RESUMEN

PROGRAMACIÓN LÓGICA

# CONCEPTOS BÁSICOS

## INTRODUCCIÓN

Aaaaa

## UNIFICACIÓN

## RECURSIÓN

# LISTAS Y ACUMULADORES

## LISTAS

Aaaaa

## ACUMULADORES

# LÓGICA DE PRIMER ORDEN

## INTRODUCCIÓN

Sea L un lenguaje de primer orden. Una **interpretación** I para L consiste en:

* Un conjunto D no vacío (dominio de interpretación)
* Un mapeo Mc de las constantes de L en D
* Un mapeo Mf para los símbolos de función L, tal que a f n-ario le corresponde una función Dn 🡪 D
* Un mapeo Mp para los símbolos de predicado de L, tal que a p n-ario le corresponde una función Dn 🡪 {v,f} (o sea una relación en Dn)

## SINTÁXIS

**Término:**

1. Una variable es un término.
2. Una constante es un término.
3. Si f es una función n-aria y t1. . . tn términos, f(t1,. . . ,tn) es un término.
4. Estos son todos los términos.

**Fórmula bien formada:**

1. Si p es una predicado n-ario y t1. . . tn términos, p(t1,. . . ,tn) es una fórmula (átomo o fórmula atómica).
2. Si F y G son fórmulas, también lo son:
   1. (¬F)
   2. (F v G)
   3. (F ^ G)
   4. (F→G)
   5. (F↔G)
3. Si F es una fórmula y x una variable, también son fórmulas:
   1. ( x F)
   2. ( x F)
4. Estas son todas las fórmulas.

**Alcance de un cuantificador:**

* El alcance de en (xF) es F
* El alcance de en (xF) es F
* Una ocurrencia de una variable está **ligada** si está inmediatamente luego de un cuantificador o es una ocurrencia dentro del alcance de un cuantificador. En caso contrario diremos que la ocurrencia se encuentra **libre**.
* Se dice que una variable es una **variable ligada (/libre)** cuando todas sus ocurrencias están ligadas (/libres).
* Una **fórmula cerrada** es una fórmula que no presenta ocurrencias de variables libres.

Un **literal** es un átomo (**literal positivo**), o su negado (**literal negativo**).

Una **cláusula** es una fórmula cerrada con la siguiente forma:

*x1 … xk(L1 V … V Lp)*

en donde cada Li es un literal y x1…xk son todas las variables que ocurren en L1V…VLp

Una **cláusula definida** es una cláusula que contiene exactamente un literal positivo.

*regla: A1 ← B1 Λ … Λ Bq*

*cláusula unitaria o hecho: A1 ←*

Un **programa definido** es un conjunto de cláusulas definidas (reglas y hechos).

Un **objetivo definido** es una cláusula sin literales positivos:

*← B1 Λ … Λ Bq*

Una **cláusula de Horn** es una cláusula definida o un objetivo definido. Tanto los programas lógicos (conjuntos de cláusulas definidas) como las consultas van a ser modeladas con cláusulas de Horn. Las consultas van a ser negadas para así obtener objetivos definidos.

## SEMÁNTICA

Sea L un lenguaje de 1er orden, I una interpretación para L de dominio D. Una asignación V de valores a variables (según I) es una función que hace corresponder a cada variable un valor en D.

Valor de una fórmula F según una interpretación I y una asignación V

1. F atómica
2. F es de la forma :
   1. F1 ^ F2
   2. F1 v F2
   3. F1 🡪 F2
   4. ¬ F1
3. F es de la forma
   1. x P(x)
   2. x P(x)

**Caso 1: F es atómica**

* A cada variable x se le da un valor h(x) según V.
* A cada constante c se le da un valor h(c) según I, h(c) = Mc(c).
* El valor asociado a un término f(t1,...,tn) es h(f(t1,...,tn) = Mf(h(t1),...,h(tn)) (h(f) en D).
* El valor de verdad asociado a una fórmula atómica p(t1,...,tn) es Mp(h(t1),...,h(tn)).

