  $$\Pi \vdash \mathcal{B}$$

  donde Π es un programa y $\mathcal{B} \equiv \exists(\mathcal{B}_1 \wedge \ldots \wedge \mathcal{B}_n)$.

- Equivalentemente, $\Pi \cup \{\neg\mathcal{B}\} \vdash \square$.

- Nótese que

  $$\begin{aligned} \neg\mathcal{B} &\Leftrightarrow \neg\exists(\mathcal{B}_1 \wedge \ldots \wedge \mathcal{B}_n) \\ &\Leftrightarrow \forall\neg(\mathcal{B}_1 \wedge \ldots \wedge \mathcal{B}_n) \\ &\Leftrightarrow \forall(\neg\mathcal{B}_1 \vee \ldots \vee \neg\mathcal{B}_n) \end{aligned}$$

- La fórmula $\forall(\neg\mathcal{B}_1 \vee \ldots \vee \neg\mathcal{B}_n)$ es una cláusula objetivo.

- en notación clausal se expresa como $\mathcal{G} \equiv \ \leftarrow (\mathcal{B}_1 \wedge \ldots \wedge \mathcal{B}_n)$

- Por consiguiente, cuando planteamos un objetivo $\mathcal{G}$ a un programa Π estamos respondiendo a la pregunta de si

  $$\exists(\mathcal{B}_1 \wedge \ldots \wedge \mathcal{B}_n)$$

  se sigue del programa Π.

4.2.2. Resolución SLD y respuesta computada.

- La estrategia de resolución SLD es un caso particular de resolución lineal (de entrada).

- Para probar la inconsistencia de $\Pi \cup \{\mathcal{G}\}$:
  1. Partimos del objetivo $\mathcal{G}$, que tomamos como cláusula en cabeza.
  2. Las clausulas laterales solamente pueden ser cláusulas del programa Π.
  3. Las cláusulas centrales son objetivos.
  4. Basta con seleccionar uno de los literales del objetivo que se está resolviendo en cada momento.
  5. No se emplea factorización ni tampoco resolución con ancestros.

Semántica Operacional.

- **Definición 4.2.1 (Regla de Computación)** *Llamamos* regla de computación *(o función de selección)* φ *a una función que, cuando se aplica a un objetivo* $\mathcal{G}$*, selecciona uno y sólo uno de los átomos de* $\mathcal{G}$*.*

- **Definición 4.2.2 (Paso de Resolución SLD)** *Dado un objetivo* $\mathcal{G} \equiv \leftarrow \mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_j \wedge \ldots \wedge \mathcal{A}_n$ *y una cláusula* $\mathcal{C} \equiv \mathcal{A} \leftarrow \mathcal{B}_1 \wedge \ldots \wedge \mathcal{B}_m$ *entonces* $\mathcal{G}'$ *es un* resolvente *de* $\mathcal{G}$ *y* $\mathcal{C}$ *(por resolución SLD)* usando la regla de computación φ *si se cumple que:*

  1. $\mathcal{A}_j = \varphi(\mathcal{G})$ *es el átomo de* $\mathcal{G}$ *seleccionado por* φ*;*
  2. $\theta = mgu(\mathcal{A}, \mathcal{A}_j)$*;*
  3. $\mathcal{G}' \equiv \leftarrow \theta(\mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_{j-1} \wedge \mathcal{B}_1 \wedge \ldots \wedge \mathcal{B}_m \wedge \mathcal{A}_{j+1} \wedge \ldots \wedge \mathcal{A}_n)$

- También diremos que $\mathcal{G}'$ se *deriva* de $\mathcal{G}$ y $\mathcal{C}$ (en un paso), en símbolos:

  - $\mathcal{G} \Longrightarrow_{SLD} \mathcal{G}'$; $\mathcal{G} \overset{\theta}{\Longrightarrow}_{SLD} \mathcal{G}'$; $\mathcal{G} \overset{[\mathcal{C},\theta]}{\Longrightarrow}_{SLD} \mathcal{G}'$

  según el grado de información que sea útil mantener.

- **Definición 4.2.3 (Derivación SLD)** *Sea* Π *un programa definido y* $\mathcal{G}$ *un objetivo definido. Una* derivación *SLD para* $\Pi \cup \{\mathcal{G}\}$*, usando la regla de computación* φ*, consiste en una secuencia de pasos de resolución SLD*

$$\mathcal{G}_0 \overset{[\mathcal{C}_1,\theta_1]}{\Longrightarrow}_{SLD} \mathcal{G}_1 \overset{[\mathcal{C}_2,\theta_2]}{\Longrightarrow}_{SLD} \cdots \overset{[\mathcal{C}_{n-1},\theta_{n-1}]}{\Longrightarrow}_{SLD} \mathcal{G}_{n-1} \overset{[\mathcal{C}_n,\theta_n]}{\Longrightarrow}_{SLD} \mathcal{G}_n$$

donde:

1. *Para* $i = 1, \ldots, n$*,* $\mathcal{G}_i$ *es un* resolvente *de* $\mathcal{G}_{i-1}$ *y* $\mathcal{C}_i$ *(por resolución SLD) usando la regla de computación* φ*,*

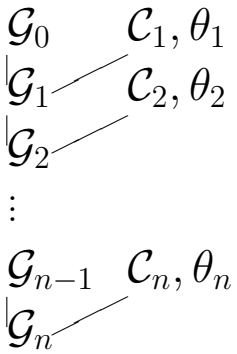
2. *Cada* $\mathcal{C}_i$ *es una variante de una cláusula* Π*, y*

3. *Cada* θ_i *es el mgu obtenido en el correspondiente paso de resolución SLD.*

Si $\mathcal{G}_n$ *es la última cláusula de la secuencia de resolventes, decimos que la derivación SLD tiene* longitud n*.*

Semántica Operacional.

- Arbol de deducción para una derivación SLD genérica:



- Estandarización de variables:
  - Las variables de la cláusulas $\mathcal{C}_i$ se renombran de forma que no haya conflicto con ninguna de las variables que hayan aparecido previamente.

- En ocasiones utilizaremos la notación $\Pi \cup \{\mathcal{G}\} \vdash_{SLD} \mathcal{G}_n$ para representar una derivación SLD para $\Pi \cup \{\mathcal{G}\}$.

- **Definición 4.2.4 (Refutación SLD)** *Sea* Π *un programa definido y* $\mathcal{G}$ *un objetivo definido. Una* refutación *SLD para* $\Pi \cup \{\mathcal{G}\}$ *es una derivación SLD finita cuyo último objetivo es la cláusula vacía.*

Semántica Operacional.

- Tipos de derivaciones SLD:
  - `Infinita`.
  - `Finita`:
    - De fallo: ninguna cláusula de Π unifica con el átomo $\mathcal{A}_j$ de $\mathcal{G}_n$ seleccionado.
    - De éxito: Si $\mathcal{G}_n = \square$.

- **Ejemplo 48**
  $\Pi = \{p(f(X')) \leftarrow p(X)\}$ *y* $\mathcal{G} \equiv \leftarrow p(X)$,

  $$\leftarrow p(X) \overset{\{X/f(X_1)\}}{\Longrightarrow_{SLD}} \leftarrow p(X_1) \overset{\{X_1/f(X_2)\}}{\Longrightarrow_{SLD}} \leftarrow p(X_2) \overset{\{X_2/f(X_3)\}}{\Longrightarrow_{SLD}} \ldots$$

  es una derivación SLD infinita para $\Pi \cup \{\mathcal{G}\}$.

- **Ejemplo 49**
  $\Pi = \{p(0) \leftarrow q(X)\}$ *y* $\mathcal{G} \equiv \leftarrow p(Z)$,

  $$\leftarrow p(Z) \overset{\{Z/0\}}{\Longrightarrow_{SLD}} \leftarrow q(X) \Longrightarrow_{SLD} fallo$$

  es una derivación SLD de fallo para $\Pi \cup \{\mathcal{G}\}$.

- **Ejemplo 50**
  $\Pi = \{\mathcal{C}_1 : p(0) \leftarrow q(X), \quad \mathcal{C}_2 : q(1) \leftarrow\}$ *y* $\mathcal{G} \equiv \leftarrow p(Z)$,

  $$\leftarrow p(Z) \overset{[\mathcal{C}_1,\{Z/0\}]}{\Longrightarrow_{SLD}} \leftarrow q(X) \overset{[\mathcal{C}_2,\{X/1\}]}{\Longrightarrow_{SLD}} \square$$

  es una derivación SLD de éxito para $\Pi \cup \{\mathcal{G}\}$.

Semántica Operacional.

- Un objetivo primordial de la programación lógica es computar. El efecto de computar se consigue devolviendo una sustitución

- **Definición 4.2.5 (Respuesta Computada)** *Sea* Π *un programa definido y* $\mathcal{G}$ *un objetivo definido. Sea*

  $$\mathcal{G} \overset{\theta_1}{\Longrightarrow}_{SLD} \mathcal{G}_1 \overset{\theta_2}{\Longrightarrow}_{SLD} \ldots \overset{\theta_n}{\Longrightarrow}_{SLD} \square$$

  una refutación para $\Pi \cup \{\mathcal{G}\}$.

  - *Una* respuesta computada *para* $\Pi \cup \{\mathcal{G}\}$ *es la sustitución*

    $$\theta = (\theta_n \circ \ldots \circ \theta_1)_{\restriction (\mathcal{V}ar(\mathcal{G})}.$$

  - *Si* $\mathcal{G} \equiv \leftarrow \mathcal{Q}$, *se dice que* $\theta(\mathcal{Q})$ *es una* instancia computada *de* $\mathcal{G}$.

- Si $\mathcal{G}_0 \overset{\theta_1}{\Longrightarrow}_{SLD} \mathcal{G}_1 \overset{\theta_2}{\Longrightarrow}_{SLD} \ldots \overset{\theta_n}{\Longrightarrow}_{SLD} \mathcal{G}_n$, en ocasiones escribiremos

  $$\mathcal{G} \overset{\theta}{\Longrightarrow}_{SLD}{}^{*} \mathcal{G}_n,$$

  donde $\theta = \theta_n \circ \ldots \circ \theta_1$. Decimos que $\mathcal{G}_n$ se deriva (por resolución SLD) en estrella pasos a partir de $\mathcal{G}_0$.

- **Ejemplo 51**
  $\Pi = \{\mathcal{C}_1 : p(a, X) \leftarrow q(X), \quad \mathcal{C}_2 : q(W) \leftarrow\}$ y $\mathcal{G} \equiv\leftarrow p(Y, b)$,

  $$\leftarrow p(Y, b) \overset{[\mathcal{C}_1, \{Y/a, X/b\}]}{\Longrightarrow_{SLD}} \leftarrow q(b) \overset{[\mathcal{C}_2, \{W/b\}]}{\Longrightarrow_{SLD}} \square$$

  - $\theta = \{Y/a, X/b, W/b\}_{\restriction(\mathcal{V}ar(\mathcal{G})} = \{Y/a\}$.
  - $\theta(p(Y, b)) = p(a, b)$ *es la instancia computada de* $\mathcal{G}$.

- `Sistema de transición de estados y resolución SLD`:
  - Concepto de estado E: un par formado por una cláusula objetivo y una sustitución.
  - *Resolución SLD* (usando la regla de computación φ): sistema de transición cuya relación de transición $\Longrightarrow_{SLD} \subseteq (E \times E)$ es la relación más pequeña que satisface:

    $$\frac{(\mathcal{G} \equiv\leftarrow \mathcal{Q}_1 \wedge \mathcal{A}' \wedge \mathcal{Q}_2), \varphi(\mathcal{G}) = \mathcal{A}', (\mathcal{C} \equiv (\mathcal{A} \leftarrow \mathcal{Q}) \ll \Pi, \sigma = mgu(\mathcal{A}, \mathcal{A}')}{\langle \mathcal{G}, \theta \rangle \Longrightarrow_{SLD} \langle \leftarrow \sigma(\mathcal{Q}_1 \wedge \mathcal{Q} \wedge \mathcal{Q}_2), \sigma \circ \theta \rangle}$$

  - $\mathcal{Q}$, $\mathcal{Q}_1$, $\mathcal{Q}_2$ representan conjunciones de átomos.
  - El símbolo "$\ll$" expresa que $\mathcal{C}$ es una cláusula de Π que se toma estandarizada aparte.
  - Derivación SLD:

    $$\langle \mathcal{G}_0, id \rangle \Longrightarrow_{SLD} \langle \mathcal{G}_1, \theta_1 \rangle \Longrightarrow_{SLD} \ldots \Longrightarrow_{SLD} \langle \mathcal{G}_n, \theta_n \rangle$$

  - Refutación SLD: $\langle \mathcal{G}_0, id \rangle \Longrightarrow_{SLD}{}^* \langle \square, \sigma \rangle$, donde σ es la respuesta computada en la refutación.

4.2.3. Arboles de búsqueda SLD y procedimientos de prueba.

- Un procedimiento de prueba por refutación puede entenderse como un proceso de búsqueda en un espacio de estados.

- Estados: los objetivos que se generan en cada paso de resolución.

- El espacio de búsqueda se puede representar como un árbol.

- **Definición 4.2.6 (Arbol de Búsqueda SLD)**
  Un árbol (de búsqueda) SLD τ_φ para $\Pi \cup \{\mathcal{G}\}$ (usando la regla de computación φ) es un conjunto de nodos que cumplen:

  1. *El nodo raíz de τ_φ es el objetivo inicial $\mathcal{G}$;*
  2. *Si $\mathcal{G}_i \equiv \leftarrow \mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_n$ es un nodo de τ_φ y supuesto que $\varphi(\mathcal{G}_i) = \mathcal{A}_k$, entonces para cada cláusula $\mathcal{C}_j \equiv \mathcal{A} \leftarrow \mathcal{Q}$ de Π (con sus variables renombradas si hace falta) t.q. $\theta = mgu(\mathcal{A}, \mathcal{A}_k) \not\equiv fail$, el resolvente*

     $$\mathcal{G}_{ij} \equiv \leftarrow \theta(\mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_{k-1} \wedge \mathcal{Q} \wedge \mathcal{A}_{k+1} \wedge \ldots \wedge \mathcal{A}_n)$$

     es un nodo de τ_φ.

Semántica Operacional.

- Cada nodo del árbol es una cláusula objetivo. $\mathcal{G}_i$ es el *nodo padre* de $\mathcal{G}_{ij}$ y $\mathcal{G}_{ij}$ es un *nodo hijo* de $\mathcal{G}_i$.

- Los nodos hojas son o bien la cláusula vacía $\square$ o *nodos de fallo*.

- Cada rama de un árbol SLD es una derivación SLD para $\Pi \cup \{\mathcal{G}\}$.

- Ramas `infinitas`, de `éxito` y de `fallo`.

- Arbol `infinito`, árbol `finito` y de `fallo finito`.

- Cada regla de computación φ da lugar a un árbol SLD para $\Pi \cup \{\mathcal{G}\}$ distinto.

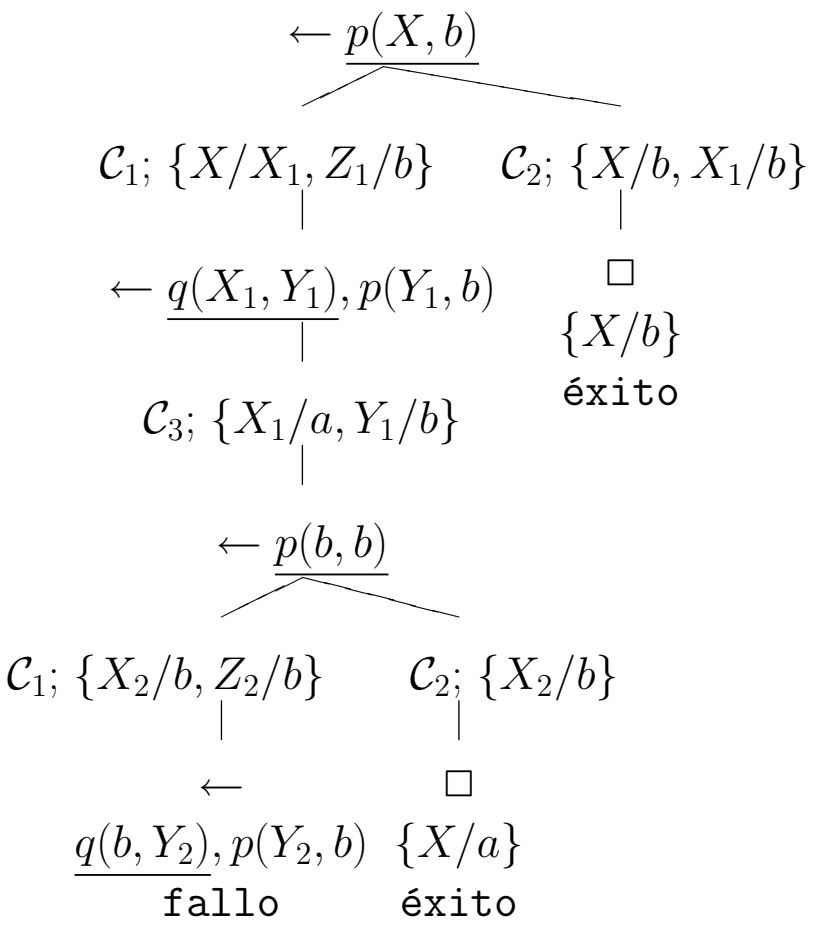
- **Ejemplo 52** *Sea el programa definido*

$$\Pi = \{ \begin{array}{l} \mathcal{C}_1 : \ p(X,Z) \leftarrow q(X,Y) \wedge p(Y,Z) \\ \mathcal{C}_2 : \ p(X,X) \leftarrow \\ \mathcal{C}_3 : \ q(a,b) \leftarrow \} \end{array}$$

y el objetivo definido $\mathcal{G} \equiv \leftarrow p(X,b)$.

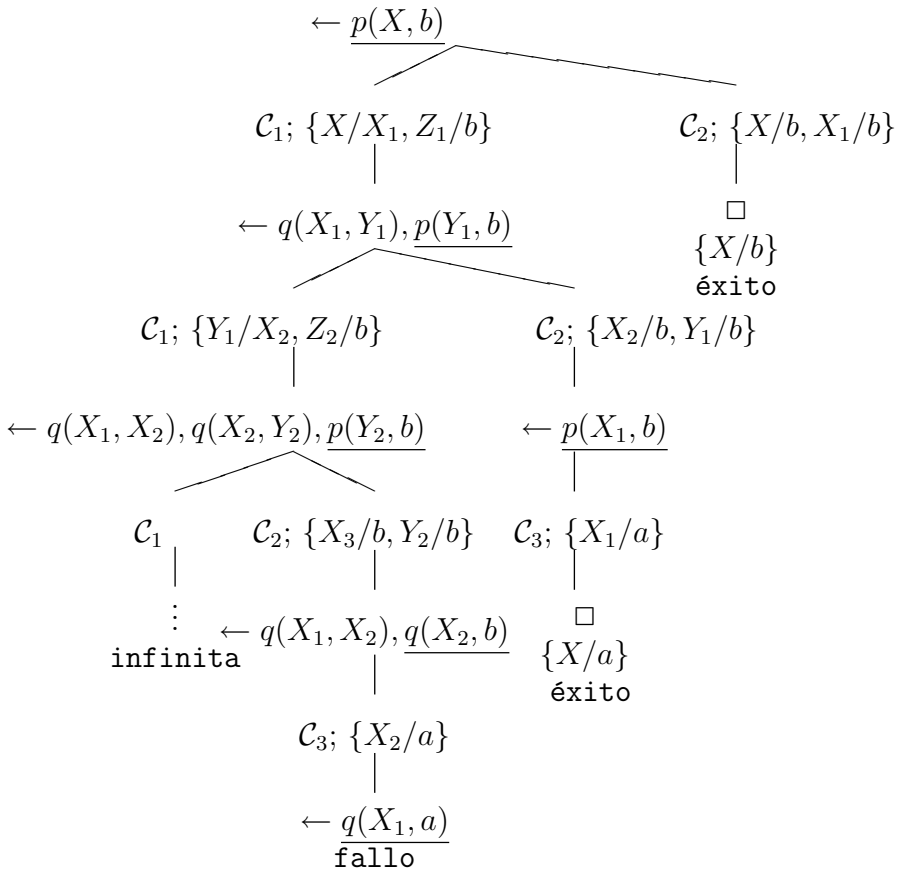
Semántica Operacional.

- Un árbol de búsqueda SLD para $\Pi \cup \{\mathcal{G}\}$ usando una regla de computación que selecciona el átomo más a la izquierda dentro del objetivo considerado.



Semántica Operacional.

- Un árbol SLD para $\Pi \cup \{\mathcal{G}\}$ que se obtiene cuando el átomo seleccionado es el que está más a la derecha.



- Cada árbol contiene dos ramas de éxito que computan las mismas respuestas $\{X/a\}$ y $\{X/b\}$.

Semántica Operacional.

- En general, el conjunto de respuestas computadas para distintos árboles SLD (construidos con diferentes reglas de computación) para un mismo programa Π y objetivo $\mathcal{G}$ es siempre el mismo.

- **Teorema 4.2.7 (Independencia de la R. de Comp.)**

  Si existe una refutación SLD para $\Pi \cup \{\mathcal{G} \equiv\leftarrow \mathcal{Q}\}$ *con respuesta computada* θ*, usando una regla de computación* φ*, entonces existirá también una refutación SLD para* $\Pi \cup \{\mathcal{G}\}$ *con respuesta computada* θ'*, usando cualquier otra regla de computación* φ'*, y tal que* $\theta'(\mathcal{Q})$ *es una variante de* $\theta(\mathcal{Q})$*.*

- El resultado anterior justifica la selección arbitraria de un literal en un objetivo: indeterminismo `don't care` (no importa).

Semántica Operacional.

- La selección de la cláusula del programa que se utiliza en un paso de resolución es determinante a la hora de alcanzar el éxito en una derivación: indeterminismo `don't know` (no se sabe).

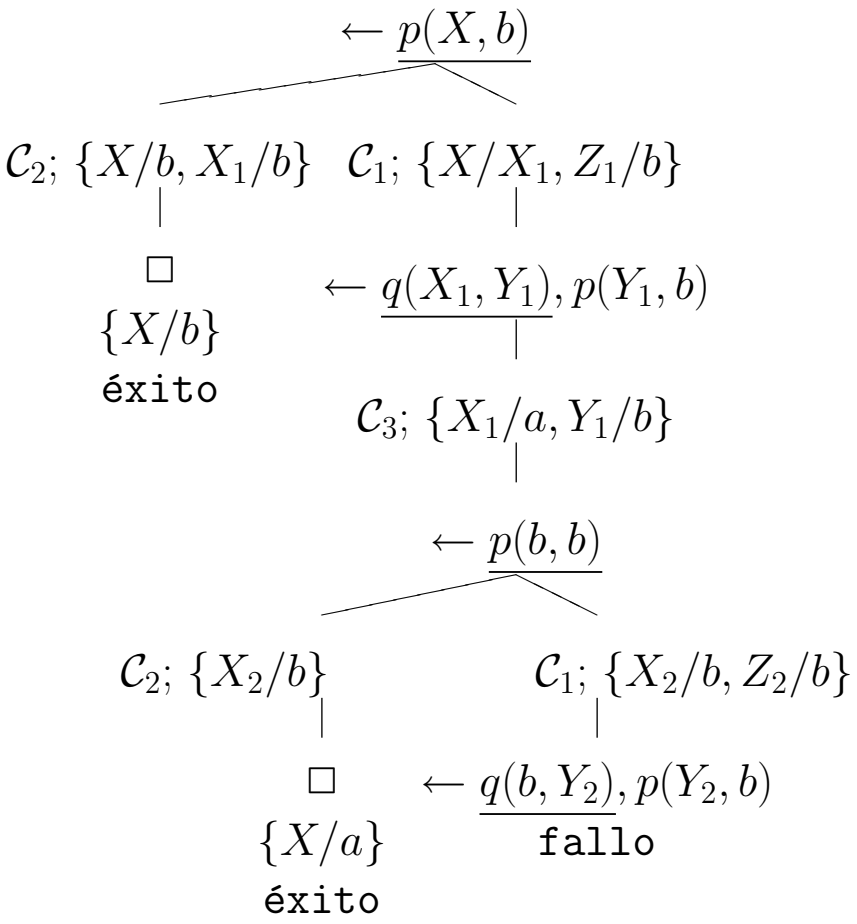
- El orden en el que se eligen las cláusulas de un proprograma influye en la forma del árbol SLD.

- Para una misma regla de computación, distintos ordenes de elección para las cláusulas producen árboles SLD diferentes en los que sus ramas aparecen permutadas.

- **Definición 4.2.8 (Regla de ordenación)**
  Una regla de ordenación *es un criterio por el que se fija el orden en que las cláusulas del programa se intentan resolver con un nodo del árbol SLD, para generar sus correspondientes nodos hijo.*

Semántica Operacional.

- Selección de abajo hacia arriba de las cláusulas del programa Π del Ejemplo 52.



Semántica Operacional.

- Para buscar las ramas de éxito en un árbol SLD es importante la estrategia con la que se recorre dicho árbol.

- **Definición 4.2.9 (Regla de Búsqueda)**
  Una regla de búsqueda *es una estrategia para recorrer un árbol SLD en búsca de una rama de éxito.*

- Hay dos estrategias de búsqueda fundamentales:

  1. Búsqueda en profundidad (*depth-first*).
     - Recorre el árbol SLD visitando primero el nodo más profundo y a la izquierda de entre aquellos todavía no visitados.

     - Implementación mediante una técnica de exploración o mediante un mécanismo de vuelta atrás (*backtracking*).

     - Es eficiente (sólo mantiene información sobre los estados de la rama que se está inspeccianando en el momento actual

     - Puede hacer que se pierda la completitud del procedimiento de refutación SLD.

2. Búsqueda en anchura (*breadth-first*).
   - Se recorre el árbol por niveles.
   - La implementación se realiza mediante una técnica de exploración $\Longrightarrow$ generación implícita del espacio de búsqueda.
   - En cada iteración del procedimiento de refutación se construye el nivel siguiente, generando todos los hijos de los estados objetivo que componen el nivel actual.
   - Si se encuentra la cláusula vacía, se termina con éxito sino se repite el proceso.
   - Mantiene la completitud del procedimiento de refutación SLD.
   - Es muy costosa (crecimiento exponencial del espacio de estados).

Semántica Operacional.

- Un `procedimiento de prueba por refutación SLD` queda completamente especificado fijando:

  1. una regla de computación;

  2. una regla de ordenación; y

  3. una regla búsqueda.

- Las dos primeras reglas fijan la forma del árbol SLD y la última la manera de recorrerlo.

4.3. Semántica Declarativa y Respuesta Correcta.

- La semántica declarativa del lenguaje de programación lógica se fundamenta en la semántica teoría de modelos.

- Dado que el lenguaje es clausal basta con utilizar interpretaciones de Herbrand para dar significado a las construcciones de nuestro lenguaje.
  $\Longrightarrow$ todo lo estudiado en el Capítulo 3 vale aquí.

- La contrapartida declarativa del concepto de respuenta computada es **el concepto de respuenta correcta**.

- **Definición 4.3.1**
  Una respuesta θ *para* $\Pi \cup \{\mathcal{G}\}$ *es cualquier sustitución para las variables de* $\mathcal{G}$.

- **Definición 4.3.2 (Respuesta Correcta)**
  Sea θ *una respuesta para* $\Pi \cup \{\mathcal{G} \equiv\leftarrow \mathcal{Q}\}$, *donde* $\mathcal{Q}$ *es una conjunción de átomos. Entonces,* θ *es una* respuesta correcta *para* $\Pi \cup \{\mathcal{G}\}$ *sii* $\Pi \models (\forall\theta(\mathcal{Q}))$.

Semántica Declarativa y Respuesta Correcta.

- De forma equivalente, θ es una respuesta correcta sii $\Pi \cup \{\neg(\forall\theta(\mathcal{Q}))\}$ es insatisfacible.

- Para comprobar que θ es una respuesta correcta no es suficiente con comprobar que $\neg(\forall\theta(\mathcal{Q}))$ es falsa para todo modelo de Herbrand del programa Π.

- `Razón`: $\neg(\forall\theta(\mathcal{Q}))$ no es una cláusula y el Teorema 3.4.2 no es aplicable
  $\Longrightarrow$ no podemos restringirnos sólo a las interpretaciones de Herbrand.

- **Proposición 4.3.3** *Sea θ una respuesta para $\Pi \cup \{\mathcal{G} \equiv \leftarrow \mathcal{Q}\}$ tal que $\theta(\mathcal{Q})$ es una expresión básica. Entonces, θ es una respuesta correcta para $\Pi \cup \{\mathcal{G}\}$ sii $\theta(\mathcal{Q})$ es verdadera para todo modelo de Herbrand del programa Π.*

Semántica Declarativa y Respuesta Correcta.

- **Ejemplo 53**
  $\Pi = \{p(X,Y) \leftarrow q(Y), \quad q(f(a)) \leftarrow\}$.

  - $\mathcal{U}_{\mathcal{L}}(\Pi) = \{a, f(a), f(f(a)), f(f(f(a))), \ldots\}$

  - *$\mathcal{I}$ es modelo si $\mathcal{M} \subseteq \mathcal{I}$, donde $\mathcal{M} = \{q(f(a))\} \cup \{p(t, f(a)) \mid t \in \mathcal{U}_{\mathcal{L}}(\Pi)\}$.*

  - *Dado el objetivo $\mathcal{G}_1 \equiv\leftarrow p(Z_1, Z_2)$.*
    - *La respuesta $\theta_1 = \{Z_1/a, Z_2/f(a)\}$ es correcta.*
    - *La respuesta $\theta_2 = \{Z_1/b, Z_2/f(a)\}$ no es correcta.*

  - *Dado el objetivo $\mathcal{G}_2 \equiv\leftarrow p(Z, Z)$.*
    - *La respuesta $\sigma_1 = \{Z/f(a)\}$ es correcta.*
    - *La respuesta $\sigma_2 = \{Z/b\}$ no es correcta.*

  - *Dado el objetivo $\mathcal{G}_3 \equiv\leftarrow q(f(a))$.*
    - *La respuesta $\gamma = id$ es correcta.*

  - *Dado el objetivo $\mathcal{G}_1 \equiv\leftarrow q(f(b))$.*
    - *No existe respuesta correcta ya que $\Pi \cup \{\neg q(f(b))\}$ es satisfacible.*

Semántica Declarativa y Respuesta Correcta.

- Existe una ligera discrepancia entre el concepto de respuesta correcta y computada:
  - Plantear un objetivo $\mathcal{G} \equiv \leftarrow \mathcal{Q}$ a un programa $\Pi \Longrightarrow$ comprobar si $\Pi \vdash \exists \mathcal{Q}$
  - Preguntar si una respuesta θ es correcta $\Longrightarrow$ comprobar si $\Pi \models \forall\theta(\mathcal{Q})$.
- La discrepancia desaparece cuando la sustitución θ conduce a una instancia $\theta(\mathcal{Q})$ básica.

4.4. Corrección y Completitud de la resolución SLD.

- **Teorema 4.4.1**
  $\Pi \cup \{\mathcal{G}\}$ *es insatisfacible sii existe una refutación SLD para* $\Pi \cup \{\mathcal{G}\}$.

Corrección y Completitud de la resolución SLD.

- Es habitual presentar los resultados de corrección y completitud en función de la relación entre las respuestas computadas y las respuestas correctas.

- **Teorema 4.4.2 (Teorema de la Corrección)**
  Toda respuesta computada para $\Pi \cup \{\mathcal{G}\}$ *es una respuesta correcta para* $\Pi \cup \{\mathcal{G}\}$.

- **Teorema 4.4.3 (Teorema de la Completitud)**
  Para cada respuesta correcta θ *para* $\Pi \cup \{\mathcal{G}\}$, *existe una respuesta computada* σ *para* $\Pi \cup \{\mathcal{G}\}$ *que es más general que* θ *cuando nos restringimos a las variables de* $\mathcal{G}$ *(en símbolos,* $\sigma \leq \theta[\mathcal{V}ar(\mathcal{G})]$*).*

- Dado que θ, por definición, está restingida a las variables del objetivo $\mathcal{G}$, escribir $\sigma \leq \theta[\mathcal{V}ar(\mathcal{G})]$ es lo mismo que decir que existe una sustitución δ tal que $\theta = (\delta \circ \sigma)_{\restriction(\mathcal{G})}$.

Corrección y Completitud de la resolución SLD.

- **Ejemplo 54**
  $\Pi = \{p(X,Y) \leftarrow q(Y), \quad q(f(a)) \leftarrow\}$ *y* $\mathcal{G} \equiv \quad \leftarrow p(Z_1, Z_2)$.

  - $\mathcal{U}_{\mathcal{L}}(\Pi) = \{a, f(a), f(f(a)), \ldots\}$
  - $\theta = \{Z_1/a, Z_2/f(a)\}$ *es una respuesta correcta,*
  - $\sigma = \{Z_1/X, Z_2/f(a)\}$ *es una respuesta computada,*
  - *y existe una sustitución* $\delta = \{X/a\}$ *tal que* $\theta = (\delta \circ \sigma)_{\restriction(\{Z_1,Z_2\})}$.

- Es importante notar que la relación $\sigma \leq \theta$ no se cumple si se elimina la restricción a las variables de $\mathcal{G}$.

- **Ejemplo 55**
  $\Pi = \{p(X,a) \leftarrow\}$ *y* $\mathcal{G} \equiv \leftarrow p(Y,a)$.
  - $\mathcal{U}_{\mathcal{L}}(\Pi) = \{a\}$
  - $\theta = \{Y/a\}$ *es una respuesta correcta (no hay otra).*
  - *La única respuesta computada es la sustitución* $\sigma = \{Y/X\}$.

- *Existe una sustitución* $\delta = \{X/a\}$ *tal que* $\theta = (\delta \circ \sigma)_{\restriction(\{Y\})}$, *sin embargo, no se cumple que* $\theta = \delta \circ \sigma$.

Corrección y Completitud de la resolución SLD.

- Formulaciones alternativas del teorema de la completitud para la resolución SLD.
  - **Proposición 4.4.4** *Si* $\Pi \cup \{\mathcal{G}\}$ *es insatisfacible entonces cada árbol SLD para* $\Pi \cup \{\mathcal{G}\}$ *contiene (al menos) una rama de éxito.*

  - **Proposición 4.4.5** *Si* $\Pi \cup \{\mathcal{G}\}$ *es insatisfacible entonces para cada átomo* $\mathcal{A} \in \mathcal{G}$ *existe una refutación SLD para* $\Pi \cup \{\mathcal{G}\}$ *con* $\mathcal{A}$ *como primer átomo seleccionado.*

4.5. Significado de los programas.

- Las semánticas formales, además de dar cuenta del significado de las construcciones generales del lenguaje, también deben asignar significado a los programas.

- Asignamos significado a los programas asociando a cada programa ciertas propiedades que denominamos *observables*:
  - el conjunto de éxitos básicos,
  - las derivaciones de éxito,
  - las derivaciones de fallo finito,
  - las respuestas computadas, etc.

- Elegido un observable $\mathcal{O}$, éste induce una relación de equivalencia, $=_{\mathcal{O}}$, sobre los programas:

  $\Pi_1 =_{\mathcal{O}} \Pi_2$ sii Π_1 y Π_2 son indistinguibles con respecto a la propiedad observable $\mathcal{O}$.

- Por razones históricas y de simplicidad expositiva, estamos interesados en el observable éxitos básicos ($\mathcal{EB}$).

4.5.1. Semántica operacional.

- La semántica operacional de un programa definido Π se caracteriza mediante el llamado conjunto de los hechos básicos que pueden establecerse a partir de él.

- **Definición 4.5.1 (Conjunto de Exitos Básicos)** *El conjunto de* éxitos básicos *de un programa definido* Π*, denotado* $\mathcal{EB}(\Pi)$*, es:*

$$\mathcal{EB}(\Pi) = \{\mathcal{A} \mid (\mathcal{A} \in \mathcal{B}_{\mathcal{L}}(\Pi)) \wedge (\Pi \cup \{\leftarrow \mathcal{A}\} \vdash_{SLD} \square)\}.$$

- **Ejemplo 56**
  $\Pi = \{\mathcal{C}_1 : p(a) \leftarrow, \quad \mathcal{C}_2 : p(b) \leftarrow, \quad \mathcal{C}_3 : r(X) \leftarrow p(X), \quad \mathcal{C}_4 : t(X,Y) \leftarrow p(X) \wedge r(Y), \quad \mathcal{C}_5 : s(c) \leftarrow\}$.
  Para calcular la semántica operacional de Π*:*

  1. *Calcular* $\mathcal{U}_{\mathcal{L}}(\Pi)$ *y* $\mathcal{B}_{\mathcal{L}}(\Pi)$*:*

$$\mathcal{U}_{\mathcal{L}}(\Pi) = \{a, b, c\}$$

y

$$\begin{aligned}\mathcal{B}_{\mathcal{L}}(\Pi) = \ & \{p(a), p(b), p(c), r(a), r(b), r(c), s(a), \\ & s(b), s(c), t(a,a), t(a,b), t(a,c), t(b,a), \\ & t(b,b), t(b,c), t(c,a), t(c,b), t(c,c)\}\end{aligned}$$

Significado de los programas.

2. Existen las siguientes refutaciones SLD para $\Pi \cup \{\leftarrow \mathcal{A}\}$, *con* $\mathcal{A} \in \mathcal{B}_{\mathcal{L}}(\Pi)$*:*

a) $\leftarrow p(a) \overset{[\mathcal{C}_1,id]}{\Longrightarrow}_{SLD} \square$.

b) $\leftarrow p(b) \overset{[\mathcal{C}_2,id]}{\Longrightarrow}_{SLD} \square$.

c) $\leftarrow s(c) \overset{[\mathcal{C}_5,id]}{\Longrightarrow}_{SLD} \square$.

d) $\leftarrow r(a) \overset{[\mathcal{C}_3,\{X_3/a\}]}{\Longrightarrow}_{SLD} \leftarrow p(a) \overset{[\mathcal{C}_1,id]}{\Longrightarrow}_{SLD} \square$.

e) $\leftarrow r(b) \overset{[\mathcal{C}_3,\{X_3/b\}]}{\Longrightarrow}_{SLD} \leftarrow p(b) \overset{[\mathcal{C}_2,id]}{\Longrightarrow}_{SLD} \square$.

f) $\leftarrow t(a,a) \overset{[\mathcal{C}_3,\{X_4/a,Y_4/a\}]}{\Longrightarrow}_{SLD} \leftarrow p(a) \wedge r(a) \overset{[\mathcal{C}_1,id]}{\Longrightarrow}_{SLD} \leftarrow r(a) \overset{[\mathcal{C}_3,\{X_3/a\}]}{\Longrightarrow}_{SLD} \leftarrow p(a) \overset{[\mathcal{C}_1,id]}{\Longrightarrow}_{SLD} \square$.

g) $\leftarrow t(a,b) \overset{[\mathcal{C}_3,\{X_4/a,Y_4/b\}]}{\Longrightarrow}_{SLD} \leftarrow p(a) \wedge r(b) \overset{[\mathcal{C}_1,id]}{\Longrightarrow}_{SLD} \leftarrow r(b) \overset{[\mathcal{C}_3,\{X_3/b\}]}{\Longrightarrow}_{SLD} \leftarrow p(b) \overset{[\mathcal{C}_2,id]}{\Longrightarrow}_{SLD} \square$.

h) $\leftarrow t(b,a) \overset{[\mathcal{C}_3,\{X_4/b,Y_4/a\}]}{\Longrightarrow}_{SLD} \leftarrow p(b) \wedge r(a) \overset{[\mathcal{C}_2,id]}{\Longrightarrow}_{SLD} \leftarrow r(a) \overset{[\mathcal{C}_3,\{X_3/a\}]}{\Longrightarrow}_{SLD} \leftarrow p(a) \overset{[\mathcal{C}_1,id]}{\Longrightarrow}_{SLD} \square$.

i) $\leftarrow t(b,b) \overset{[\mathcal{C}_3,\{X_4/b,Y_4/b\}]}{\Longrightarrow}_{SLD} \leftarrow p(b) \wedge r(b) \overset{[\mathcal{C}_2,id]}{\Longrightarrow}_{SLD} \leftarrow r(b) \overset{[\mathcal{C}_3,\{X_3/b\}]}{\Longrightarrow}_{SLD} \leftarrow p(b) \overset{[\mathcal{C}_2,id]}{\Longrightarrow}_{SLD} \square$.

El resto de los átomos $\mathcal{A}$ *de la base de Herbrand* $\mathcal{B}_{\mathcal{L}}(\Pi)$ *no poseen refutaciones SLD para* $\Pi \cup \{\leftarrow \mathcal{A}\}$.

Por consiguiente,

$$\begin{aligned}\mathcal{EB}(\Pi) = \ & \{p(a), p(b), r(a), r(b), s(c),\\ & t(a,a), t(a,b), t(b,a), t(b,b)\}\end{aligned}$$

4.5.2. Semántica declarativa.

- Podemos asignar significado declarativo a un programa, seleccionando uno de sus modelos de Herbrand: el *más simple*.

- **Proposición 4.5.2 (Prop. de Intersec. de Modelos)**

  Sea I un conjunto de índices y un conjunto no vacío *de modelos de Herbrand de un programa* Π, $\{\mathcal{M}_i \mid (i \in I) \wedge (\mathcal{M}_i \text{ es modelo de } \Pi)\}$. *Entonces, el conjunto* $\bigcap_{i \in I} \mathcal{M}_i$ *es un modelo de Herbrand de* Π.

- La propiedad de intersección de modelos sólo se cumple para cláusulas de Horn definidas. Por ejemplo, la cláusula $\mathcal{C} \equiv (p \vee q)$ tiene tres modelos de Herbrand:
  - $\mathcal{M}_1 = \{p\}$;
  - $\mathcal{M}_2 = \{q\}$;
  - $\mathcal{M}_3 = \{p, q\}$;

  cuya intersección es el conjunto vacío, que no es modelo de $\mathcal{C}$.

Significado de los programas.

- Debido a que $\mathcal{B}_{\mathcal{L}}(\Pi)$ es un modelo de Herbrand de un programa Π, el conjunto de todos los modelos de Herbrand de Π no es vacío.