**Caso 2:**

* Se utilizan las reglas para los conectores ^, v, 🡪 y ¬

**Caso 3:**

* F = x G

F es verdadera si existe d en D t.q. G es verdadera para I y V(x/d), donde V(x/d) es V excepto para la variable x que se asigna a d; en caso contrario es falsa.

* F = x G.

F es verdadera si para todo d en D G es verdadera para I y V(x/d) en caso contrario es falsa.

## MODELOS

Sea F una fórmula cerrada e I una interpretación, I es un **modelo** para F si F es verdadera según I (I satisface F).

Sea S un conjunto de fórmulas cerradas, I es un **modelo** para S si satisface todas las fórmulas de S.

Sea S un conjunto de fórmulas cerradas:

* S es **satisfactible** si existe al menos un modelo para S.
* S es **válido** (lógicamente válido) si toda interpretación es modelo.
* S es **insatisfactible** si ninguna interpretación es modelo.

Sea S un conjunto de fórmulas cerradas y F una fórmula cerrada. F es **consecuencia lógica** de S si todo modelo de S es modelo de F.

**Teo:** Sea S={F1,F2,...,Fn}, F es consecuencia lógica de S sii (F1^F2^...^Fn) 🡪 F es una fórmula válida.

**Teo:** Sea S un conjunto de fórmulas cerradas y F una fórmula cerrada sobre L de primer orden: S ╞ F sii S U {¬ F} es insatisfactible.

## RESOLUCIÓN

Para probar que una consulta C es consecuencia lógica de un programa lógico P basta con probar que P U {¬ C} es insatisfactible. A este tipo de demostración se le llama **prueba por refutación**: se niega lo que se quiere probar, llegando a un absurdo.

La cláusula vacía:

* Si nuestro conjunto de cláusulas es un par de literales complementarios en un paso de resolución llegamos a la cláusula vacía. Por ejemplo, resolviendo ¬P y P obtenemos como resultado una cláusula sin literales.
* La cláusula vacía (se escribe □ ) no tiene modelos, puesto que no tiene literales que puedan hacerse verdaderos para una interpretación.
* Como la resolución mantiene los modelos (la resolvente es consecuencia lógica de las dos cláusulas que se resuelven), si de un conjunto de cláusulas se obtiene la cláusula vacía por aplicación de uno o más pasos de resolución, podemos deducir que el conjunto de cláusulas no tiene ningún modelo. O sea, es insatisfactible.

# RESOLUCIÓN Y UNIFICACIÓN

## SUSTITUCIONES

## UNIFICACIÓN

## RESOLUCIÓN

## SELECTION LINEAR DEFINITE (SLD)

**Resolución SLD**

* La resolución con cláusula de entrada es completa para cláusulas de Horn (cláusulas definidas + objetivo definido).
* Resolución SLD (Selection Linear Definite) es un tipo particular de resolución lineal con cláusulas de entrada para conjuntos de cláusulas de Horn.
* Siempre se parte del objetivo, por lo que en todos los pasos las resolventes también son objetivos.

**Derivación SLD:**

Sea P un programa definido, G un objetivo definido 🡨 A1,...,An. Una derivación SLD de P U {G} es:

* una secuencia G0=G, G1, G2,... de objetivos,
* una secuencia C1,C2,... de variantes de cláusulas de P y
* una secuencia θ1, θ2,... de mgus tales que cada Gi+1 se deriva de Gi y Ci con unificador θi.

**Refutación SLD:**

Una refutación-SLD para P U {G} es una derivación-SLD finita cuyo último objetivo es □.

Procedimiento de refutación SLD:

* El objetivo de un intérprete es, dado un programa P y una consulta C, encontrar una refutación-SLD para P U {G}, siendo G = ¬C.
* Una refutación está compuesta por una cantidad finita de pasos de resolución-SLD. En cada uno de estos pasos hay 2 aspectos que podrían resolverse de más de un modo:
  + cuál es el átomo del objetivo que se selecciona para reducir
  + cuál es la cláusula del programa que reduce con el átomo seleccionado del objetivo.