- La anterior precisión, la Observación 3.5.8 y la Proposición 4.5.2 $\Longrightarrow$ la intersección de todos los modelos de Herbrand de un programa Π es su *modelo mínimo de Herbrand*.

- **Teorema 4.5.3**

$$\mathcal{M}(\Pi) = \{\mathcal{A} \mid \mathcal{A} \in \mathcal{B}_{\mathcal{L}}(\Pi) \wedge \Pi \models \mathcal{A}\}$$

- **Ejemplo 57**

$$\Pi = \{\mathcal{C}_1 : q(X) \leftarrow p(X), \quad \mathcal{C}_2 : p(a) \leftarrow, \mathcal{C}_3 : q(b) \leftarrow\}.$$

Para calcular la semántica declarativa de Π*, primero, calculamos* $\mathcal{B}_{\mathcal{L}}(\Pi)$*, depués, las H-interpretaciones que son modelos de* Π*:*

1. $\mathcal{U}_{\mathcal{L}}(\Pi) = \{a, b\}$ *y* $\mathcal{B}_{\mathcal{L}}(\Pi) = \{p(a), p(b), q(a), q(b)\}$.

2. Posibles H-interpretaciones:

- $\mathcal{I}_1 = \emptyset$,

- $\mathcal{I}_2 = \{p(a)\}$, $\mathcal{I}_3 = \{p(b)\}$, $\mathcal{I}_4 = \{q(a)\}$, $\mathcal{I}_5 = \{q(b)\}$,

- $\mathcal{I}_6 = \{p(a), p(b)\}$, $\mathcal{I}_7 = \{p(a), q(a)\}$, $\mathcal{I}_8 = \{p(a), q(b)\}$, $\mathcal{I}_9 = \{p(b), q(a)\}$, $\mathcal{I}_{10} = \{p(b), q(b)\}$, $\mathcal{I}_{11} = \{q(a), q(b)\}$,

- $\mathcal{I}_{12} = \{p(a), p(b), q(a)\}$, $\mathcal{I}_{13} = \{p(a), p(b), q(b)\}$, $\mathcal{I}_{14} = \{p(a), q(a), q(b)\}$, $\mathcal{I}_{15} = \{p(b), q(a), q(b)\}$,

- $\mathcal{I}_{16} = \{p(a), p(b), q(a), q(b)\}$.

Así pues, aún para este sencillo programa hay que realizar un gran esfuerzo de comprobación.

3. *Las únicas H-interpretación modelos de* Π *son:* $\mathcal{I}_{14}$ *e* $\mathcal{I}_{16}$.

La interpretación modelo mínimo de Herbrand es:

$$\mathcal{M}(\Pi) = \mathcal{I}_{14} \cap \mathcal{I}_{16} = \{p(a), q(a), q(b)\}$$

4.5.3. Semántica por punto fijo.

- Objetivo: dar una visión constructiva del significado de un programa siguiendo la teoría del punto fijo.

- La idea es asociar a cada programa un operador continuo.

Teoría del punto fijo.

- Notación: Escribimos $inf(X)$ para denotar el ínfimo de un conjunto X y $sup(X)$ para denotar el supremo de X.

- **Definición 4.5.4** *Un conjunto parcialmente ordenado L es un retículo completo si y sólo si para todo subconjunto X de L existe $inf(X)$ y $sup(X)$.*

- El $sup(L)$ lo denotamos por $\top$ y el $inf(L)$ por $\bot$.

- **Definición 4.5.5** *Sea L un retículo completo y $X \subseteq L$. X es un* conjunto inductivo *(*directed*) si todo subconjunto finito de X tiene una cota superior en X.*

Significado de los programas..

- **Definición 4.5.6** *Sea L un retículo completo y $T : L \longrightarrow L$ una aplicación. Decimos que T es:*

  1. monótona *si y sólo si, siempre que $x \leq y$ entonces $T(x) \leq T(y)$.*
  2. continua *si y sólo si para todo conjunto inductivo X de L entonces, $T(sup(X)) = sup(T(X))$.*

- **Definición 4.5.7** *Sea L un retículo completo y $T : L \longrightarrow L$ una aplicación.*

  - *$a \in L$ es un punto fijo de T si $T(a) = a$.*

  - *Si $a \in L$ es un punto fijo de T, decimos que a es un* menor punto fijo *de T si y sólo si $a \leq b$ para todo punto fijo b de T.*

- El menor punto fijo de T lo denotamos por $mpf(T)$.

- **Proposición 4.5.8 (Teorema del Punto Fijo)** *Sea L un retículo completo y $T : L \longrightarrow L$ una aplicación monótona. Entonces*

  - *T tiene un menor punto fijo $mpf(T)$, y*

  - $$\begin{aligned} mpf(T) &= inf(\{x \mid T(x) = x\}) \\ &= inf(\{x \mid T(x) \leq x\}) \end{aligned}$$

Significado de los programas..

- **Números ordinales.**
  - Números `ordinales finitos`; números `ordinales transfinitos`; primer `ordinal transfinito` ω.
  - Se cumple que $n < w$ para todo ordinal finito n, pero ω no es el sucesor de ningún ordinal finito.
  - ω es un *ordinal límite*.

- **Definición 4.5.9** *Sea L un retículo completo y $T : L \longrightarrow L$ una aplicación monótona. Entonces, definimos las potencias ordinales de T inductivamente como:*

$$\begin{array}{lll} T \uparrow 0 & = \bot & \\ T \uparrow \alpha & = T(T \uparrow (\alpha - 1)), & \textit{si } \alpha \textit{ ordinal sucesor,} \\ T \uparrow \alpha & = inf(\{T \uparrow \beta \mid \beta < \alpha\}), & \textit{si } \alpha \textit{ ordinal limite,} \end{array}$$

- **Proposición 4.5.10** *Sea L un retículo completo y $T : L \longrightarrow L$ una aplicación continua. Entonces,*

$$mpf(T) = T \uparrow \omega$$

El operador de consecuencias lógicas inmediatas.

- `Retículo completo de las interpretaciones de Herbrand`:
  el conjunto $\wp(\mathcal{B}_{\mathcal{L}}(\Pi))$, con el orden "$\subseteq$" de inclusión de conjuntos es un retículo completo.

- **Definición 4.5.11** *Sea* Π *un programa definido. El operador de consecuencias lógicas inmediatas* T_Π *es una aplicación* $T_\Pi : \wp(\mathcal{B}_{\mathcal{L}}(\Pi)) \longrightarrow \wp(\mathcal{B}_{\mathcal{L}}(\Pi))$ *definida como:*

$$T_\Pi(\mathcal{I}) = \{\mathcal{A} \mid \mathcal{A} \in \mathcal{B}_{\mathcal{L}}(\Pi) \wedge (\mathcal{A} \leftarrow \mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_n) \in \mathcal{B}asicas(\Pi) \wedge \{\mathcal{A}_1, \ldots, \mathcal{A}_n\} \subseteq \mathcal{I}\}$$

  donde $\mathcal{I}$ *es una interpretación de Herbrand para* Π *y* $\mathcal{B}asicas(\Pi)$ *es el conjunto de instancias básicas de las cláusulas de* Π.

- La aplicación T_Π es continua.

- A la interpretación $T_\Pi(\mathcal{I})$ pertenecen todos los átomos de la base de Herbrand que son consecuencia lógica inmediata del programa Π.

Significado de los programas.

- **Ejemplo 58**

$$\Pi = \{\mathcal{C}_1 : q(X) \leftarrow p(X), \quad \mathcal{C}_2 : p(a) \leftarrow, \mathcal{C}_3 : q(b) \leftarrow\}.$$

Si $\mathcal{I}_9 = \{p(b), q(a)\}$, *para calcular* $T_\Pi(\mathcal{I}_9)$*: Generar las instancias básicas de las cláusulas de* Π *y comprobar las que se ajustan a las condiciones de la Definición 4.5.11:*

- *Una instancia básica de* $\mathcal{C}_1$ *es* $q(a) \leftarrow p(a)$*, pero no se cumple que* $\{p(a)\} \subseteq \mathcal{I}_9$.

- *Una instancia básica de* $\mathcal{C}_1$ *es* $q(b) \leftarrow p(b)$ *y se cumple que* $\{p(b)\} \subseteq \mathcal{I}_9$*, por lo tanto* $q(b) \in T_\Pi(\mathcal{I}_9)$.

- *La cláusula* $\mathcal{C}_2 \equiv p(a) \leftarrow$ *es básica . Su cuerpo es* $\emptyset$*. Se cumple, trivialmente, que* $\emptyset \subseteq \mathcal{I}_9$ *y por lo tanto* $p(a) \in T_\Pi(\mathcal{I}_9)$.

- *Respecto a la cláusula* $\mathcal{C}_3$*, por razonamiento identico al último caso,* $q(b) \in T_\Pi(\mathcal{I}_9)$.

Por consiguiente

$$T_\Pi(\mathcal{I}_9) = \{p(a), q(a), q(b)\}$$

Significado de los programas.

- Las H-interpretaciones que son modelo del programa Π pueden caracterizarse en términos del operador T_Π.

- **Proposición 4.5.12** *Sea* Π *un programa definido e* $\mathcal{I}$ *una H-interpretación de* Π. *Entonces,* $\mathcal{I}$ *es modelo de* Π *si y sólo si* $T_\Pi(\mathcal{I}) \subseteq \mathcal{I}$.

- **Teorema 4.5.13 (Caract. por Punto Fijo de $\mathcal{M}_\Pi$)**

$$\mathcal{M}(\Pi) = mfp(T_\Pi) = T_\Pi \uparrow \omega$$

- Este resultado nos proporciona un procedimiento constructivo para el cómputo del modelo mínimo de Herbrand de un programa.

- Informalmente, este procedimiento consiste en:
  1. añadir los hechos al conjunto de partida.
  2. aplicar las reglas a los hechos para generar nuevos hechos, que se añaden al conjunto anterior;
  3. repitir las operaciones (1) y (2) hasta que no se obtengan hechos nuevos.

Significado de los programas.

- **Ejemplo 59**

Consideremos de nuevo el pograma del Ejemplo 56.

$$
\begin{aligned}
T_\Pi \uparrow 0 &= \emptyset \\
\\
T_\Pi \uparrow 1 &= T_\Pi(\emptyset) \\
&= \{p(a), p(b), s(c)\} \\
\\
T_\Pi \uparrow 2 &= T_\Pi(T_\Pi \uparrow 1) \\
&= \{p(a), p(b), s(c)\} \cup \{r(a), r(b)\} \\
&= \{p(a), p(b), s(c), r(a), r(b)\} \\
\\
T_\Pi \uparrow 3 &= T_\Pi(T_\Pi \uparrow 2) \\
&= \{p(a), p(b), s(c), r(a), r(b)\} \\
&\quad \cup \{t(a,a), t(a,b), t(b,a), t(b,b)\} \\
&= \{p(a), p(b), s(c), r(a), r(b), \\
&\quad t(a,a), t(a,b), t(b,a), t(b,b)\} \\
\\
T_\Pi \uparrow 4 &= T_\Pi(T_\Pi \uparrow 3) = T_\Pi \uparrow 3
\end{aligned}
$$

Se alcanza el punto fijo en la cuarta iteración y por consiguiente,

$$
\begin{aligned}
\mathcal{M}_\Pi = \{&p(a), p(b), s(c), r(a), r(b), \\
&t(a,a), t(a,b), t(b,a), t(b,b)\}.
\end{aligned}
$$

- **Ejemplo 60**
  Consideremos de nuevo el pograma del Ejemplo 57.

$$T_\Pi \uparrow 0 = \emptyset$$

$$T_\Pi \uparrow 1 = T_\Pi(\emptyset) = \{p(a), q(b)\}$$

$$\begin{aligned} T_\Pi \uparrow 2 = T_\Pi(T_\Pi \uparrow 1) &= \{p(a), q(b)\} \cup \{q(a)\} \\ &= \{p(a), q(b), q(a)\} \end{aligned}$$

$$T_\Pi \uparrow 3 = T_\Pi(T_\Pi \uparrow 2) = T_\Pi \uparrow 2$$

Se alcanza el punto fijo en la tercera iteración y por consiguiente,

$$\mathcal{M}_\Pi = \{p(a), q(b), q(a)\}.$$

4.5.4. Equivalencia entre semánticas.

- **Teorema 4.5.14**
  Sea Π *un programa definido.* $\mathcal{EB}(\Pi) = \mathcal{M}(\Pi)$

Capítulo 5

Programación Lógica y Prolog.

- Prolog (PROgramación en LOGica – Colmenauer et. al. 1973) es la realización más conocida de las ideas de la programación lógica.

- En esencia, un programa Prolog es un conjunto de cláusulas de Horn, que por motivos de eficiencia se tratan como una secuencia, y junto con un objetivo se ejecuta utilizando el mecanismo operacional de laresolución SLD.

- Para usar la programación lógica como un lenguaje de programación útil ha sido necesario mejorar sus prestaciones.

- Prolog se aleja de algunos de los presupuestos de la programación lógica.

Programación Lógica y Prolog.

- El lenguaje Prolog incorpora:
  1. Modificaciones en el mecanismo operacional para conseguir una mayor eficiencia:
     a) eliminación de la regla de *occur check* en el procedimiento de unificación;

     b) la posibilidad de utilizar el operador de *corte* para reducir el espacio de búsqueda.
  2. Aumento de la expresividad a través de una sintaxis extendida:
     a) conectivas "$\vee$" y "$\neg$" en el cuerpo de las cláusulas;

     b) operadores predefinidos (*built-in*):
     - aritméticos (e.g., "`+`", "`-`", "`*`", "`div`", y "`/`");
     - de comparación (e.g., "`=:=`", "$<$", "`>`");

     c) predicados predefinidos para facilitar las operaciones de entrada y salida (e.g., "`read`" y "`write`");

     d) predicados predefinidos para la manipulación de términos (e.g., "`=..`", "`functor`", "`arg`" y "`name`");

e) predicados predefinidos para la manipulación de cláusulas (e.g., "`assert`" y "`retract`");

f) predicados predefinidos para la metaprogramación (e.g., "`call`" y "`clause`");

- En este capítulo se discuten cada una de estas características y su influencia en el lenguaje.

5.1. Prolog Puro.

5.1.1. Sintaxis.

- Las construcciones del lenguaje son términos y sus instrucciones son cláusulas definidas.

- Notación que utiliza Prolog para designar las conectivas

Conectiva lógica		Sintaxis
Nombre	Símbolo	Prolog
Conjunción	$\wedge$	`,`
Disjunción	$\vee$	`;`
Condicional reciproco (en una regla)	$\leftarrow$	`:-`
Negación	$\neg$	`not`
Condicional reciproco (en una cáusula objetivo)	$\leftarrow$	`?-`

- Las cáusulas deben terminar con un punto ".".

Prolog Puro.

- Los símbolos de constante, función, relación y los símbolos de variable se denotan mediante identificadores.

- **Identificador**: cualquier cadena de caracteres alfanuméricos (`a`, . . ., `z`, `A`, . . . ,`Z`, `0`, . . ., `9`) y el subrayado "_"

- Los identificadores para los símbolos de constante, función, y relación deben comenzar por una letra minúscula.

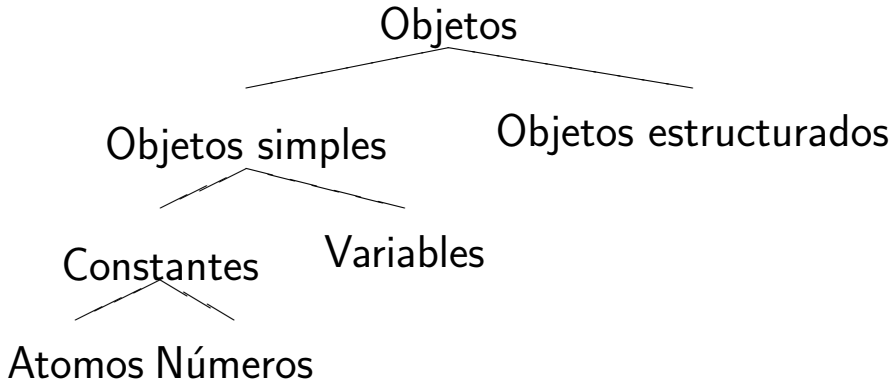
- Los nombres de constantes pueden ser identificadores encerrados entre comillas simples.

- Los identificadores para las variables deben comenzar por una letra mayúscula o subrayado.

- Sintaxis ambivalente.

Prolog Puro.

- Clasificación de los objetos (términos) de Prolog.



- En lo que sigue mantendremos la nomenclatura de la programación lógica dejando de lado la jerga introducida por el lenguaje Prolog.

5.1.2. Mecanismo operacional.

- Prolog puro usa la estrategia de resolución SLD modificada con las siguientes restricciones:

  1. `Regla de computación`: selecciona el átomo situado más a la izquierda dentro de la secuencia.

  2. *Regla de ordenación*: toma las cláusulas de arriba hacia abajo.

  3. *Regla de búsqueda*: en profundidad con vuelta atrás (*backtracking*). $\Longrightarrow$
     Se recorre primero la rama más a la izquierda del árbol de búsqueda.

  4. Prolog omite la prueba de ocurrencia de variables *occur check* cuando realiza la unificación de dos expresiones. $\Longrightarrow$
     es necesario modificar el algoritmo de unificación, eliminando la regla 6 de la Definición 3.6.15.

  5. La posibilidad de utilizar sintaxis ambivalente $\Longrightarrow$ necesidad de modificar la regla 5 de la Definición 3.6.15:

     - `Regla de fallo`:
       $$\frac{\langle \{P(t_1, \ldots, t_n) = Q(s_1, \ldots, s_m)\} \cup E, \theta \rangle}{\langle Fallo, \theta \rangle}$$
       si $P \neq Q$ o $n \neq m$.

Prolog Puro.

- Un *árbol de búsqueda de Prolog* es un árbol de búsqueda SLD en el que las reglas de computación, ordenación y búsqueda están fijadas en la fórma especificada más arriba.

- Debido a que estas reglas están fijadas, para un programa Π y un objetivo $\mathcal{G}$, el árbol de búsqueda de Prolog es único.

- Exploración de un árbol de búsqueda de Prolog.

Algoritmo 8 (Exploración en Profundidad)

Entrada: *Un Programa* $\Pi = \{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$ *y un objetivo* $\mathcal{G}$.

Salida: *Una respuesta computada* σ *y* `éxito` *o una condición de* `fallo`.

Comienzo
  Inicialización: $NuevoEstado = \langle \mathcal{G}, id \rangle$; $PilaEstados = pilaVacia$;
  $apilar(NuevoEstado, PilaEstados)$;
  Repetir
  1. $\langle \mathcal{Q} \equiv \leftarrow \mathcal{B} \wedge \mathcal{Q}', \theta \rangle = desapilar(PilaEstados)$;
  2. **Si** $\mathcal{Q} \equiv \square$ **entonces**
     Comienzo
       Devolver $\sigma = \theta$ *y* `éxito`;
       Si *se desean más respuestas* **entonces** **continuar**
       sino **terminar**;
     Fin
  3. **Sino Comienzo**
     Para $i = k$ **hasta** 1 **hacer**
       Comienzo
         $\mathcal{C}_i \equiv \mathcal{A}_i \leftarrow \mathcal{Q}_i$;
         Si $\theta_i = mgu(\mathcal{A}_i, \mathcal{B}) \not\equiv$ `fallo` **entonces**
           Comienzo
             $NuevoEstado = \langle \leftarrow \theta_i(\mathcal{Q}_i \wedge \mathcal{Q}'), \theta_i \circ \theta \rangle$;
             $apilar(NuevoEstado, PilaEstados)$;
           Fin
       Fin
     Si $\mathcal{Q}$ *no tiene hijos* **entonces** **Devolver** `fallo`
     Fin
  Hasta que $PilaEstados \equiv pilaVacia$;
  Devolver `fallo`.
Fin

Prolog Puro.

- El Algoritmo 8 genera, parcialmente, un árbol de búsqueda en un espacio de estados. Un estado es un par objetivo $\mathcal{G}$ sustitución σ, denotado $\langle \mathcal{G}, \sigma \rangle$ (véase el final del Apartado 4.2.2).

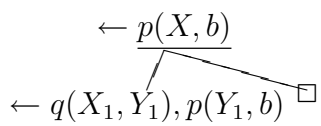
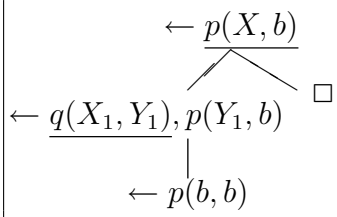
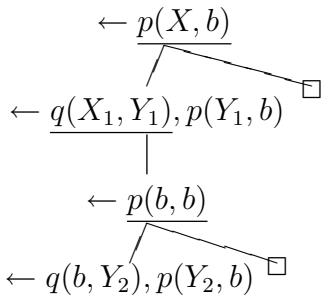
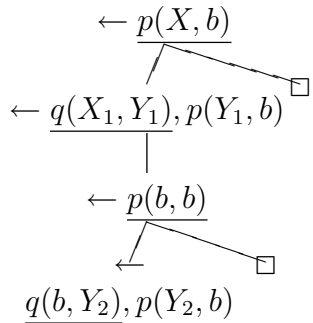
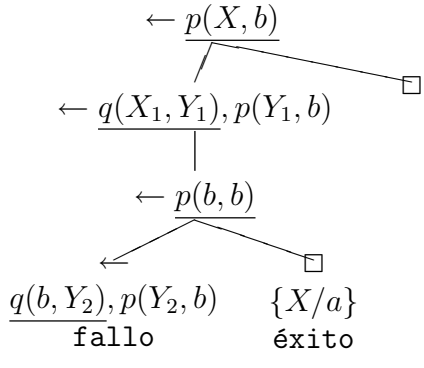
- Puesto que pueden existir ramas infinitas, este algoritmo no siempre termina.

- El Algoritmo 8 se transforma en un algoritmo de exploración en anchura, sin más que realizar ligeros cambios.

- Detalle de como el Algoritmo 8 genera el árbol de búsqueda de Prolog para el programa Π y el objetico $\mathcal{G}$ del Ejemplo 52.

- Vuelta atrás y enlace/desenlace de las variables.

Prolog Puro.

$\leftarrow p(X,b)$	$\Longrightarrow$ Iteración 1	
	$\Longrightarrow$ Iteración 2	
	$\Longrightarrow$ Iteración 3	
	$\Longrightarrow$ Iteración 4	
	$\Longrightarrow$ Iteración 5	

5.1.3. Programación en Prolog puro.

Los números naturales.

- Los postulados de Peano son el sistema de axiomas que caracterízan los números naturales:

  1. $0 \in \mathbb{N}$ (i.e., el 0 es un número natural).

  2. Para todo número n, si $n \in \mathbb{N}$ entonces $s(n) \in \mathbb{N}$. el número natural $s(n)$ se denomina el *sucesor* de n.

  3. Para todo número n, si $n \in \mathbb{N}$ entonces $s(n) \neq 0$.

  4. Para todo par números naturales n y m, $s(n) = s(m)$ si y sólo si $n = m$.

  5. Para todo conjunto A de números naturales, si $0 \in A$ y $s(n) \in A$ siempre que $n \in A$ entonces todo número natural está contenido en A.

Prolog Puro.

- Los postulados primero y segundo.

```
% natural(N), N es un natural
natural(cero).
natural(suc(N)) :- natural(N).

?- natural(suc(cero)).
yes

?- natural(suc(suc(suc(cero)))).
yes

?- natural(a).
no

% INVERSIBILIDAD (ADIRECCIONALIDAD)
?- natural(X).
X = cero ;
X = suc(cero) ;
X = suc(suc(cero))
```

- El algoritmo de unificación sirve para construir estructuras que son enlazadas a las variables como valores.

Prolog Puro.

- Los axiomas tercero y cuarto pueden entenderse como la definición de una relación de igualdad entre los números naturales.
  - El tercer axioma enuncia la negación de un hecho y no puede representarse explícitamente.
  - El axioma cuarto presenta la peculiaridad de que se formaliza en la lógica de primer orden empleando el bicondicional.
  - Conviene desechar la parte "sólamente-si" del bicondicional y quedarse con la parte "si", cuando se representa la fórmula en notación clausal.
  - Se obtiene el siguiente programa:

```
% eqNatural(N, M), N y M son iguales
eqNatural(cero, cero).
eqNatural(suc(N), suc(M)) :-
          natural(N), natural(M),
          eqNatural(N, M).
```

- El quinto axioma contiene un cuantificador de segundo orden que no es expresable en notación clausal.

Prolog Puro.

- Axiomas adicionales:
  - Para todo natural n, $0 + n = n$.
  - Para todo natural n y m, $s(n) + m = s(n + m)$.
  - Para todo natural n, $0 * n = 0$.
  - Para todo natural n y m, $s(n) * m = (n * m) + m$.

- Las funciones binarias se representan en Prolog mediante relaciones ternarias:

```
% suma(N, M, S), la suma de N y M es S
suma(cero, N, N) :- natural(N).
suma(suc(N), M, suc(S)) :-
     natural(N), natural(M), suma(N, M, S).
% producto(N, M, S), el producto de N y M es P
producto(cero, N, cero) :- natural(N).
producto(suc(N), M, P) :-
     natural(N), natural(M),
     producto(N, M, P1), suma(P1, M, P).
```

Prolog Puro.

- Los predicados definidos en este apartado son una muestra de *definición inductiva* (recursiva) basada en la estructura de los elementos del dominio de discurso.

- Patrones. Definición por ajuste de Patrones.

- Debido a que la mayoría de las implementaciones de Prolog no incorporan un sistema de tipos de datos, las comprobaciones de tipos son responsabilidad del programador.

- Comprobar los dominios (tipos) de los argumentos de un predicado afectará al rendimiento del programa.

- **Solución**: Eliminar las comprobaciones o utilizar funciones auxiliares:

```
% suma(N, M, S), la suma de N y M es S
suma(N, M, S) :-
     natural(N), natural(M), sum(N, M, S).
sum(cero, N, N).
sum(suc(N), M, suc(S)) :- sum(N, M, S).
```

Listas.

- La representación interna de las listas puede tratarse como un caso especial de la representación de términos (mediante árboles binarios).

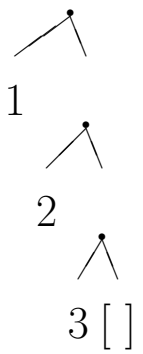
- una lista puede definirse inductivamente como sigue:
  1. [ ] es una lista;
  2. $.(X, L)$ es una lista, si X es un elemento y L una lista; X se denomina la *cabeza* y L es la *cola* de la lista.

- La constante "[ ]" representa la lista vacía.

- El símbolo de función "." es un operador binario que añade un elemento a una lista.

- "[ ]" y "." son los constructores del dominio de las listas.

- La lista $.(1, .(2, .(3, [\,])))$ representada como un árbol binario:



- **Ejemplo 61** *Algunos ejemplos de listas son:*

  $.(a, .(b, .(c, .(d, [\,]))))$,
  $.(1, .(.(juan, .(pedro, [\,])), .(2, [\,])))$,
  $.([\,], .(.(a, .(b, [\,])), .(.(1, .(2, .(3, [\,]))), [\,])))$,
  $.([\,], [\,])$

- La representación de las listas que utiliza el operador de concatenación "." se presta a la confusión.

- El lenguaje Prolog proporciona al programador una notación alternativa para representar las listas:
  - el símbolo de función "." se sustituye por el operador infijo "`[.|..]`".

- **Ejemplo 62** *Las listas del Ejemplo61 pueden denotarse como:*

  ```
  [a|[b|[c|[d|[ ]]]]],
  [1|[[juan|[pedro|[ ]]]|[2|[ ]]]],
  [[ ]|[[a|[b|[ ]]]|[[1|[2|[3|[ ]]]]|[ ]]]],
  [[ ]|[ ]]
  ```

- Esta notación todavía no es lo suficientemente legible, por lo que se permiten las siguientes abreviaturas:
  1. `[`e_1`|[`e_2`|[`e_3`|...[`e_n`|L]]]]`
     se abrevia a `[`e_1`, `e_2`, `e_3`, ..., `e_n`|L]`; y
  2. `[`e_1`, `e_2`, `e_3`, ..., `e_n`|[ ]]`
     se abrevia a `[`e_1`, `e_2`, `e_3`, ..., `e_n`]`

- La lista `[a|[b|[c|[d|[ ]]]]]` puede representarse como:
  - `[a, b, c, d]`,
  - `[a|[b, c,d]]`, o
  - `[a, b, c|[d]]`.

- Una lista no tiene, necesariamente, que acabar con la lista vacía: `[a, b|L]`.

- **Ejemplo 63** *Con las nuevas facilidades, las listas del Ejemplo 62 pueden denotarse como:*

  ```
  [a, b, c, d],
  [1, [juan, pedro], 2],
  [[ ], [a, b], [[1, 2, 3]],
  [[ ],[ ]]
  ```

Prolog Puro.

- Predicado `esLista`: confirma si un objeto es o no una lista.

```
% esLista(L), L es una lista
esLista([ ]).
esLista([_|R]) :- esLista(R).
```

- Predicados predefinidos para la manipulación de listas: `member` y `append`.

- Con la ayuda del predicado member y append se pueden definir una gran variedad de predicados.

- **Conjuntos**.
  - El predicado `conjunto(C)` es verdadero cuando C es un conjunto.

```
% Tipo de datos Conjunto
repeticiones([X|R]) :-
   member(X, R); repeticiones(R).

% conjunto(C), la lista C es un conjunto
conjunto(C) :- not repeticiones(C).
```

  - Se emplea `not` con un sentido puramente declarativo, interpretandolo como la conectiva "¬".

Prolog Puro.

- El predicado `en(X, C)` define la relación de pertenencia de un elemento `X` a un conjunto `C`.

```
% en(X, C), X pertenece al conjunto C
en(X, C) :- conjunto(C), member(X, C).
```

- El predicado subc(A, B) define la relación de inclusión entre conjuntos.

```
% subc(A, B), A es subconjunto de B
subc(A, B) :-
   conjunto(A), conjunto(B), subc1(A, B).
subc1([], _).
subc1([X|R], C) :- member(X, C), subc1(R, C).
```

- El predicado `dif(C1, C2, D)` define la operación diferencia de conjuntos. El conjunto diferencia `D`, está formado por los miembros de `C1` que no están en `C2`.

```
% dif(C1, C2, D), D es el conjunto diferencia
% de C1 y C2
dif(C1, C2, D) :-
   conjunto(C1), conjunto(C2), dif1(C1, C2, D).
dif1(C1, [], C1).
dif1([], _, []).
dif1([X|R], C2, D) :-
   member(X, C2), dif1(R, C2, D).
dif1([X|R], C2, [X|D1]) :-
   not member(X, C2), dif1(R, C2, D1).
```

- El predicado `union(C1, C2, U)`. La unión `U` de dos conjuntos C1 y C2 es el conjunto formado por los elementos que pertenecen a C1 o a C2.

```
% union(C1, C2, U), U es la union de C1 y C2
union(C1, C2, U) :-
   conjunto(C1), conjunto(C2), uni(C1, C2, U).
uni(U, [], U).
uni([], U, U).
uni([X|R], C2, U) :-
   member(X, C2), uni(R, C2, U).
uni([X|R], C2, [X|U]) :-
   not member(X, C2), uni(R, C2, U).
```

- El predicado `ultimo(U, L)`, donde `U` es el último elemento de la lista `L`

```
% ultimo(U, L), U es el ultimo elemento
% de la lista L
ultimo(U, [U]).
ultimo(U, [_|R]) :- ultimo(U, R).
```

También puede definirse usando la relación `append`.

```
% ultimo(U, L), U es el ultimo elemento
% de la lista L
ultimo(U, L) :- append(_, [U], L).
```

Prolog Puro.

- Un prefijo de una lista es un segmento inicial de la misma. Un sufijo es un segmento final.

```
% prefijo(P, L), P es el prefijo de la lista L
prefijo(P, L) :- esLista(L), append(P, _, L).
% sufijo(P, L), P es el sufijo de la lista L
sufijo(P, L) :- esLista(L), append(_, P, L).
% sublista(S, L), S es una sublista de la lista L
sublista(S, L) :- prefijo(S, L1), sufijo(L1, L).
```

  La última definición afirma que `S` es una sublista de `L` si `S` está contenida como prefijo de un sufijo de `L`.

- **Invertir una lista**:

```
% invertir(L,I), I es la lista que resulta
% de invertir L
invertir([],[]).
invertir([H|T],L) :- invertir(T,Z), append(Z,[H],L).
```

  El número de llamadas al operador "`[.|..]`" es cuadrático con respecto al número de elementos de la lista.

Prolog Puro.

- Buscando la eficiencia:

  1. Parámetros de acumulación.

```
% invertir(L,I), I es la lista inversa de L
invertir(L,I) :- inv(L, I, []).
% inv(Lista, Invertida, Acumulador)
inv([], I, I).
inv([X|R], I, A) :- inv(R, I, [X|A]).
```

     El número de llamadas a "`[.|..]`" es lineal.

  2. Eliminar las comprobaciones.

```
% prefijo(P, L), P es el prefijo de L
prefijo(P, L) :- append(P, _, L).
% sufijo(P, L), P es el sufijo de L
sufijo(P, L) :- append(_, P, L).
```

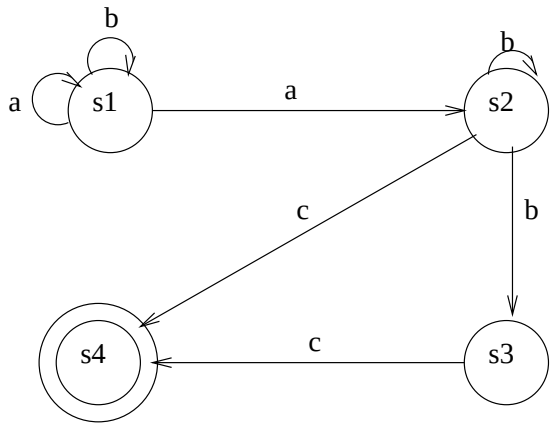
  3. Transformación de Desplegado (Unfolding):

```
% sublista(S, L), S es una sublista de L
sublista(S, L) :-
   prefijo(S, L1), sufijo(L1, L).
```

     Sustituir la llamada por el cuerpo de la regla que la define: `prefijo(S, L1)` $\Longrightarrow$ `append(S, _, L1)`.
     `sufijo(L1, L)` $\Longrightarrow$ `append(_, L1, L)`.

```
sublista(S, L) :-
   append(S, _, L1), append(_, L1, L).
```

5.1.4. Autómata finito no determinista.



- Estado inicial: s1. Estado final: s4 (enmarcado en un doble círculo).

- Son posibles transiciones no deterministas

- El autómata de la figura puede especificarse mediante:
  - Estados: las constantes `s1`, `s2`, `s3`, `s4`.
  - Símbolos de entrada: las constantes `a`, `b`, `c`. Las cadenas de símbolos de entrada serán listas.
  - Las relaciones:
    1. `final(S)`, que caracteríza el estado final del autómata.
    2. `trans(S1, X, S2)`, define las transiciones posibles.

Prolog Puro.

```
% Descripcion del automata
% final(s4), s4 es el estado final
final(s4).
% trans(S1, X, S2), transicion de S1 a S2
% cuando la entrada es X
% Transiciones que producen cambio de estado
trans(s1, a, s2).
trans(s2, c, s4).
trans(s2, b, s3).
trans(s3, c, s4).
% Transiciones que no producen cambio de estado
trans(s1, a, s1).
trans(s1, b, s1).
trans(s2, b, s2).
```

- El autómata acepta una cadena si existe un camino que:
  1. comienza en el estado `Estado`, que se considera inicial;
  2. termina en el estado final `s4` y
  3. las etiquetas de los arcos se corresponden con la cadena de entrada.

- La relación `acepta(Estado, Cadena)`.

```
% acepta(Estado, Cadena), la Cadena es aceptada,
% cuando el automata esta en el estado Estado
acepta(E, []) :- final(E).
acepta(E, [C|R]) :-
    trans(E, C, E2), acepta(E2, R).
```

Prolog Puro.

Los programas que se han estudiado hasta el momento ponen de manifiesto algunos puntos importantes:

1. Los programas pueden extenderse añadiendo nuevas cláusulas.

2. Las cláusulas son de dos tipos:

   a) Hechos: declaran propiedades incondicionalmente verdaderas.

   b) Reglas: declaran propiedades que son verdaderas dependiendo unas condiciones.

3. Las reglas se definen habitualmente por inducción (recursión). Inducción estructural si se definen propiedades de objetos sintácticos.

4. Las reglas se definen sin pensar en la forma en la que se ejecutará el programa (significado procedural). Las reglas se definen pensando en su significado declarativo.

5. Los objetos matemáticos abstractos (como el autámata del Apartado 5.1.4) tienen una representación directa en Prolog.

5.2. Prolog puro no es pura lógica.

- No realizar la comprobación de ocurrencia de variables (*occur check*) durante el proceso de unificación lleva a una perdida de la corrección

  Ejemplo 64 *Dado el programa:*

  ```
  prodigio :- es_hijo(X,X).
  es_hijo(Y, padre(Y)).
  ```

  el intérprete responde sí a la pregunta "`?- prodigio.`*". Pero es evidente que prodigio no puede ser una consecuencia lógica del programa.*

- Adoptar una estrategia de búsqueda en profundidad hace perder la completitud.

  Ejemplo 65 *Dado el programa:*

  ```
  natural(suc(X)) :- natural(X).
  natural(cero).
  ```

  el intérprete es incapaz de responder a la pregunta "`?- natural(X).`*", si bien existen infinitas respuestas (*`cero`*, `suc(cero)`, `suc(suc(cero))`, ...*). El problema es que el intérprete entra por una rama infinita antes de encontrar solución alguna.*

Prolog puro no es pura lógica.

- La programación en Prolog puede encerrar ciertas sutilezas:

```
% HECHOS
% padre(X, Y), X es padre de Y
padre(teraj, abraham).
padre(teraj, najor).
padre(teraj, haran).
padre(teraj, sara).
padre(haran, lot).
padre(haran, melca).
padre(haran, jesca).
padre(najor, batuel).
padre(batuel, rebeca).
padre(batuel, laban).
padre(abraham,ismael).
padre(abraham, isaac).
padre(isaac, esau).
padre(isaac, jacob).

% REGLAS
% asc_directo(X, Y), X es ascendiente
% directo de Y
asc_directo(X, Y) :-
   (padre(X, Y); madre(X, Y)).

% ascendiente(X, Y), X es ascendiente de Y
ascendiente(X, Z) :- asc_directo(X, Z).
ascendiente(X, Z) :-
   asc_directo(X, Y), ascendiente(Y, Z).
```

Prolog puro no es pura lógica.

- Ampliamos el programa con diferentes versiones de la relación ascendiente:
  - Versión 2. Se intercambia el orden de las cláusulas de la versión original.

```
ascendiente2(X, Z) :-
   asc_directo(X, Y), ascendiente2(Y, Z).
ascendiente2(X, Z) :- asc_directo(X, Z).
```

  - Versión 3. Se intercambia el orden de los objetivos de la versión original.

```
ascendiente3(X, Z) :- asc_directo(X, Z).
ascendiente3(X, Z) :-
   ascendiente3(Y, Z), asc_directo(X, Y).
```

  - Versión 4. Se intercambia el orden de los objetivos y de las cláusulas de la versión original.

```
ascendiente4(X, Z) :-
   ascendiente4(Y, Z), asc_directo(X, Y).
ascendiente4(X, Z) :- asc_directo(X, Z).
```

- Una sesión con un intérprete Prolog:

```
?- ascendiente(A, laban).
     A = batuel ;
     A = teraj ;
     A = teraj ;
     A = haran ;
     A = najor ;
     A = melca ;
     no
```

Prolog puro no es pura lógica.

```
?- ascendiente2(A, laban).
     A = teraj ;
     A = teraj ;
     A = haran ;
     A = najor ;
     A = melca ;
     A = batuel ;
     no

?- ascendiente3(A, laban).
     A = batuel ;
     A = najor ;
     A = melca ;
     A = teraj ;
     A = haran ;
     A = teraj ;
Not enough heap space. Resetting system.
System usage: :
control: 3276:6546:3263:-239 variables: 22890:24877
<Heap space exhausted>:
(3264:3275) asc_directo(_11417,teraj). : a

?- ascendiente4(A, laban).
Not enough heap space. Resetting system.
System usage: :
control: 3081:6162:3082:0 variables: 21570:26197
<Heap space exhausted>:
| | | | (3081:3080) ascendiente4(_10780,laban). : a
?-
```

Prolog puro no es pura lógica.

- Los programas que se han comentado en este apartado muestran las discrepancias entre el significado declarativo y el procedural de un programa.

- El significado declarativo de un programa no cambia al alterar el orden de las cláusulas en el programa o de los objetivos dentro de una cláusula. Sin embargo vemos que su significado procedural sí.

- Recomendaciones (I. Bratko):
  - Intenta hacer primero las cosas simples.
  - Hay que centrarse en los aspectos declarativos del problema. Después comprobar, mediante su ejecución, que el programa se comporta según su significado esperado y si falla proceduralmente entonces intentar reordenar las cláusulas y los objetivos en un orden conveniente.

5.3. Aritmética.

- Operaciones básicas disponibles en la mayoría de las implementaciones.

Operación	Símbolo
Suma	`+`
Resta	`-`
Multiplicación	`*`
División entera	`div`
Módulo. resto de la división entera	`mod`
División	`/`

- **Ejemplo 66** *Ejemplos de expresiones aritméticas son:*

  i) `3 + 2 * 5`, *ii)* `6 / 2`, *iii)* `3 / 2 + 2 * 5`,
  iv) `3 mod 2`, *v)* `6 div 2`, *vi)* `50 + 10 / 3 mod 2`,
  vi) `X + 10 / 2`.

- En Prolog las expresiones no se evaluan automáticamente y es preciso forzar su evaluación mediante el operador "`is`".

- `?-` *Expresion1* `is` *Expresion2*.
  - Fuerza la evaluación de las expresiones.
  - Las variables deberán estar instanciadas en el momento de la ejecución del objetivo.
  - Si *Expresion1* es una variable, queda instanciada al valor de la *expresion2*.

Aritmética.