**Respuesta computada**

Una respuesta computada para P U {G} es la sustitución obtenida restringiendo a las variables de G la composición de mgus utilizados en una refutación-SLD de P U {G}.

La correctitud de la resolución-SLD ya fue demostrada (fue demostrada respecto a resolución en general, se cumple para resolución-SLD). Es posible demostrar además un resultado de correctitud respecto a la respuesta computada.

**Respuesta correcta**

Sean P un programa lógico, G un objetivo y θ una sustitución para las variables de G, entonces θ es una respuesta correcta si:

x1...xk((A1,A2,...,Ak) θ) es consecuencia lógica de P.

**Completitud**

Para cláusulas de Horn la resolución-SLD es completa. En otros términos, si P U {G} es insatisfactible, existe una refutación-SLD para P U {G}.

**Árbol SLD**

Una **regla de computación** es una política de selección del siguiente átomo a reducir en un objetivo. La regla de computación no afecta la existencia de una refutación-SLD.

Considerando una regla de computación fija, el espacio de búsqueda (asociado a las distintas formas de elegir las cláusulas del programa) es un árbol. Lo llamamos **árbol-SLD**.

* Cada rama del árbol SLD corresponde a una posible derivación SLD.
* Por lo tanto, el árbol SLD representa todas las posibles derivaciones SLD a partir de un objetivo, utilizando una determinada regla de computación.
* La regla de computación determina las ramas del árbol: para un mismo programa lógico y objetivo, el árbol SLD posiblemente cambie al cambiar la regla de computación.
* A las ramas que corresponden a refutaciones (la hoja es un objetivo vacío) se les llama **ramas exitosas**.
* A las ramas que corresponden a derivaciones infinitas se les llama **ramas infinitas**.
* A las ramas que corresponden a derivaciones fallidas se les llama **ramas de falla**.

**Refutación SLD**

# GENERACIÓN Y CHEQUEO

## GENERACIÓN Y CHEQUEO

Es una técnica para diseñar y programar algoritmos:

* Un proceso genera soluciones candidatas al problema
* Otro proceso chequea los candidatos, intentando encontrar aquellos que resuelven el problema

Forma general:

*solucion(X) :-*

*generacion(X),*

*chequeo(X).*

Ejemplo:

*ordenar(L,P) :-*

*permutacion(L,P),*

*ordenada(P).*

## EFICIENCIA

Un problema que se suele dar con esta técnica es la ineficiencia.

Una técnica para optimizar programas consiste en “empujar” el chequeo dentro del generador lo más profundo posible. También se debe intentar hacer fallar una solución lo antes posible.

# PREDICADOS METALÓGICOS

## RECONOCIMIENTO DE TIPOS

Reconocimiento de tipos de términos:

* integer(X) 🡨 X es un entero
* float(X) 🡨 X es un número de punto flotante
* atom(X) 🡨 X es un átomo
* compound(X) 🡨 X es un término estructurado (functor y argumentos)

Otros predicados (basados en los anteriores):

* number(X) 🡨 X es un entero o un real
* atomic(X) 🡨 X es un número o un átomo

## INSTANCIACIÓN DE VARIABLES

Predicados para el análisis del estado de instanciación de un término:

* var(X) ← X es una variable sin instanciar
* nonvar(X) ← X está instanciado
* ground(X) ← X está completamente instanciado

## MANIPULACIÓN DE TÉRMINOS

Predicados asociados a la estructura e instanciación de los términos. Permiten:

* Reconocer distintos tipos de términos
* Analizar el estado de instanciación de un término
* Descomponer términos en functor y argumentos
* Crear nuevos términos

**Predicado functor:**

*functor(Termino,Functor,Aridad) ← Termino es un término compuesto con nombre Functor y Aridad argumentos.*

**Predicado arg:**

*arg(N,Termino,Argumento) ← Permite inspeccionar el N-ésimo argumento de un término*

**Predicado univ (=..):**

*Termino =.. Lista ← Permite construir e inspeccionar términos n-arios*

?- g(A,f(1),atom) =.. X.