- El operador "`is`" no puede utilizarse para producir la instanciación de una variable dentro de una expresión:

```
?- 40 is X + 10 / 2.
Illegal argument to is. _0.
```

- el operador "`is`" puede servir para realizar comparaciones:

```
?- 36.5 is 30 + 13 / 2.
yes
```

- Las expresiones aritméticas se procesan de izquierda a derecha y rigen las siguientes reglas de precedencia:

Precedencia	Operaciones
500	`+`, `-`
400	`*`, `div`, `/`
300	`mod`

  - El operador con mayor número de precedencia en una expresión es el functor principal de la expresión.

  - Los operadores con menor número de precedencia son los que enlazan con más fuerza a sus argumentos.

Aritmética.

- **Ejemplo 67** *Reglas de precedencia y expresiones del Ejemplo 66: i)* `3 + 2 * 5`, *equivale a* `3 + (2 * 5)`*; ii)* `3 / 2 + 2 * 5`, *equivale a* `(3 / 2) + (2 * 5)` *iii)* `50 + 10 / 3 mod 2`, *equivale a* `50 + (10 / (3 mod 2))`.

- Operadores de comparación.

Operación	Símbolo	Significado
Mayor que	`>`	`E1 >E2`. `E1` es mayor que `E2`
Menor que	`<`	`E1 <E2`. `E1` es menor que `E2`
Mayor o igual que	`>=`	`E1 >= E2`. `E1` es mayor o igual que `E2`
Menor o igual que	`=<`	`E1 =<E2`. `E1` es menor o igual que `E2`
Igual que	`=:=`	`E1 =:= E2`. `E1` es igual que `E2`
Distinto que	`=\=`	`E1 =\= E2`. `E1` no es igual que `E2`

- Los operadores de comparación fuerzan la evaluación de las expresiones aritméticas y no producen la instanciación de las variables.

- En el momento de la evaluación todas las variables deberán estar instanciadas a un número.

5.4. Distintas clases de Igualdad y Unificación.

- Igualdad de dos expresiones aritméticas: "`=:=`"

- Igualdad sintáctica de dos expresiones: "`==`". (El operador "`\==`" es la negación de "`==`".)

- El símbolo "`=`" representa la relación de unificación. Se define mediante una única cláusula

  ```
  % X = Y, X unifica con Y
  ```

- El objetivo "`?-` *Expresion1* `=` *Expresion2*." tiene éxito si las expresiones *Expresion1* y *Expresion2* unifican.

- Prolog no comprueba la ocurrencia de variables (*occur check*)

  ```
  ?- X = f(X).
  X = f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(
  f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(
  f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(
  f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(
  f(f(f(f(f(#)))))))))))))))))))))))))))))
  ))))))))))))))))))))))))))))))))))))))
  ```

  Pero no falla.

5.5. El Corte, Control de la Vuelta Atrás y Estructuras de Control.

- En Prolog el control está fijodo automáticamente por la implementación de este lenguaje:
  - recursión,
  - unificación,
  - vuelta atrás automática,

- Prolog proporciona un predicado extralógico, denominado *el corte*, denotado como "!", que permite podar el espacio de búsqueda y previene la vuelta atrás.

- Utilizarlo si se desea optimizar el tiempo de ejecución, la gestión de memoria o evitar soluciones que no interesan.

- El corte siempre tiene éxito y falla al intentar resatisfacerlo.

- Forma de actuación: Cuando al recorrer una rama del árbol de búsqueda se alcanza el corte, todas las posibles alternativas de los puntos de elección entre el *objetivo padre* (el objetivo que lanzó la cláusula que contiene el corte) y la posición en la que se ejecuta el corte son desechadas.

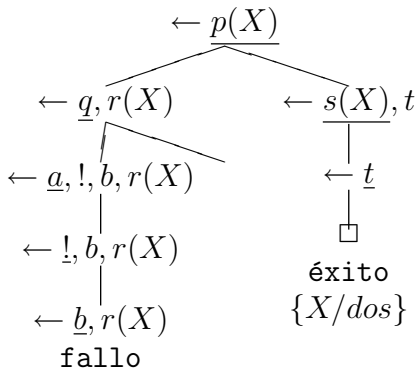
El Corte, Control de la Vuelta Atrás y Estructuras de Control.

- **Ejemplo 68** *Dado el programa*

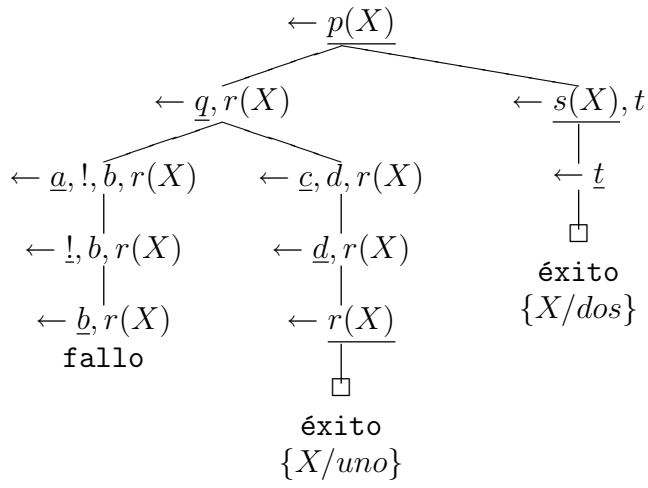
```
p(X) :- q,r(X).     t.
p(X) :- s(X),t.     a.
q :- a,!,b.         b:- fail.
q :- c,d.           c.
r(uno).             d.
s(dos).
```

  - *El árbol para "*`?- p(X).`*" se muestra en la figura.*

- Efecto del corte en el árbol de búsqueda.



Arbol de búsqueda para el objetivo "$\leftarrow p(X)$" y el programa del Ejemplo 68.



Arbol de búsqueda para el objetivo "$\leftarrow p(X)$" cuando se elimina el corte en el programa del Ejemplo 68.

El Corte, Control de la Vuelta Atrás y Estructuras de Control.

- Efectos del corte sobre la eficiencia de los programas.

  Ejemplo 69 *Sea la función escalón:*

$$f(x) = \begin{cases} 0 & si\ (x < 3); \\ 2 & si\ (3 \leq x \wedge x < 6); \\ 4 & si\ (6 \leq x). \end{cases}$$

  Podemos pensar en las siguientes alternativas de programación:

```
% Alternativa 1
f1(X, 0) :- X <3.
f1(X, 2) :- X >= 3, X <6.
f1(X, 4) :- X >= 6.

% Alternativa 2
f2(X, 0) :- X <3, !.
f2(X, 2) :- X >= 3, X <6, !.
f2(X, 4) :- X >= 6.
```

  Comparamos el comportamiento de ambas alternativas ante el objetivo "`?- f(1, Y), 2<Y.`*":*

```
?- trace.
?- f1(1, Y), 2<Y.
<Spy>: (1:0) Call: f1(1,_0). :
<Spy>: |(2:1) Call: 1 <3. :
<Spy>: |(2:1) Exit: 1 <3. :
<Spy>: (1:0) Exit: f1(1,0). :
<Spy>: (1:1) Call: 2 <0. :
```

```
<Spy>: (1:1) Fail: 2 <0. :
<Spy>: (1:0) Redo: f1(1,_0). :
<Spy>: |(2:1) Call: 1 >= 3. :
<Spy>: |(2:1) Fail: 1 >= 3. :
<Spy>: (1:0) Redo: f1(1,_0). :
<Spy>: |(2:1) Call: 1 >= 6. :
<Spy>: |(2:1) Fail: 1 >= 6. :
no

?- f2(1, Y), 2<Y.
<Spy>: (1:0) Call: f2(1,_0). :
<Spy>: |(2:1) Call: 1 <3. :
<Spy>: |(2:1) Exit: 1 <3. :
<Spy>: |(2:1) Call: !. :
<Spy>: |(2:1) Exit: !. :
<Spy>: (1:0) Exit: f2(1,0). :
<Spy>: (1:1) Call: 2 <0. :
<Spy>: (1:1) Fail: 2 <0. :
no
```

- Cuando las reglas son mutuamente excluyentes resulta útil no permitir la vuelta atrás haciendo uso del corte.

- La versión 2 resulta mucho mas eficiente, por la poda de ramas que sabemos con seguridad que conducirán a un fallo.

El Corte, Control de la Vuelta Atrás y Estructuras de Control.

- Se han identificado dos tipos de cortes:
  - Los *cortes verdes*, que no afectan al significado declarativo del programa.
  - Los *cortes rojos*, que sí afectan al significado declarativo del programa original (Puede que se pierdan respuestas).

- **Ejemplo 70**
  - *El corte introducido en el programa del Ejemplo 68 es rojo(se elimina la respuesta $\{X/uno\}$).*
  - *El corte introducido en la alternativa 2 del Ejemplo 69 es verde.*

- El corte, en general, y los cortes rojos, en particular, deben utilizarse con sumo cuidado

- El corte hace ilegible los programas, al impedir la lectura declarativa de los mismos.

- El corte juega el mismo papel que la construcción "`goto`" en los lenguajes imperativos.

- La regla es "evitar el corte siempre que sea posible".

El Corte, Control de la Vuelta Atrás y Estructuras de Control.

- Prolog incorpora otras facilidades de control:
  - `true`, el objetivo que siempre tiene éxito.
  - `fail`, el objetivo que siempre falla, forzando la vuelta atrás.

- **Ejemplo 71** *Dada la pequeña base de datos de paises*

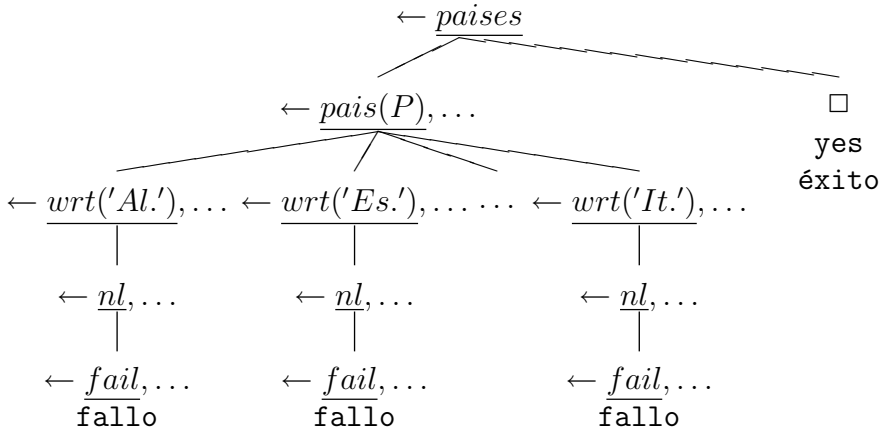
```
pais('Alemania').   pais('Gran Bretaña').
pais('España').     pais('Italia').
pais('Francia').
```

El predicado paises sirve para listar su contenido

```
paises :- pais(P), write(P), nl, fail.
paises.
```

  - *El papel de* `fail` *es esencial para lograr este efecto.*

  - *El listado de los nombres es un efecto lateral de la definición del predicado* `paises`*. La respuesta al objetivo "*`?- paises.`*" es "*`yes.`*".*



El Corte, Control de la Vuelta Atrás y Estructuras de Control.

- La combinación del corte y los objetivos `fail` y `true` pueden emplearse para simular algunas estructuras de control de los lenguajes convencionales.

- La construcción `if_then_else`:

```
% version 1
% ifThenElse1(Condicion, Accion1, Accion2),
% if Condicion then Accion1 else Accion2
ifThenElse1(C, A1, _) :- C, !, A1.
ifThenElse1(_, _, A2) :- A2.
```

- En esta definición las variables `C`, `A1` y `A2` son metavariables que pueden sustituirse por cualquier predicado.

- Una versión más compacta pero totalmente equivalente:

```
% version 1
% ifThenElse1(Condicion, Accion1, Accion2),
% if Condicion then Accion1 else Accion2
ifThenElse1(C, A1, A2) :- C, !, A1; A2.
```

- Una forma más declarativa (usando `not`):

```
% version 2
% ifThenElse2(Condicion, Accion1, Accion2),
% if Condicion then Accion1 else Accion2
ifThenElse2(C, A1, _) :- C, A1.
ifThenElse2(C, _, A2) :- not(C), A2.
```

  menos eficientemente: `Condicion` se comprueba 2 veces.

- El comportamiento operacional es distinto si `!` va al final:

```
% version 3
% ifThenElse3(Condicion, Accion1, Accion2),
% if Condicion then Accion1 else Accion2
ifThenElse3(C, A1, _) :- C, A1, !.
ifThenElse3(_, _, A2) :- A2.
```

- **Ejemplo 72** *Sea el programa*

```
% Un programa tonto para testear las diferentes
% versiones del predicado ifThenElse
acc1(X) :- X = 1.
acc1(X) :- X = 2.
acc2(X) :- X = 3.
acc2(X) :- X = 4.
```

  ampliado con las definiciones del predicado `ifThenElse`.

El Corte, Control de la Vuelta Atrás y Estructuras de Control.

Una sesión en un intérprete de Prolog produce estos resultados:

```
?- ifThenElse1(true, acc1(X), acc2(Y)).
X = 1, Y = _1 ;
X = 2, Y = _1 ;
no
?- ifThenElse1(fail, acc1(X), acc2(Y)).
X = _0, Y = 3 ;
X = _0, Y = 4 ;
no
?- ifThenElse2(true, acc1(X), acc2(Y)).
X = 1, Y = _1 ;
X = 2, Y = _1 ;
no
?- ifThenElse2(fail, acc1(X), acc2(Y)).
X = _0, Y = 3 ;
X = _0, Y = 4 ;
no
?- ifThenElse3(true, acc1(X), acc2(Y)).
X = 1, Y = _1 ;
no
?- ifThenElse3(fail, acc1(X), acc2(Y)).
X = _0, Y = 3 ;
X = _0, Y = 4 ;
no
```

El Corte, Control de la Vuelta Atrás y Estructuras de Control.

- En muchos de los sistema Prolog la construcción `if_then_else` es un operador predefinido.

- "`Condicion ->Accion1; Accion2`" debe leerse como "`if Condicion then Accion1 else Accion2`".

- El operador "`->`" está definido conforme a la versión 1 del predicado `ifThenElse`.

- Es equivalente a la construcción "`(Condicion, !, Accion1; Accion2)`".

- El operador "`->`" puede encadenarse para simular la construcción case de los lenguajes convencionales.

```
case :- ( Condicion1 ->Accion1;
Condicion2 ->Accion2;
⋮
CondicionN ->AccionN;
OtroCaso).
```

El Corte, Control de la Vuelta Atrás y Estructuras de Control.

- Es una constucción más clara y fácil de entender que la equivalente

```
case :- ( Condicion1, !, Accion1;
Condicion2, !, Accion2;
⋮
CondicionN, !, AccionN;
OtroCaso).
```

  en la que se emplea el corte directamente.

- El predicado predefinido `repeat`. Este predicado se comporta como si estuviese definido por las siguientes cláusulas:

```
repeat.
repeat :- repeat.
```

- Un ejemplo de uso:

```
hacer_tratamiento :-
repeat,
read(X),
( X = stop, !
;
tratamiento(X),
fail
).
```

El Corte, Control de la Vuelta Atrás y Estructuras de Control.

- Otros predicados predefinidos para el control:

  1. `quit` o `abort`, que abortan cualquier objetivo activo, devolviendo el control al sistema (i.e., volvemos al prompt del sistema "`?-`").

  2. `exit` o `halt` nos sacan del propio intérprete, terminando la sesión en curso.

- **Observaciones 5.5.1**

  1. *Como se ha comentado el corte es como la construcción "*`goto`*" de los lenguajes convencionales. La mejor forma de usarlo es confinandolo en el interior de operadores que admitan una lectura más clara (e.g., "*`not`*" y "*`->`*").*
  2. *El corte introduce una visión procedural que, en general, no resulta positiva, ya que suele provocar que se utilice Prolog como un lenguaje imperativo.*

5.6. Negación.

5.6.1. El problema de la informción negativa.

- Para programas definidos es imposible extraer información negativa utilizando resolución SLD.

- **Proposición 5.6.1** *Sea* Π *un programa definido y* $\mathcal{A} \in \mathcal{B}_{\mathcal{L}}(\Pi)$. *Se cumplen los siguientes enunciados equivalentes:*

  1. *El literal* $\neg\mathcal{A}$ *no es conseguencia lógica de* Π.
  2. *El conjunto* $\Pi \cup \{\mathcal{A}\}$ *es satisfacible.*
  3. *No existe una refutación SLD para* $\Pi \cup \{\mathcal{A}\}$.

- Pueden alegarse dos razones a la hora de justificar la necesidad de tratar el problema de la negación:

  1. Dado el programas definidos.

     $$carnívoro(gato) \leftarrow, \quad hervívoro(oveja) \leftarrow,$$
     $$carnívoro(perro) \leftarrow, \quad hervívoro(vaca) \leftarrow,$$
     $$hervívoro(cabra) \leftarrow, \quad hervívoro(conejo) \leftarrow$$

     Sería interesante que informaciones como

     $$hervívoro(gato) \leftarrow, \quad hervívoro(perro) \leftarrow$$

     que no se suministran en el programa, pudiesen interpretarse como una afirmación de:

     $$\neg hervívoro(gato) \leftarrow, \quad \neg hervívoro(perro) \leftarrow,$$

  2. Es preciso dotar a los programas de una mayor expresividad $\Longrightarrow$ uso de cláusulas y programas normales.

5.6.2. La negación en la programación lógica.

- El problema de la extracción de información negativa se trata ampliando el sistema de inferencia con una nueva regla.

- La nueva regla se utiliza para obtener una caracterización adicional que completa el significado de un programa definido: *semántica de la negación*.

- **Suposición de un mundo cerrado (CWA).**
  - Es una caracterización de la negación surgida en el ámbito de las bases de datos deductivas (Reiter 1978).
  - Punto de vista operacional:
    $$\frac{(\mathcal{G} \equiv \leftarrow \mathcal{Q}_1 \wedge \neg\mathcal{A} \wedge \mathcal{Q}_2), \mathcal{A} \in \mathcal{B}_{\mathcal{L}}(\Pi), \Pi \nvdash_{SLD} \mathcal{A}}{\mathcal{G} \Longrightarrow_{CWA} \leftarrow \mathcal{Q}_1 \wedge \mathcal{Q}_2}$$
  - Punto de vista declarativo: Para un programa Π y un átomo $\mathcal{A} \in \mathcal{B}_{\mathcal{L}}(\Pi)$, podemos afirmar $\neg\mathcal{A}$ si $\mathcal{A}$ no es consecuencia lógica del programa Π.
  - Semántica de la negación (CWA):
    $\mathcal{CWA}(\Pi) = \{\neg\mathcal{A} \mid (\mathcal{A} \in \mathcal{B}_{\mathcal{L}}(\Pi)) \wedge (\Pi \cup \{\leftarrow \mathcal{A}\} \nvdash_{SLD} \square)\}$.
  - Semántica de un programa definido Π: $\mathcal{EB}(\Pi) \cup \mathcal{CWA}(\Pi)$.

Negación.

- Problemas con la CWA:
  - Conduce a errores cuando no se cumple que todos los hechos positivos relevantes para el problema están en la base de datos.

  - $\Pi \cup \mathcal{CWA}(\Pi)$ puede ser un programa inconsistente cuando Π es un programa normal (o general).

    Ejemplo 73 *Sea* $\Pi = \{p \leftarrow \neg q\}$.
    $\mathcal{CWA}(\Pi) = \{\neg p, \neg q\}$.
    Pero
    $\{p \leftarrow \neg q, \neg p, \neg q\}$
    es un conjunto insatisfacible.

- **Negación como fallo (NAF).**
  - Es una restricción de la regla de inferencia CWA.

  - Una refutación SLD para $\Pi \cup \{\leftarrow \mathcal{A}\}$ no existir debido a que:
    1. tiene un árbol SLD que falla finitamente, o bien
    2. tiene un árbol SLD infinito.

Negación.

- En el último de los casos, vemos que la CWA no es efectiva desde el punto de vista operacional.

- No se podría decir nada sobre el éxito o el fracaso de la refutación.

- Para evitar este problema se introdujo la regla de NAF (Clark 1978) para extraer información negativa de un programa.

$$\frac{(\mathcal{G} \equiv \leftarrow \mathcal{Q}_1 \wedge \neg\mathcal{A} \wedge \mathcal{Q}_2), \mathcal{A} \in \mathcal{B}_{\mathcal{L}}(\Pi), \exists \tau_{\varphi}^{FF}}{\mathcal{G}, \Longrightarrow_{NAF} \leftarrow \mathcal{Q}_1 \wedge \mathcal{Q}_2}$$

  τ_{φ}^{FF}: árbol SLD de fallo finito para $\Pi \cup \{\leftarrow \mathcal{A}\}$.

- Semántica de la negación (NAF):

$$\mathcal{NAF}(\Pi) = \{\neg\mathcal{A} \mid (\mathcal{A} \in \mathcal{B}_{\mathcal{L}}(\Pi)) \wedge \exists \tau_{\varphi}^{FF}\}.$$

- Semántica de un programa definido Π: $\mathcal{EB}(\Pi) \cup \mathcal{NAF}(\Pi)$.

- Los problemas respecto a la consistencia son los mismos que en el caso de la CWA.

Negación.

- **resolución SLDNF:**
  Estrategia de resolución SLD + regla de NAF = estrategia de resolución SLDNF.

- La estrategia de resolución SLDNF es la base de los sistemas actuales de programación lógica (e.g. Prolog).

- La estrategia de resolución SLDNF procede del siguiente modo:
  - cuando la regla de computación selecciona un literal positivo, entonces se utiliza la estrategia de resolución SLD para derivar un nuevo objetivo;
  - si el literal seleccionado es un **átomo básico** negativo, se desencadena un proceso recursivo para aplicar la regla de NAF;
  - si el átomo negado seleccionado no es básico, el cómputo debe detenerse.

5.6.3. La negación en el Lenguaje Prolog.

- Prolog proporciona una forma de negación basada en NAF.

- Se utiliza el operador unario "`not`", que es un predicado predefinido, haciendo uso del corte:

```
not(A) :- A, !, fail.
not(A) :- true.
```

- Observación: `A` es una metavariable.

- `not(A)` falla si `A` tiene éxito y, por contra, `not(A)` tiene éxito si `A` falla.

- Esta definición no se corresponde exáctamente con la regla de NAF:
  - no se controla si el átomo `A` es básico $\Longrightarrow$ un cómputo prosigue aunque sólo los átomos negados no sean básicos.

  - La regla de computación de Prolog puede dar lugar a resultados erroneos aún cuando el átomo negado sea básico.

    Ejemplo 74 *Dado el programa*

    ```
    q(a) :- p(b), q(b).
    p(b) :- p(b).
    ```

    y el objetivo "`?- not q(a).`*"*

5.7. Entrada/Salida.

- Los predicados predefinidos de E/S son dependientes de la implementación del sistema Prolog.

- Se estudia una muestra de los predicados de E/S disponibles en las implementaciones que siguen la sintaxis de Edimburgo.

- **Flujos de E/S.**
  - La visión que suministra Prolog de los ficheros es la de una secuencia de objetos (átomos, términos, o simplemente caracteres).

  - Un programa Prolog puede leer datos de diferentes flujos de entrada y escribir datos en diferentes flujos de salida.

  - Pero, en cada instante, sólo están abiertos dos flujos:

    1. flujo de entrada estándar (CIS–Current Input Stream);

    2. flujo de salida estándar (COS–Current Output Stream).

Entrada/Salida.

- Los nombres de los ficheros se eligen siguiendo las normas sintácticas del sistema operativo sobre el que se ejecuta el intérprete de Prolog.

- El teclado y la pantalla son tratados como un único flujo, que recibe el nombre de "`user`".

- Cuando se inicia una sesión con un sistema Prolog, el CIS es el teclado y el COS la pantalla, que no necesitan abrirse.

- Las operaciones típicas cuando se trabaja con ficheros son:

  1. **abrir** el fichero para leer o escribir;

  2. realizar las acciones de lectura o escritura pertinentes;

  3. **cerrar** el fichero.

- No se puede tener un mismo fichero abierto para lectura y escritura (a excepción del `user`).

Entrada/Salida.

▪ Predicados de E/S.

- Características generales de los predicados de E/S:
  1. cuando se evalúan siempre tienen éxito;
  2. nunca se pueden resatisfacer;
  3. tienen como efecto lateral la E/S de un carácter, un término, etc.

- **Manipulación de ficheros.**
  - `see(Fich)`.
  - `seeing(Fich)`.
  - `seen`.
  - `tell(Fich)`.
  - `telling(Fich)`.
  - `told`.

- **Lectura y escritura de términos.**
  - `read(Term)`.
  - `write(Term)`.
  - `display(Term)`.

- **Lectura y escritura de caracteres.**
  - `get0(Char)`.
  - `get(Char)` (sólo lee los caracteres imprimibles).
  - `skip(X)`.
  - `put(Char)`.
  - El predicado `nl` y el predicado `tab(Num)`.

Entrada/Salida.

- **El problema de las ocho reinas.**

```
% vector(Sol), verdadero si las reinas no se atacan
vector(Sol) :- Sol = [c(1, _), c(2, _), c(3, _), c(4, _), c(5, _),
c(6, _), c(7, _), c(8, _)], solucion(Sol).

% solucion(L), L es una lista con posiciones desde las que
% las reinas no pueden atacarse
solucion([]).
solucion([c(X, Y)|Resto]) :- solucion(Resto),
member(Y, [1, 2, 3, 4, 5, 6, 7, 8]), not ataca(c(X, Y), Resto).

% ataca(c(X, Y), L), la reina en c(X, Y) ataca a alguna de las
% situadas en las posiciones de L
ataca(c(_, Y), [c(_, Y1)|_]) :-
        Y =:= Y1, !.% misma horizontal
ataca(c(X, Y), [c(X1, Y1)|_]) :-
        Y1 is X1 + (Y - X), !.% misma diagonal (pend = 1)
ataca(c(X, Y), [c(X1, Y1)|_]) :-
        Y1 is (Y + X) - X1, !.% misma diagonal (pend = -1)
ataca(c(X, Y), [c(_, _)|Resto]) :-
        ataca(c(X, Y), Resto).% en otro caso

% muestra el tablero y la solucion
tablero(Sol) :- vector(Sol), esc_cabecera, esc_tablero(Sol).

esc_cabecera :- nl, nl, nl,
   tab(3), write('---------------------------------'), nl,
   tab(3), write('! 1 ! 2 ! 3 ! 4 ! 5 ! 6 ! 7 ! 8 !'), nl,
   tab(3), write('---------------------------------'), nl.

esc_tablero([]).
esc_tablero([c(N, R)|Resto]) :- esc_fila(N, R), esc_tablero(Resto).

esc_fila(N, R) :- tab(2), write(N), esc_casillas(R), nl,
   tab(3), write('---------------------------------'), nl.

esc_casillas(1) :- write('! * ! ! ! ! ! ! !').
esc_casillas(2) :- write('! ! * ! ! ! ! ! !').
esc_casillas(3) :- write('! ! ! * ! ! ! ! !').
esc_casillas(4) :- write('! ! ! ! * ! ! ! !').
esc_casillas(5) :- write('! ! ! ! ! * ! ! !').
esc_casillas(6) :- write('! ! ! ! ! ! * ! !').
esc_casillas(7) :- write('! ! ! ! ! ! ! * !').
esc_casillas(8) :- write('! ! ! ! ! ! ! ! *').
```

5.8. Otros predicados predefinidos.

5.8.1. Predicados predefinidos para la catalogación y construcción de términos.

- **Catalogación de términos.**
  - `var(X).`
  - `nonvar(X).`
  - `atom(X).`
  - `integer(X).`
  - `atomic(X).`
- **Construcción de términos.**
  - `functor(T,F,N).`
  - `arg(N,T,A).`
  - `E =.. L.`
  - `name(A,L).`

5.8.2. Predicados predefinidos para la manipulación de cláusulas y la metaprogramación.

- **Carga de programas.**
  - `consult(Fich)`.
  - `reconsult(Fich)`.
- **Cosulta.**
  - `listing(Nom)`.
- **Adición y supresión.**
  - `asserta(C)`.
  - `assertz(C)` (añade la cláusula `C` al final).
  - `retract(C)`.
- **Metaprogramación.**
  Estas facilidades suponen la extensión del lenguaje clausal con características de *orden superior*.
  - `call(Obj)`.
  - `clause(Cabeza,Cuerpo)`.

5.8.3. El predicado `setof`.

- Estrategias de "*una respuesta cada vez*" (*tuple-at-a-time*).

- Estrategias de "*un conjunto de respuestas cada vez*" (*set-at-a-time*).

- El predicado `setof`:
  `setof(Term, Obj, Lista)` tiene éxito si `Lista` es la lista de instancias ordenadas y sin repeticiones del término `Term` para las que el objetivo `Obj` tiene éxito.

- **Ejemplo 75** *Una sesión de trabajo con el programa de las relaciones familiares del Apartado 5.2:*

  - *Buscar todos los hijos de* `haran`.

    ```
    ?- setof(Hijo, padre(haran, Hijo), LHijos).
    Hijo = _0, LHijos = [jesca,lot,melca] ;
    no
    ```

    Nótese que la pregunta no puede plantearse en estos términos,

    ```
    ?- setof(_, padre(haran, _), LHijos).
    LHijos = [_4] ;
    LHijos = [_4] ;
    LHijos = [_4] ;
    no
    ```

Otros predicados predefinidos.

- *Buscar todos los ascendientes de* `isaac`.

```
?- setof(Ascend, ascendiente(Ascend, isaac), LAscend).
Ascend = _0, LAscend = [abraham,sara,teraj] ;
no
```

- *Buscar todos los ascendientes de* `lot`.

```
?- setof(Ascend, ascendiente(Ascend, lot), LAscend).
Ascend = _0, LAscend = [haran,teraj] ;
no
```

- *Buscar todos los ascendientes de* `isaac` *que no son de* `lot`.

```
?- setof(Ascend,
(ascendiente(Ascend, isaac), not ascendiente(Ascend, lot)),
LAscend).
Ascend = _0, LAscend = [haran,teraj] ;
no
```

- *Buscar todos los ascendientes de* `isaac` *que no son de* `lot`.

```
?- setof(Ascend,
(ascendiente(Ascend, isaac), not ascendiente(Ascend, lot)),
LAscend).
Ascend = _0, LAscend = [abraham,sara] ;
no
```

- *Buscar todos los ascendientes de* `isaac` *o de* `lot`.

```
?- setof(Ascend,
(ascendiente(Ascend, isaac); ascendiente(Ascend, lot)),
LAscend).
Ascend = _0, LAscend = [abraham,haran,sara,teraj] ;
no
```

Otros predicados predefinidos.

- *Buscar todos los hijos de cada padre.*

```
?- setof(Hijo, padre(Padre, Hijo), LHijos).
Hijo = _0, Padre = abraham, LHijos = [isaac,ismael] ;
Hijo = _0, Padre = batuel, LHijos = [laban,rebeca] ;
Hijo = _0, Padre = haran, LHijos = [jesca,lot,melca] ;
Hijo = _0, Padre = isaac, LHijos = [esau,jacob] ;
Hijo = _0, Padre = najor, LHijos = [batuel] ;
Hijo = _0, Padre = teraj, LHijos = [abraham,haran,najor,sara] ;
no
```

Si queremos disponer todos los hijos en una lista, independientemente del padre que los haya procreado, esta sería la forma.

```
setof(Hijo, Padre ^ padre(Padre, Hijo), LHijos).
Hijo = _0, Padre = _1,
LHijos = [abraham,batuel,esau,haran,isaac,ismael,jacob,
jesca,laban,lot,melca,najor,rebeca,sara] ;
no
```

El operador predefinido "`^`*" sirve para formar una conjunción de objetivos.*

El objetivo anterior admite esta lectura: " buscar todos los `Hijo`*'s tales que* `padre(Padre, Hijo)` *es verdadero para algún* `Padre`*". Así que, "*`Padre ^`*" puede interpretarse como "existe un* `Padre`*".*

Capítulo 6

Aplicaciones de la programación lógica.

- El objetivo de este capítulo es:
  1. Presentar algunas de las aplicaciones de la programación lógica.
  2. Introducir nuevas técnicas de programación.

- Se han elegido la representación del conocimiénto y la resolución de problemas.

- Estas han sido tema de estudio desde los primeros tiempos en el área de la *Inteligencia Artificial* (IA).

- La IA es una parte de la informática que investiga como construir programas que realicen tareas inteligentes.

6.1. Representación del conocimiento.

6.1.1. El problema de la representación del conocimiento.

- **conocimiento**: conjunto de informaciones referentes a un área que permiten a un experto resolver problemas (no triviales) relativos a ese área.

- Un sistema que resuelva un problema en un determinado dominio diremos que tiene conocimiento en ese dominio.

- Estamos interesados en las formas de representar el conocimiento (de un agente humano) en un computador.

- Una representación del conocimiento es una descripción formal que puede emplearse para la computación simbólica

- Se desea que esa descripción pueda producir un comportamiento inteligente.

Representación del conocimiento.

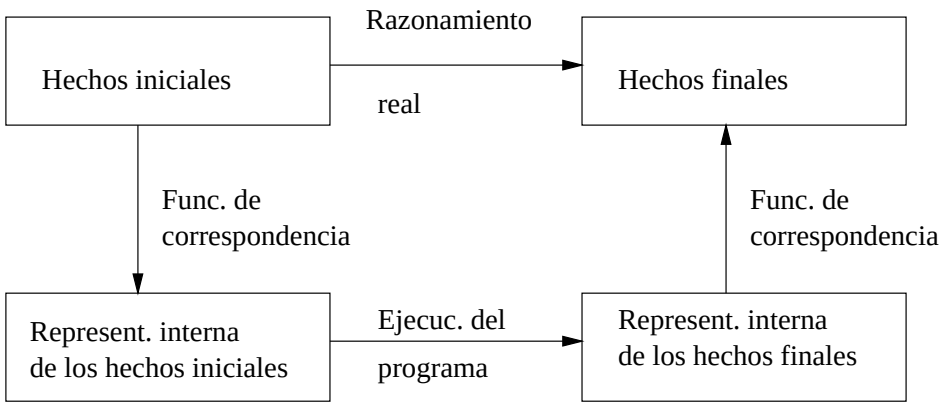
- Una característica en todas las representaciones del conocimiento es que manejamos dos tipos de entidades:
  1. los hechos, verdades de un cierto mundo que queremos representar;
  2. la representación de los hechos en un determinado formalismo.

- Estas entidades inducen la necesidad de distinguir entre dos niveles [Newell 1982]:
  1. el *nivel del conocimiento*, donde se describen los hechos y el comportamiento;
  2. el *nivel simbólico*, donde se describen los objetos del nivel del conocimiento mediante símbolos manipulables por programas.

- Siguiendo la orientación de [Rich y Knight, 1994] nos centraremos en los hechos, en las representaciones y en la correspondencia que debe existir entre ellos.

Representación del conocimiento.

- El razonamiento real es simulado por el funcionamiento de los programas, que manipulan las representaciones internas de los hechos.



- Son necesarias *funciones de correspondencia* para:

  1. dado un hecho inicial dotarlo de una representación interna y

  2. dado una representación interna final establecer el hecho final correspondiente.

- `Problema`: las funciones de correspondencia no suelen ser biunívocas.

Representación del conocimiento.

- Un buen sistema de representación del conocimiento, de un dominio particular, debe poseer las siguientes propiedades:
  1. Suficiencia de la representación: la capacidad de representar todos los tipos de conocimiento necesarios en el dominio.
     ("un león es un carnívoro" versus "un animal que come otros animales es un carnívoro")
  2. Suficiencia deductiva: la capacidad de manipular las representaciones internas de los hechos con el fin de obtener otros nuevos.
  3. Eficiencia deductiva: la capacidad de incorporar información (de control) adicional en las estructuras de conocimiento con el fin de guiar el proceso deductivo en la dirección más adecuada.
     $\Longrightarrow$ Conocimiento experto y heurística.
  4. Eficiencia en la adquisición: la capacidad de adquirir nueva información con facilidad.
     $\Longrightarrow$ Sencillez en la notación empleada.

- No existe una técnica de representación del conocimiento que cumpla todos estos objetivos. Unas son mejores que otras según para que cosas.

- Lenguajes para la representación: los lenguajes declarativos disponen de una sintaxis formal y una semántica clara, así como capacidades deductivas
  $\Longrightarrow$ bien adaptados como lenguajes de representación (simbólica).

6.1.2. Representación mediante la lógica de predicados.

- Consiste en transformar el conocimiento sobre un dominio concreto en fórmulas bien formadas.

- Características que la hacen atractiva:

  1. Es una forma natural de expresar relaciones.

  2. Es precisa, ya que existen métodos formales para determinar el significado y la validez de una fórmula.
  3. Posee diferentes cálculos deductivos que permiten la construcción de derivaciones (Punto fuerte).

  4. Existen resultados de corrección y completitud.

  5. Puede automatizarse el proceso deductivo de una forma flexible, lo que facilita el uso de la lógica de predicados como lenguaje de programación.

  6. Es modular: puede aumentarse la teoría sin afectar la validez de los resultados anteriormente deducidos.

- La mayoría de las veces, el conocimiento sobre un dominio está expresado en un lenguaje natural.
  (Representar el conocimiento $\Longrightarrow$ traducción al lenguaje de la lógica de predicados)

- `Traducción`:
  - Es una tarea inversa a la de interpretar.
  - Hay que partir de una interpretación. Después, se sustituye cada parte del enunciado del lenguaje natural por el símbolo que lo representa.

- `Problemas`:
  - No hay reglas precisas.
  - El lenguaje natural es ambiguo $\Longrightarrow$ para un enunciado del lenguaje natural pueden haber varias representaciones.

- **Consejos prácticos para la traducción**:
  - `Regla de oro`:
    1. meditar detenidamente sobre el sentido del enunciado del lenguaje natural;
    2. seleccionar una interpretación de partida $\mathcal{I}$ que sea decuada y facilite la posterior representación del enunciado;
    3. derivar una fórmula, que cuando se interprete tenga un significado lo más próximo posible al enunciado original.

Representación del conocimiento.

- En las próximas páginas se presenta un recordatorio de material ya conocido.

  - `Nombres pronombres y designadores compuestos` se traducen, respectivamente, por constantes, variables y términos encabezados por un símbolo de función.

  - Al representar los pronombres por variables puede hacer que se pierda información.

    (Solución: escribir "pascual" en vez de "X", si es Pascual quien dijo "Yo he ido al cine").

  - `Los relatores` se traducen por fórmulas atómicas.
    "Juan es más alto que Pascual" $\Longrightarrow masAlto(juan, pascual)$.

    Convención: El símbolo de relación se emplea en forma prefija y se toma como primer argumento al sujeto de la oración.

  - Los relatores monarios tradicionalmente se denominan *predicados* (y coinciden con la parte de la oración gramatical de ese nombre).

    Notad que los predicados pueden usarse para definir conjuntos (clases). Por ejemplo: $par(X)$.

  - `Las conjunciones` se traducen por conectivas (lógicas).

    1. Negación ($\neg$). La fórmula $\neg\mathcal{A}$ permite simbolizar enunciados del tipo: *no* $\mathcal{A}$; *ni* $\mathcal{A}$; *no es cierto que* $\mathcal{A}$; *es falso que* $\mathcal{A}$.

    2. Conjunción ($\wedge$). La fórmula $\mathcal{A} \wedge \mathcal{B}$, simboliza enunciados del lenguaje natural de la forma: $\mathcal{A}$ *y* $\mathcal{B}$; $\mathcal{A}$ *y también* $\mathcal{B}$; $\mathcal{A}$ *e igualmente* $\mathcal{B}$; *tanto* $\mathcal{A}$ *como* $\mathcal{B}$; $\mathcal{A}$ *pero* $\mathcal{B}$; $\mathcal{A}$ *no obstante* $\mathcal{B}$; $\mathcal{A}$ *sin embargo* $\mathcal{B}$.

    3. La partícula "*ni*": se presenta en términos de una conjunción de negaciones
       ("*ni* $\mathcal{A}$ *ni* $\mathcal{B}$" =¿"$\neg\mathcal{A} \wedge \neg\mathcal{B}$".

    4. Disyunción ($\vee$). La fórmula $\mathcal{A} \vee \mathcal{B}$ simboliza enunciados de la forma: $\mathcal{A}$ *o* $\mathcal{B}$; *al menos* $\mathcal{A}$ *o* $\mathcal{B}$.

    5. Condicional ($\rightarrow$). La fórmula $\mathcal{A} \rightarrow \mathcal{B}$ simboliza enunciados de la forma: *si* $\mathcal{A}$ *entonces* $\mathcal{B}$; *si* $\mathcal{A}$, $\mathcal{B}$; $\mathcal{A}$ *implica* $\mathcal{B}$; $\mathcal{A}$ *sólo si* $\mathcal{B}$; $\mathcal{A}$ *suficiente para* $\mathcal{B}$; $\mathcal{B}$ *si* $\mathcal{A}$; $\mathcal{B}$ *necesario para* $\mathcal{A}$; *no* $\mathcal{A}$ *a menos que* $\mathcal{B}$; $\mathcal{B}$ *cuandoquiera que* $\mathcal{A}$; $\mathcal{B}$ *siempre que* $\mathcal{A}$.

6. Bicondicional($\leftrightarrow$). $\mathcal{A} \leftrightarrow \mathcal{B}$ denota enunciados de la forma: $\mathcal{A}$ *si y sólo si* $\mathcal{B}$; $\mathcal{A}$ *necesario y suficiente para* $\mathcal{B}$

- **Ejemplo 76**
  $\mathcal{I} = \langle \mathcal{D}_\mathcal{I}, \mathcal{J} \rangle$ *donde* $\mathcal{D}_\mathcal{I} = \mathbb{N}$ *y la función de interpretación* $\mathcal{J}$ *asigna:*

$par(X):$	X *es par;*
$impar(X):$	X *es impar;*
$primo(X):$	X *es primo;*
$X = Y:$	X *es identico a* Y*,*
$X < Y:$	X *es menor que* Y*;*
$X > Y:$	X *es mayor que* Y*;*
$suma(X, Y, S):$	$X + Y = S$;
$mult(X, Y, M):$	$X \times Y = M$;

  y, por simplicidad, cada número natural es representado por él mismo.