X = [g,A,f(1),atom].

Predicado setarg:

*Permite setear el valor de un argumento en un término*

?- Termino=f(a,b,c), setarg(2,Termino,otro).

Termino = f(a,otro,c).

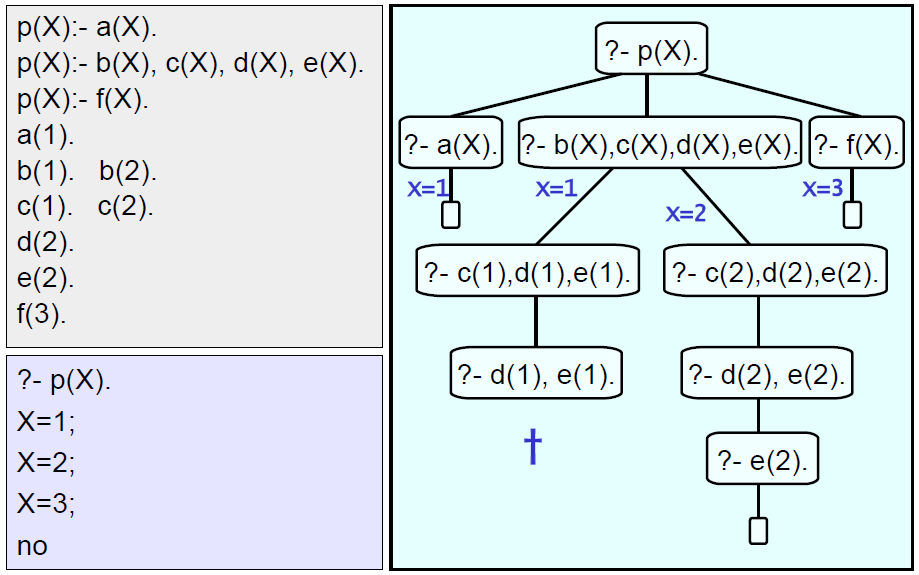
## predicado cut (!)

El backtracking es un rasgo característico de Prolog, que se produce cada vez que se llega a un punto de falla en el árbol SLD.

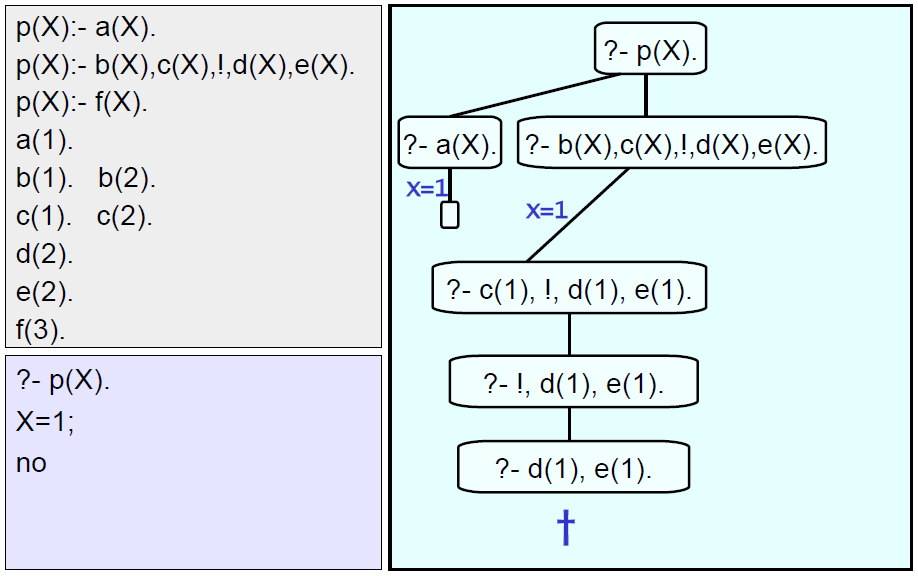
El predicado cut !/0 permite controlar el backtracking, realizando una poda en el árbol SLD. Lo que hace es eliminar puntos de opción del árbol SLD.

Cuts que no cambian el significado de un programa se llaman **cuts verdes**, mientras que los que sí cambian el significado del programa son llamados **cuts rojos**.

**Ejemplo sin cut:**



**Ejemplo con cut:**



## predicado not (\+)

En el esquema de Prolog puro la negación no existe. Se implementa en Prolog la negación según hipótesis de mundo cerrado: podemos deducir no(C) si no podemos probar C a partir de nuestro programa P.

Para deducir no(C) intentamos probar C. Si el árbol-SLD para P U <- C es finitamente fallado, deducimos no C.

El cut permite implementar la negación. Utilizamos además un predicado del sistema que siempre falla: fail.

*not (X) :- X, !, fail.*

*not (X).*

* El predicado not se denota \+ en SWI.
* \+ no instancia las variables, es útil para chequear propiedades.
* La negación es costosa, ya que implica recorrer todo el árbol-SLD.
* Es incompleta, ya que el árbol-SLD puede ser infinito.

# PREDICADOS EXTRALÓGICOS

## PREDICADOS DE SEGUNDO ORDEN

Son “predicados que tienen como parámetro otros predicados”.

**Predicado findall:**

*findall(+Template,+Goal,-Bag) ← Bag es la lista con todas las instancias de Template para las cuales se satisface Goal. En caso de que Goal sea insatisfactible, Bag unifica con la lista vacía.*

**Predicado bagof:**

*bagof(+Template,+Goal,-Bag) ← Bag es la lista con todas las instancias de Template para las cuales se satisface Goal para una “asignación” de las variables que NO aparecen en Template. En caso de que Goal no sea satisfactible, falla.*

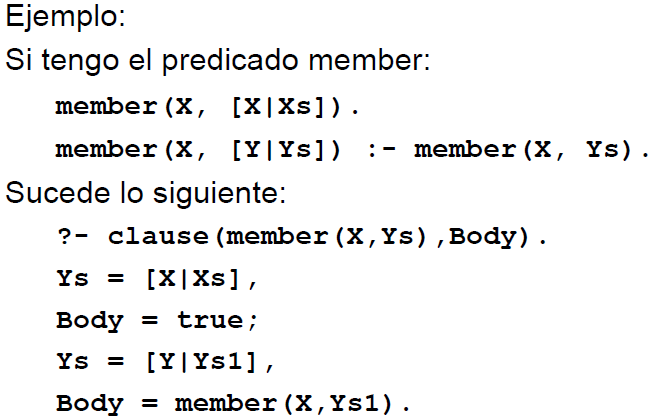
**Predicado setof:**

*setof(+Template,+Goal,-Bag) ← Ídem a bagof, pero con Bag ordenada y sin repetidos.*

## MANIPULACIÓN DE PROGRAMAS

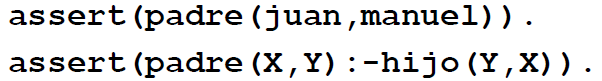
**Predicado clause:**

*clause(+Head, ?Body) 🡨 Permite acceder a las cláusulas de un programa*



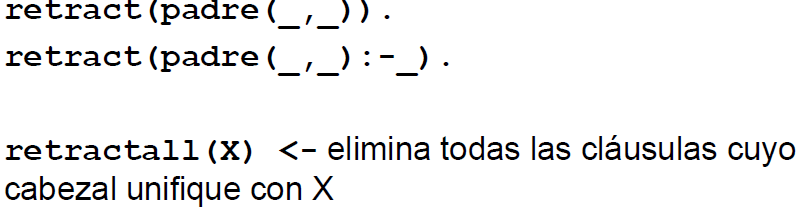
**Predicado assert:**

*assert(+Clause) 🡨 Agrega una cláusula al programa*



**Predicado retract:**

*retract(+Term) 🡨 Elimina una cláusula del programa*



## PREDICADOS NON-BACKTRACKEABLE

*No es muy importante.*

## ENTRADA/SALIDA

*No es muy importante*.