  Ahora podemos traducir los siguientes enunciados del lenguaje natural al lenguaje formal:

  - *"2 es par":* $par(2)$.
  - *"2 no es impar":* $\neg impar(2)$.
  - *"es un número impar y mayor que tres":* $impar(X) \wedge X > 3$.
  - *"no es el caso de que 5 sea el producto de 2 por 3":* $\neg mult(2, 3, 5)$.
  - *"ser primo es suficiente para ser impar":* $primo(X) \rightarrow impar(X)$.
  - *"2 o es par o es impar":* $(par(2) \vee impar(2)) \wedge \neg(par(2) \wedge impar(2))$.

- **Ejemplo 77**
  $\mathcal{I} = \langle \mathcal{D}_\mathcal{I}, \mathcal{J} \rangle$ *donde* $\mathcal{D}_\mathcal{I}$ *es el conjunto de los seres humanos y* $\mathcal{J}$ *asigna:*

$padre(X, Y):$	X *es el padre de* Y*;*
$madre(X, Y):$	X *es la madre de* Y*;*
$marido(X, Y):$	X *es el marido de* Y*;*
$hermano(X, Y):$	X *es hermano de* Y*;*
$hermana(X, Y):$	X *es hermana de* Y*;*
$m:$	*María;*
$e:$	*Enrique;*
$g:$	*Guillermo;*
$a:$	*Arturo;*

  Ahora podemos traducir los siguientes enunciados del lenguaje natural al lenguaje formal:

  - *"Enrique es padre":* $padre(e, X)$.
  - *"María es abuela":* $madre(m, X) \wedge (padre(X, Y) \vee madre(X, Y))$.
  - *"Guillermo es el cuñado de María":* $(marido(g, X) \wedge hermana(X, m))$.

- *"María es la tía de Arturo":*
  $[(marido(Z, m) \wedge hermano(Z, W)) \vee hermana(m, W)] \wedge (padre(W, a) \vee madre(W, a))$.

- `Granularidad y representación canónica`
  1. La traducción de un enunciado depende del vocabulario introducido en la interpretación de la que se parte.
  2. Si ampliamos la interpretación $\mathcal{I}$ con el símbolo de relación "$esPadre(X)$", la traducción del enunciado habría sido: $esPadre(e)$.
  3. `Granularidad`: el conjunto de primitivas utilizadas en el vocabulario para representar el conocimiento.
  4. Una interpretación proporciona un conjunto de primitivas con el que representar el conocimiento de una forma canónica.

- Una traducción más precisa de los enunciados del Ejemplo 77, requiere utilizar el cuantificador existencial.

- `Cuantificadores`:
  - Los enunciados del lenguaje natural que se basan en el uso de la partícula "todos" se representan empleando el *cuantificador universal* "$\forall$".
  - Otras partículas que conllevan el uso del cuantificador universal son: "cada", "cualquier", "cualquiera", "todo", "toda", "nada", "ningún", "ninguno", "ninguna".
  - Los enunciados del lenguaje natural que se basan en el empleo de "algunos" se representan empleando el *cuantificador existencial* "$\exists$".
  - Otras partículas que conllevan el uso del cuantificador existencial son: "existe", "hay", "algún", "alguno", "alguna", "algunas".

Enunciado	**Formalización**
Todo $\mathcal{A}$ es $\mathcal{B}$	$(\forall X)(\mathcal{A}(X) \rightarrow \mathcal{B}(X))$
Algún $\mathcal{A}$ es $\mathcal{B}$	$(\exists X)(\mathcal{A}(X) \wedge \mathcal{B}(X))$
Ningún $\mathcal{A}$ es $\mathcal{B}$	$(\forall X)(\mathcal{A}(X) \rightarrow \neg\mathcal{B}(X))$
Algún $\mathcal{A}$ no es $\mathcal{B}$	$(\exists X)(\mathcal{A}(X) \wedge \neg\mathcal{B}(X))$

- Los artículos determinados "el", "los", etc. pueden expresar cuantificación universal.
  (e.g., "el hombre es un mamifero" o "Los gatos persiguen a los ratones")

- Los artículos indeterminados "un", "unos", etc. pueden expresar cuantificación existencial.
  (e.g., "unos hombres son mejores que otros")

- Tambíen las palabras "tiene" y "tienen" expresan exitencia.
  (e.g., "Juan tiene un progenitor que le ama")

▪ **Ejemplo 78** *Consideramos la interpretación $\mathcal{I}$ del Ejemplo 76:*

- *"existe un número natural mayor que (cualquier) otro dado":*

  $$(\forall X)(existsY)(Y > X).$$

- *"0 es un número natural menor que (cualquier) otro":*

  $$(\forall X)(\neg(X = 0) \rightarrow 0 < X).$$

- *"Si el producto de dos números es par, entonces al menos uno de ellos es par":*

  $$(\forall X)(\forall Y)[(\exists Z)((mult(X, Y, Z) \wedge par(Z)) \rightarrow (par(X) \vee par(Y))]$$

  o también

  $$(\forall X)(\forall Y)(\forall Z)[(mult(X, Y, Z) \wedge par(Z)) \rightarrow (par(X) \vee par(Y))].$$

- *"Todo entero par mayor que 4 es la suma de dos primos":*

  $$(\forall Z)[(par(Z) \wedge Z > 4) \rightarrow (\exists X)(\exists X)(suma(X, Y, Z) \wedge primo(X) \wedge primo(Y))].$$

- *"Todo entero mayor que 1 es divisible por algún primo":*

  $$(\forall X)[X > 1 \rightarrow (\exists Y)(\exists Z)(primo(Y) \wedge mult(Y, Z, X))].$$

▪ **Ejemplo 79** *Empleando el cuantificador existencial podemos traducir con mayor corrección los enunciados del Ejemplo 77 al lenguaje formal:*

- *"Enrique es padre":* $(\exists X)padre(e, X)$.
- *"María es abuela":* $(\exists X)(\exists Y)(madre(m, X) \wedge (padre(X, Y) \vee madre(X, Y)))$.
- *"Guillermo es el cuñado de María":* $(\exists X)(marido(g, X) \wedge hermana(X, m))$.
- *"María es la tía de Arturo":*

  $$(\exists Z)(\exists W)\{[(marido(Z, m) \wedge hermano(Z, W)) \vee hermana(m, W)] \wedge (padre(W, a) \vee mad$$

- **Ejemplo 80** *Consideramos la interpretación $\mathcal{I}$ del Ejemplo 77:*
  - *"Todo el mundo tiene padre":* $(\forall X)(\exists Y)(padre(Y, X)$.
  - *"Ningún padre de alguien es madre de nadie":*

    $$(\forall X)[(\exists Y)padre(X, Y) \rightarrow \neg((\exists Z)madre(X, Z))],$$

    o bien

    $$(\forall X)(\forall Y)(\forall Z)[padre(X, Y) \rightarrow \neg madre(X, Z)].$$
  - *"Todos los abuelos son padres":*

    $$(\forall X)[(\exists Y)(\exists Z)(padre(X, Y) \wedge padre(Y, Z)) \rightarrow (\exists Y)padre(X, Y)]$$

    o bien

    $$(\forall X)(\forall Y)(\forall Z)(\exists W)[(padre(X, Y) \wedge padre(Y, Z)) \rightarrow padre(X, W)].$$

- Notad que la traducción de los enunciados del Ejemplo 80 y precedentes no refleja el estilo de representación de la programación lógica, orientado al análisis descendente y a la definición de predicados.

6.1.3. Representación en forma clausal.

- `Objetivo`: estudiar algunas peculiaridades de la representación del conocimiento mediante cláusulas.

- Traducción a la forma clausal (procedimiento general):

  1. traducir el enunciado a una fórmula de la lógica de predicados, siguiendo las recomendaciones del apartado anterior;

  2. aplicar el Algoritmo 3 para obtener una forma normal de Skolem;

  3. extraer las clásulas de la forma normal de Skolem y representarlas en notación cláusal.

  Debido al paso (1), el proceso completo no puede formalizarse como un algoritmo.

Representación en forma clausal.

- **Ejemplo 81**
  Dado el conjunto de enunciados castellanos

  1. *Marco Bruto era un hombre.*

  2. *Marco Bruto era partidario de Pompeyo.*

  3. *Todos los partidarios de Pompeyo eran romanos.*

  4. *Julio César fue un gobernante.*

  5. *Todos los romanos, que eran enemigos de Julio César, le odiaban.*

  6. *Todo el mundo es enemigo de alguien.*

  7. *Los romanos sólo intentan asesinar a los gobernantes de los que son enemigos.*

  8. *Marco Bruto asesinó a Julio César.*

Representación en forma clausal.

Traducción a forma clausal:

- `Paso` 1.
  Seleccionamos la interpretación de partida $\mathcal{I}$: $\mathcal{D}_{\mathcal{I}}$, son los seres humanos y la función de interpretación $\mathcal{J}$ asigna:

$hombre(X)$:	*X es un hombre;*
$romano(X)$:	*X es un romano;*
$gobernante(X)$:	*X es un gobernante;*
$partidario(X, Y)$:	*X es partidario de Y;*
$enemigo(X, Y)$:	*X es enemigo de Y;*
$odia(X, Y)$:	*X odia a Y;*
$asesina(X, Y)$:	*X asesina a Y;*
$bruto$:	*Marco Bruto;*
$pompeyo$:	*Pompeyo;*
$cesar$:	*Julio César;*

  Ahora podemos traducir los siguientes enunciados del lenguaje natural al lenguaje formal:

  1. *"Marco Bruto era un hombre": $hombre(bruto)$.*

  2. *"Marco Bruto era partidario de Pompeyo":*
     $partidario(bruto, pompeyo)$.

  3. *"Todos los partidarios de Pompeyo eran romanos":*
     $(\forall X)(partidario(X, pompeyo) \rightarrow romano(X))$.

Representación en forma clausal.

4. *"Julio César fue un gobernante": gobernante(cesar).*

5. *"Todos los romanos, que eran enemigos de Julio César, le odiaban.":*
$$(\forall X)\ (romano(X) \wedge enemigo(X, cesar) \rightarrow odia(X, cesar)).$$
Nótese que el pronombre relativo "que" (y de igual modo "quien" y "donde") se refiere a individuos ya mencionados en el mismo enunciado.

6. *"Todo el mundo es enemigo de alguien":*
$$(\forall X)(\exists Y) enemigo(X, Y).$$
Si el enunciado se hubiese interpretado en el sentido de que "hay alguien de quien todo el mundo es enemigo" se hubiese formalizado como:
$$(\exists Y)(\forall X) enemigo(X, Y).$$

7. *" Los romanos sólo intentan asesinar a los gobernantes de los que son enemigos":*
$$(\forall X)(\forall Y)\ (romano(X) \wedge gobernante(Y) \wedge asesina(X, Y) \rightarrow enemigo(X, Y)).$$
Si el enunciado se hubiese interpretado en el sentido de que "lo único que los romanos intentan hacer es asesinar a los gobernantes de los que son enemigos" se hubiese formalizado como:
$$(\forall X)(\forall Y)\ (romano(X) \wedge gobernante(Y) \wedge enemigo(X, Y) \rightarrow asesina(X, Y)).$$

Representación en forma clausal.

8. "Marco Bruto asesinó a Julio César": $asesina(bruto, cesar)$.

- Paso 2.

  Aplicamos el algoritmo de transformación a forma normal:

  1. $hombre(bruto)$

  2. $partidario(bruto, pompeyo)$

  3. $(\forall X)(partidario(X, pompeyo) \rightarrow romano(X))$
  $\Leftrightarrow$ $(\forall X)(\neg partidario(X, pompeyo) \vee romano(X))$
  $\Rightarrow$ $(\neg partidario(X, pompeyo) \vee romano(X))$

  4. $gobernante(cesar)$

  5. $(\forall X)(romano(X) \wedge enemigo(X, cesar) \rightarrow odia(X, cesar))$
  $\Leftrightarrow$ $(\forall X)\neg(romano(X) \wedge enemigo(X, cesar) \vee odia(X, cesar))$
  $\Leftrightarrow$ $(\forall X)(\neg romano(X) \vee \neg enemigo(X, cesar) \vee odia(X, cesar))$
  $\Rightarrow$ $\neg romano(X) \vee \neg enemigo(X, cesar) \vee odia(X, cesar)$

  6. $(\forall X)(\exists Y)enemigo(X, Y)$
  $\Rightarrow$ $(\forall X)enemigo(X, elEnemigoDe(X))$
  $\Rightarrow$ $enemigo(X, elEnemigoDe(X))$

  7. $(\forall X)(\forall Y)(romano(X) \wedge gobernante(Y) \wedge asesina(X, Y)$
  $\rightarrow enemigo(X, Y))$
  $\Leftrightarrow$ $(\forall X)(\forall Y)(\neg(romano(X) \wedge gobernante(Y) \wedge asesina(X, Y))$
  $\vee enemigo(X, Y))$
  $\Leftrightarrow$ $(\forall X)(\forall Y)(\neg romano(X) \vee \neg gobernante(Y) \vee \neg asesina(X, Y)$
  $\vee enemigo(X, Y))$
  $\Rightarrow$ $\neg romano(X) \vee \neg gobernante(Y) \vee \neg asesina(X, Y) \vee enemigo(X, Y)$

  8. $asesina(bruto, cesar)$

Representación en forma clausal.

- `Paso` 3.
  Finalmente, el conjunto de clausulas expresadas en notación clausal es:
  1. $hombre(bruto) \leftarrow$
  2. $partidario(bruto, pompeyo) \leftarrow$
  3. $romano(X) \leftarrow partidario(X, pompeyo)$
  4. $gobernante(cesar) \leftarrow$
  5. $odia(X, cesar) \leftarrow romano(X) \wedge enemigo(X, cesar)$
  6. $enemigo(X, elEnemigoDe(X)) \leftarrow$
  7. $enemigo(X, Y) \leftarrow$
     $romano(X) \wedge gobernante(Y) \wedge asesina(X, Y)$
  8. $asesina(bruto, cesar) \leftarrow$

  Esta representación es directamente implementable en Prolog y nos permite preguntar, por ejemplo, quien es enemigo de César.

- Este ejemplo nuevamente ilustra que algunos enunciados del lenguaje natural son ambiguos (e.g., el 6 y el 7) y que hay más de una forma de representar el conocimiento.

Representación en forma clausal.

- La `elección de la granulariadad` podría haber sido otra: e.g., distinguir entre "intenta asesinar" y "asesinar".
  - El séptimo enunciado se habría formalizado así:
    $enemigo(X, Y) \leftarrow$
    $romano(X) \wedge gobernante(Y) \wedge intenta\_asesinar(X, Y).$
  - Pero ahora falta conocimiento sobre el problema (la relación "intenta_asesinar" no está definida). Debe añadirse un nuevo enunciado:
    $$intenta\_asesinar(X, Y) \leftarrow asesina(X, Y).$$
  - En general, no es probable que en un conjunto de enunciados esté todo el conocimiento necesario para razonar sobre el tema.
- Los enunciados del tipo "... es un ..." se simbolizan mediante hechos que representan subconjuntos de individuos del universo de discurso.

Representación en forma clausal.

- En nuestro ejemplo el uso de funciones de Skolem ha sido inmediato, pero en otras ocasiones encierra ciertas sutilezas.

  Volviendo al Ejemplo 12:

  "todo el mundo tiene madre":

  $$(\forall X)[esHumano(X) \rightarrow (\exists Y)(esHumano(Y) \wedge esMadre(Y,X))]$$

  $$\Leftrightarrow \; (\forall X)[\neg esHumano(X) \vee (\exists Y)(esHumano(Y) \wedge esMadre(Y,X))]$$

  $$\Leftrightarrow \; (\forall X)(\exists Y)[\neg esHumano(X) \vee (esHumano(Y) \wedge esMadre(Y,X))]$$

  $$\Leftrightarrow \; (\forall X)(\exists Y)[(\neg esHumano(X) \vee esHumano(Y)) \wedge (\neg esHumano(X) \vee esMadre(Y,X))]$$

  $$\Rightarrow \; (\forall X)[(\neg esHumano(X) \vee esHumano(madre(X))) \wedge (\neg esHumano(X) \vee esMadre(madre(X),X))]$$

  $$\Rightarrow \; \neg esHumano(X) \vee esHumano(madre(X)),$$
  $$\neg esHumano(X) \vee esMadre(madre(X),X)$$

  $$\Rightarrow \; esHumano(madre(X)) \leftarrow esHumano(X)$$
  $$esMadre(madre(X),X) \leftarrow esHumano(X)$$

  `Hecho destacable`: partiendo de un enunciado obtenemos dos clausulas.

Representación en forma clausal.

- `Atajos`: Dada una fórmula:
  $(\forall X)(\forall Y)\ (romano(X) \wedge gobernante(Y) \wedge asesina(X,Y) \rightarrow enemigo(X,Y))$,
  para pasar a notación clausal eliminar los cuantificadores e introducir la implicación inversa:
  $enemigo(X,Y) \leftarrow$
  $romano(X) \wedge gobernante(Y) \wedge asesina(X,Y)$.

Fórmula de la lógica de predicados	**Notación clausal**
$(\forall X)\mathcal{B}(X)$	$\mathcal{B}(X) \leftarrow$
$(\forall X)(\mathcal{A}(X) \rightarrow \mathcal{B}(X))$	$\mathcal{B}(X) \leftarrow \mathcal{A}(X)$
$(\forall X)(\mathcal{Q}(X) \rightarrow \mathcal{B}(X))$	$\mathcal{B}(X) \leftarrow \mathcal{Q}(X)$
$(\forall X)(\mathcal{Q}(X) \rightarrow \mathcal{B}_1(X) \wedge \ldots \wedge \mathcal{B}_n(X))$	$\mathcal{B}_1(X) \leftarrow \mathcal{Q}(X)$ $\vdots$ $\mathcal{B}_n(X) \leftarrow \mathcal{Q}(X)$
$(\forall X)(\mathcal{Q}_1(X) \vee \ldots \vee \mathcal{Q}_n(X) \rightarrow \mathcal{B}(X))$	$\mathcal{B}(X) \leftarrow \mathcal{Q}_1(X)$ $\vdots$ $\mathcal{B}(X) \leftarrow \mathcal{Q}_n(X)$
$(\forall X)(\mathcal{Q}(X) \rightarrow \mathcal{B}_1(X) \vee \ldots \vee \mathcal{B}_n(X))$	$\mathcal{B}_1(X) \leftarrow \mathcal{Q}(X) \wedge \neg\mathcal{B}_2(X) \wedge \ldots \wedge \neg\mathcal{B}_n(X)$
$(\forall X)\neg\mathcal{Q}(X)$	$\leftarrow \mathcal{Q}(X)$
$(\forall X)(\mathcal{A}(X) \rightarrow \neg\mathcal{B}(X))$	$\leftarrow \mathcal{A}(X) \wedge \mathcal{B}(X)$
$(\forall X)(\mathcal{Q}_1(X) \rightarrow \neg\mathcal{Q}_2(X))$	$\leftarrow \mathcal{Q}_1(X) \wedge \mathcal{Q}_2(X)$
$(\forall X)(\mathcal{Q}_1(X) \rightarrow \neg\mathcal{Q}_2(X) \vee \mathcal{B}(X))$	$\mathcal{B}(X) \leftarrow \mathcal{Q}_1(X) \wedge \mathcal{Q}_2(X)$
$(\forall X)(\mathcal{Q}_1(X) \rightarrow (\mathcal{Q}_2(X) \rightarrow \mathcal{B}(X)))$	$\mathcal{B}(X) \leftarrow \mathcal{Q}_1(X) \wedge \mathcal{Q}_2(X)$
$(\exists X)\mathcal{A}(X)$	$\mathcal{A}(a) \leftarrow$
$(\exists X)(\mathcal{A}_1(X) \wedge \ldots \wedge \mathcal{A}_n(X))$	$\mathcal{A}_1(a) \leftarrow$ $\vdots$ $\mathcal{A}_n(a) \leftarrow$
$(\exists X)(\mathcal{A}_1(X) \vee \ldots \vee \mathcal{A}_n(X))$	$\mathcal{A}_1(a) \leftarrow \neg\mathcal{A}_2(a) \wedge \ldots \wedge \neg\mathcal{A}_n(a)$
$(\exists X)(\mathcal{A}(X) \wedge \neg\mathcal{B}(X))$	$\mathcal{A}(a) \leftarrow$ $\leftarrow \mathcal{B}(a)$
$(\exists X)\neg\mathcal{Q}(X)$	$\leftarrow \mathcal{Q}(a)$
$(\forall X)((\exists Y)\mathcal{Q}(X,Y) \rightarrow \mathcal{B}(X))$	$\mathcal{B}(X) \leftarrow \mathcal{Q}(X,Y)$
$(\forall X)(\mathcal{Q}(X) \rightarrow (\exists Y)\mathcal{B}(X,Y))$	$\mathcal{B}(X, f(X)) \leftarrow \mathcal{Q}(X)$

Los $\mathcal{A}_i$'s y los $\mathcal{B}_j$'s representan átomos, mientras que los $\mathcal{Q}_k$'s representan conjunciones de literales. Los símbolos a y f son, respectivamente, una constante y una función de Skolem.

Representación en forma clausal.

- **Cuantización existencial, extravariables, negación y respuesta de preguntas.**

- `Objetivo`: poner en relación la cuantización existencial con la aparición de extravariables en las cláusulas, la negación y la respuesta de preguntas.

- **Ejemplo 82** *Dado el enunciado "Una persona es abuelo de otra si tiene algún hijo que sea progenitor de esa otra" y la interpretación del Ejemplo 77 ampliada con la relación*

$$progenitor(X,Y): \ X \text{ es un padre o una madre } Y,$$

podemos formalizarlo como:

$$(\forall X)(\forall Y) \ [(\exists Z)(padre(X,Z) \wedge progenitor(Z,Y)) \rightarrow abuelo(X,Y)].$$

La traducción de esta fórmula a la forma clausal es:

$$\begin{array}{ll} & (\forall X)(\forall Y)[(\exists Z)(padre(X,Z) \wedge progenitor(Z,Y)) \rightarrow abuelo(X,Y)] \\ \Leftrightarrow & (\forall X)(\forall Y)[\neg(\exists Z)(padre(X,Z) \wedge progenitor(Z,Y)) \vee abuelo(X,Y)] \\ \Leftrightarrow & (\forall X)(\forall Y)[(\forall Z)(\neg padre(X,Z) \vee \neg progenitor(Z,Y)) \vee abuelo(X,Y)] \\ \Leftrightarrow & (\forall X)(\forall Y)(\forall Z)(\neg padre(X,Z) \vee \neg progenitor(Z,Y) \vee abuelo(X,Y)) \\ \Rightarrow & (\neg padre(X,Z) \vee \neg progenitor(Z,Y) \vee abuelo(X,Y)) \end{array}$$

Ahora el enunciado original tiene la siguiente representación en forma clausal:

$$abuelo(X,Y) \leftarrow padre(X,Z) \wedge progenitor(Z,Y).$$

Representación en forma clausal.

- La variable Z aparece en el cuerpo de la cláusula pero no es su cabeza. Este tipo de variables se denominan *extravariables*.

- Las extravariables están asociadas a un cuantificador existencial en la fórmula de la lógica de predicados $\Longrightarrow$ expresan existencia.

- Para una cláusula del tipo "$\mathcal{B}(X) \leftarrow \mathcal{Q}(X, Y)$" son posibles las siguientes lecturas:
  1. para todo X e Y, se cumple $\mathcal{B}(X)$ si se cumple $\mathcal{Q}(X, Y)$;
  2. para todo X, se cumple $\mathcal{B}(X)$ si existe un Y tal que se cumple $\mathcal{Q}(X, Y)$.

- En la notación clausal, la única posibilidad de representar la negación es en forma indirecta, como objetivo.

- **Ejemplo 83** *"Bat no es un hombre" se representa en la lógica de predicados mediante la fórmula*

  $$\neg hombre(bat)$$

  que se corresponde con el objetivo

  $$\leftarrow hombre(bat).$$

Representación en forma clausal.

Ejemplo 84 *"Nada es malo y bueno a la vez" se representa en la lógica de predicados mediante la fórmula*

$$(\forall X)\neg(malo(X) \wedge bueno(X))$$

que se corresponde con el objetivo

$$\leftarrow (malo(X) \wedge bueno(X)).$$

- El símbolo "$\leftarrow$" de los objetivos puede entenderse como una negación.

- `Respuesta de preguntas (existenciales)`:
  1. Preguntas del tipo "si/no"
     (e.g. ¿Bruto es partidario de Pompeyo? $\Longrightarrow$ $partidario(bruto, pompeyo)$).
  2. Preguntas del tipo "llenar espacios en blanco"
     (e.g. ¿Bruto odia a algún gobernante? $\Longrightarrow$ $(\exists X)[odia(bruto, X) \wedge gobernante(X)]$).

- En general las preguntas del segundo tipo vienen formalizadas como:

$$(\exists X)\ldots(\exists Z)\mathcal{Q}(X,\ldots,Z),$$

Representación en forma clausal.

- Para responder preguntas utilizando las técnicas de la programación lógica tendremos que:
  1. transformar los enunciados de la teoría a forma clausal.
  2. negar lo que queremos probar:

     $$\neg(\exists X)\ldots(\exists Z)\mathcal{Q}(X,\ldots,Z) \Leftrightarrow (\forall X)\ldots(\forall Z)\neg\mathcal{Q}(X,\ldots,Z),$$

     de manera que la pregunta original se transforma en la cláusula objetivo:

     $$\leftarrow \mathcal{Q}(X,\ldots,Z).$$

  3. aplicar la estrategia de resolución SLD.

- Las variables de un objetivo están asociadas con un cuantificador existencial.

- Desde el punto de vista de lo que se prueba, un objetivo como

  $$\leftarrow odia(bruto, X) \wedge gobernante(X)$$

  admite la siguiente lectura: existe un X tal que $odia(bruto, X)$ y $gobernante(X)$.

Representación en forma clausal.

- `Respuesta de preguntas que son implicaciones`:

  $(\forall X)(\mathcal{Q}_1(X) \rightarrow \mathcal{Q}_2(X))$.

  Debemos negar la fórmula y transformarla a forma clausal:

  $$\begin{array}{ll} & \neg(\forall X)(\mathcal{Q}_1(X) \rightarrow \mathcal{Q}_2(X)) \\ \Leftrightarrow & \neg(\forall X)(\neg\mathcal{Q}_1(X) \vee \mathcal{Q}_2(X)) \\ \Leftrightarrow & (\exists X)\neg(\neg\mathcal{Q}_1(X) \vee \mathcal{Q}_2(X)) \\ \Leftrightarrow & (\exists X)(\mathcal{Q}_1(X) \wedge \neg\mathcal{Q}_2(X)) \\ \Rightarrow & \mathcal{Q}_1(a) \wedge \neg\mathcal{Q}_2(a)) \qquad \text{paso a forma clausal} \end{array}$$

  y que en notación clausal se expresa como:

  $\mathcal{Q}_1(a) \leftarrow$,
  $\leftarrow \mathcal{Q}_2(a)$.

- **Conjuntos y las relaciones "instancia" y "es_un".**

- En castellano se emplean palabras que hacen referencia a conjuntos de individuos (e.g. "hombres", "animales", "números naturales").

- De una forma muy simple podemos identificar los conjuntos con los tipos de datos de los lenguajes de programación.

Representación en forma clausal.

- En la lógica de predicados se emplea:
  - un predicado para representar un conjunto o tipo;
  - un predicado aplicado sobre una variable para expresar la pertenencia de un individuo indeterminado a ese conjunto o tipo
    (e.g., "$hombre(X)$", "$animal(X)$", "$natural(X)$").

- Para tratar los tipos de datos como individuos y no como propiedades de individuos suele emplearse símbolos de constante y las relaciones:

  $instancia(X, A)$: X pertenece al conjunto A;
  $es\_un(A, B)$: A es subconjunto de B;

- Cuando se representan los conjuntos mediante las relaciones "$instancia$" y "es_un" se necesita introducir un nuevo axioma en la teoría:

$$(\forall X)(\forall A)(\forall B)\ (instancia(X, A) \wedge es\_un(A, B) \rightarrow instancia(X, B),$$

  Representada en notación clausal, esta fórmula se escribe como:

$$instancia(X, B) \leftarrow instancia(X, A) \wedge es\_un(A, B).$$

Representación en forma clausal.

- **Ejemplo 85** *Si representamos los enunciados*
  1. *Los hombres son mortales.*
  2. *Todo romano es un hombre.*
  3. *César fue un gobernante romano.*

  en notación clausal, obtenemos:

  1. $mortal(X) \leftarrow hombre(X)$
  2. $hombre(X) \leftarrow romano(X)$
  3. $gobernante(cesar) \leftarrow$
  4. $romano(cesar) \leftarrow$

  Estas fórmulas se puede escribir empleando la nueva forma de representar los conjuntos como:

  1. $es\_un(hombre, mortal) \leftarrow$
  2. $es\_un(romano, hombre) \leftarrow$
  3. $instancia(cesar, gobernante) \leftarrow$
  4. $instancia(cesar, romano) \leftarrow$
  5. $instancia(X, B) \leftarrow instancia(X, A) \wedge es\_un(A, B)$

  Las constantes "hombre", "romano", "gobernante", y "mortal" son constantes que representan conjuntos.

- Dado que son posibles varias formas de representar la noción de conjunto, hay que ser consistente y una vez elegida una de ellas no deben mezclarse.

Representación en forma clausal.

- **Representación mediante relaciones binarias.**

- Los argumentos de una relación n-aria tienen una función según el orden que ocupan en la fórmula.

- Toda relación n-aria puede expresarse como la unión de $n+1$ relaciones binarias que ponen de manifiesto de forma explícita la función de cada uno de sus argumentos.

- **Ejemplo 86** *La relación ternaria "Juan dio un libro a María" puede representarse mediante el hecho:*

  $dar(juan, libro, maria) \leftarrow .$

  Esta relación puede expresarse en los siguientes términos:

  1. *Hay una acción que denominamos d1*
  2. *que es una instancia de un acto de dar*
  3. *por un dador, juan*
  4. *de un objeto, libro*
  5. *a un receptor, maria*

  La relación ternaria inicial puede escribirse en términos de cuatro relaciones binarias:

  1. $instancia(d1, dar) \leftarrow$
  2. $dador(d1, juan) \leftarrow$
  3. $objeto(d1, libro) \leftarrow$
  4. $receptor(d1, maria) \leftarrow$

- Notar que se ha prescindido del hecho que indica que "*dar*" es del tipo "*acción*" (i.e., $es\_un(dar, acción)$).

Representación en forma clausal.

- Para sustituir una relación n-aria por $n+1$ relaciones binarias:

  1. Tratar la relación n-aria como una individualidad, asignandole un nombre (e.g., "$d1$") y, también, asignar un nombre a la acción que simboliza $d1$ (e.g., "dar").

  2. Introducir una relación binaria que indique la pertenencia de la relación n-aria a una determinada acción (e.g., "$d1$ es un acto de dar", que puede simbolizarse como: $instancia(d1, dar)$).

  3. Introducir una relación binaria por cada argumento de la relación n-aria, para fijar el papel de cada argumento en la acción "$d1$".

- Las relaciones binarias como "$dador(d1, juan)$" reciben el nombre de *slots*.

  1. "$dador$" es el nombre del *slot* y

  2. "$juan$" es el valor del *slot*.

Representación en forma clausal.

- Entre las ventajas de la representación binaria podemos citar las siguientes:

  1. Es más fácil de leer que la representación n-aria.

  2. Es más expresiva que la representación n-aria.
     (Las relaciones pueden emplearse como argumento de otra relación)

  3. Facilita la inserción de hechos nuevos.

  4. Facilita la formulación de preguntas.

  5. Facilita la especificación de reglas generales y restricciones de integridad.

- Algunas desventajas de la representación binaria son:

  1. Hay muchas relaciónes que se expresan más facilmente como relaciones n-arias.

  2. El acceso a todas las informaciones relacionadas sobre una entidad u objeto es difícil y poco eficiente.

6.1.4. Redes semánticas y representación clausal.

- `Objetivo`: mostrar las relaciones existentes entre las redes semánticas y la lógica.

- Desarrolladas por Quillian (1968) como modelo psicológico de la memoria asociativa humana.

- Utilizadas en la IA como herramientas para la representación del conocimiento.

- Permiten implementar una de las formas más útiles de inferencia: la herencia de propiedades.

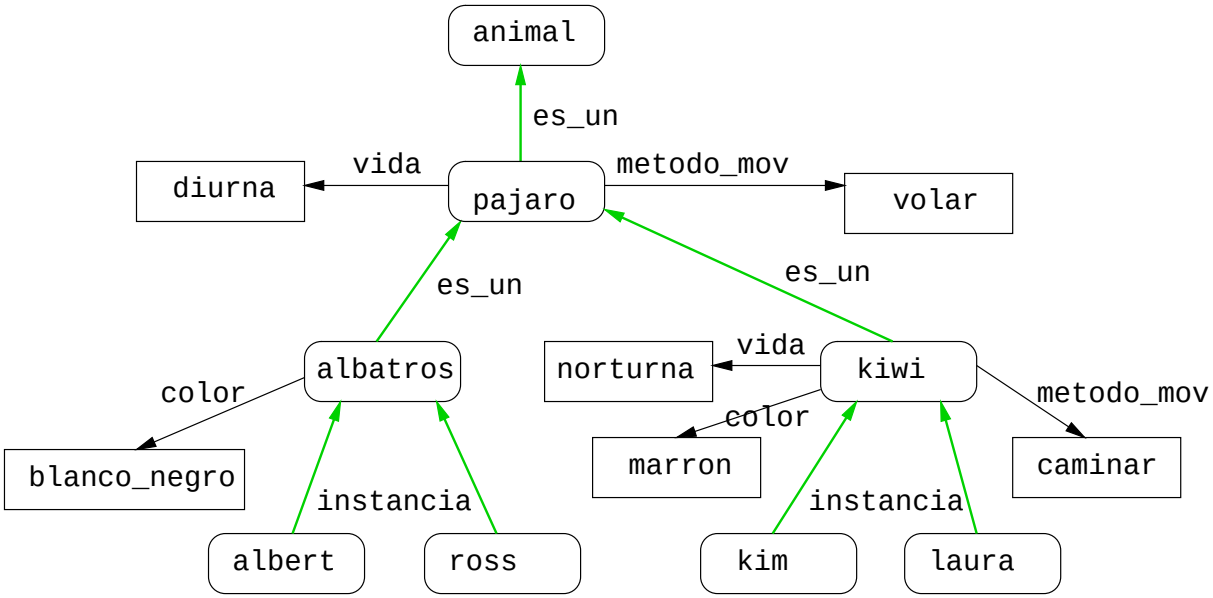
- Una *red semántica* es un grafo dirigido constituido por nodos y por arcos.

  1. Los *nodos* representan entidades que pueden ser individuos, objetos, clases (conjuntos) de individuos u objetos, conceptos, acciones genéricas (eventos) o valores.

  2. Los *arcos* representan relaciones entre nodos o bien atributos (i.e., características) de un nodo.

  Una red semántica consiste en un conjunto de entidades relacionadas.

Redes semánticas y representación clausal.



- Ventajas:

  1. Facilitan la representación jerárquica de grandes conjuntos de hechos estructurados.

  2. **`Indexación de la información`**: La capacidad de acceder a todas las informaciones relevantes sobre una entidad (mediante el uso de punteros).

  3. **`Herencia de propiedades`**: Los valores de los atributos de una entidad pueden ser heredados por otras entidades que están en la relación "es_un" e "instancia".

- **Algoritmo 9 (Herencia de Propiedades)**

Entrada: *Una red semántica S y una petición de acceso a un valor V de un (supuesto) atributo A de una entidad E.*

Salida: *El valor V o una condición de* `fallo`.

Comienzo

Inicialización: $NuevoE = E$; $Resultado =$ `fallo`; $SalirBucle =$ `falso`.

Repetir

En caso de que $NuevoE$ *posea*

1. *El atributo A:*
   $Resultado = V$; $SalirBucle =$ `verdadero`;
   `Fin caso`.
2. *El atributo instancia:*
   $NuevoE = E'$, *donde*
   E' *es el valor del atributo instancia;*
   `Fin caso`.
3. *El atributo es_un:*
   $NuevoE = E'$, *donde*
   E' *es el valor del atributo es_un;*
   `Fin caso`.
4. **En otro caso:**
   $SalirBucle =$ `verdadero`.

Fin caso

Hasta que $SalirBucle$.

Devolver $Resultado$

Fin

- Para mantener simple el Algoritmo 9 no se ha considerado la *herencia múltiple*.

Redes semánticas y representación clausal.

- A pesar de las diferencias las redes semánticas y la lógica de predicados son técnicas equivalentes.

- **Ejemplo 87** *Es inmediato representar en forma clausal el conocimiento que encierra la red semántica de la figura anterior.*

```
% Entidad pajaro
es_un(pajaro,animal).
metodo_movimiento(pajaro,volar).
vida(pajaro,diurna).

% Entidad albatros
es_un(albatros,pajaro).
color(albatros,blanco_negro).

% Entidad albert
instancia(albert,albatros).


% Entidad ross
instancia(ross,albatros).


% Entidad kiwi
es_un(kiwi,pajaro).
color(kiwi,marron).
metodo_movimiento(kiwi,caminar).
vida(kiwi,nocturna).

% Entidad kim
instancia(kim,kiwi).
```

Redes semánticas y representación clausal.

Es necesario el axioma adicional:

```
% conocimiento implicito
instancia(X, B) :- es_un(A, B), instancia(X, A).
```

Reglas capaces de implemetar la herencia de propiedades (Algoritmo 9):

```
% pos(Hecho), el Hecho es posible
% Reglas que modelan la herencia de propiedades
% Un Hecho es posible si es deducible directamente
% de la red
pos(Hecho) :- Hecho,!.
% Un Hecho es posible si puede ser inferido por
% herencia
pos(Hecho) :- Hecho =.. [Rel,Arg1,Arg2],
              (es_un(Arg1,SuperArg);
               instancia(Arg1,SuperArg)),
              SuperHecho =.. [Rel,SuperArg,Arg2],
              pos(SuperHecho).
```

- Nótese que la variable "`Hecho`" es una metavariable.

- Ni la red semántica ni su representación en forma clausal pueden responder a preguntas como: ¿Qué entidades tienen vida diurna? (i.e., "`?- pos(vida(X, diurna)).`").

- El Ejemplo 87 ilustra un tipo de programación denominada *orientada a objetos*.
  - La guía para la estructuración del conocimiento son los nombres y no los verbos.

6.2. Resolución de problemas.

- Para construir un sistema que resuelva un problema específico es necesario realizar las siguientes acciones:

  1. Definir el problema con precisión.
     Especificar situaciones iniciales y situaciones finales que se aceptarían como solución.

  2. Analizar el problema.
     Identificar características que permitan determinar la técnica más adecuada para su resolución.

  3. Aislar y representar el conocimiento necesario para la resolución del problema.

  4. Elegir las mejores técnicas que resuelvan el problema y aplicarlas.

- Los problemas a los que se enfrenta la IA no tienen una solución algorítmica directa.

- La única forma posible de solución es la *búsqueda en un espacio de estados*.

6.2.1. Definición de un problema mediante una búsqueda en un espacio de estados.

- La representación de un problema como un *espacio de estados* es una idea fundamental que proviene de los primeros trabajos de investigación en el área de la IA.

- El esquema general de solución de este tipo de problemas responde a la siguiente estructura:

  1. Definir formalmente el espacio de estados del sistema:

     - Identificar una noción de estado.

     - Identificar uno o varios estados iniciales y estados objetivos.

  2. Definir las reglas de producción, operaciones o movimientos:
     Un conjunto de reglas que permiten cambiar el estado actual y pasar de un estado a otro cuando se cumplen ciertas condiciones.

3. Especificar la estrategia de control:

   Dirige la búsqueda comprobando
   - Cuándo una regla es aplicable (a un estado).
   - Qué reglas aplicar.
   - si se cumplen las condiciones de terminación.
   - si se cumplen las condiciones de terminación y registrando las reglas que se han ido aplicando.

   Una estrategia de control debe cumplir los siguientes requisitos:

   a) que cause algún cambio de estado (necesidad de *cambio local*);

   b) que sea sistemática y evite la obtención de secuencias de estados redundantes (necesidad de *cambio global*).

- Podemos pensar en este espacio de estados (conceptual) como un grafo dirigido:
  - los nodos son estados del problema;
  - los arcos están asociados a las reglas u operaciones.

Resolución de problemas.

- Ahora la solución de un problema consiste en encontrar un camino, en el espacio de estados, que nos lleve desde el estado inicial a un estado objetivo.

6.2.2. Resolución de problemas y programación lógica.

- Muchos autores separan la especificación del espacio de estados de la descripción del control.

- La separación entre lógica y control es una de las máximas de la programación declarativa.

- Los lenguajes de programación lógica (Prolog) están bien adaptados para la resolución de problemas:
  - permitien que nos centremos en la especificación del problema;
  - hacien innecesario (en el mejor de los casos) programar la búsqueda.

- **Ejemplo 88** *Problema del mono y la banana.*

  El espacio de estados se especifica mediante un término `s(Pm, Sm, Pc, Tm)`*. Los argumentos indican:*

  - `Pm`*: El mono esta en la posicion* `Pm`*;*
  - `Sm`*: El mono esta sobre el suelo (constante* `suelo`*) o sobre la caja (constante* `encima`*);*
  - `Pc`*: La caja esta en la posicion* `Pm`*;*
  - `Tm`*: El mono "tiene" o "notiene" la banana;*

  Posiciones posibles son: `puerta`*,* `ventana` *y* `centro`*, una variable* `X` *indica cualquier otra posicion.*

  La solución propuesta es la siguiente:

```
% Estado inicial
posible(s(puerta, suelo, ventana, notiene)).

% Movimientos
move(s(P1, suelo, P, T), andar(P1, P2),
s(P2, suelo, P, T)).
move(s(P1, suelo, P1, T),empujar(P1, P2),
s(P2, suelo, P2, T)).
move(s(P, suelo, P, T), subir, s(P, sobre, P, T)).
move(s(centro, sobre, centro, notiene), coger,
s(centro, sobre, centro, tiene)).

% Axioma del espacio de estados
posible(S2) :- posible(S1), move(S1, M, S2).
```