# METAINTÉRPRETES

## METAPROGRAMAS

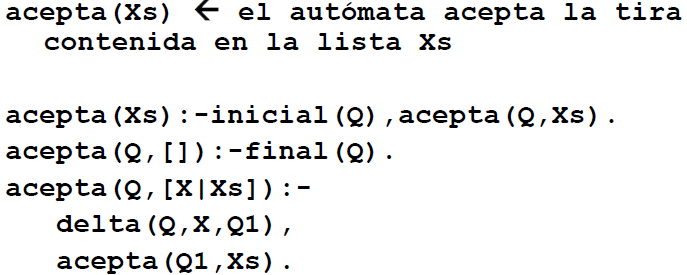
Metaprogramas en Prolog:

* Tratan a otros programas como datos
* En Prolog hay equivalencia entre programas y datos: ambos son términos.
* Es muy simple escribir programas para interpretar otros programas en Prolog, o para interpretar autómatas de distinto tipo.

## INTÉRPRETES AFND

Los lenguajes reconocidos por AFNDs son la clase de lenguajes regulares.

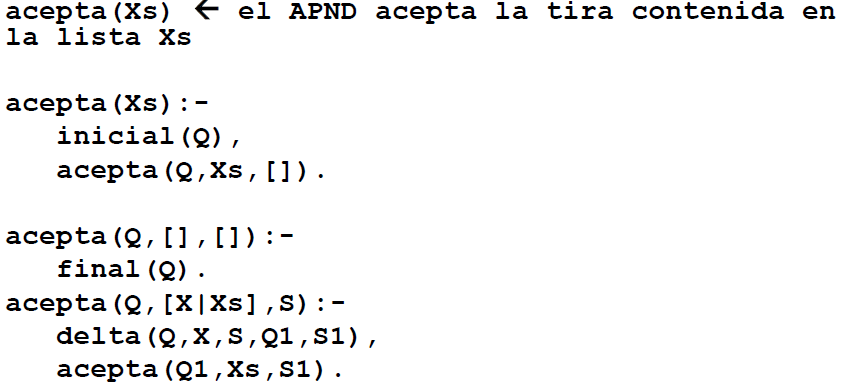
Es muy sencillo escribir un intérprete para un AFND:



El no determinismo surge naturalmente de la ejecución de Prolog.

## INTÉRPRETES APND

Los APND reconocen la clase de lenguajes libres de contexto.



El no determinismo surge naturalmente de la ejecución de Prolog.

## METAINTÉRPRETES

Un metaintérprete de un lenguaje es un intérprete para el lenguaje escrito en el mismo lenguaje

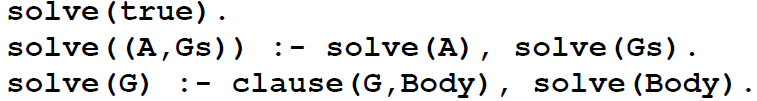
* Es un caso particular de metaprograma
* Las facilidades de Prolog hacen que resulte sencillo interpretarse a sí mismo

Prolog tiene facilidades especiales para tratar como datos sus propios programas o para tratar datos como programas:

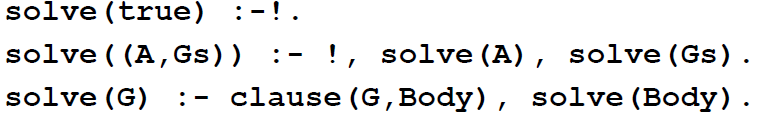
* Facilidad de metavariable:
  + call(A) … o simplemente A
* Inspección de términos:
  + functor
  + arg
  + univ
* Inspección de programas:
  + clause(Cabeza,Cuerpo)
* Modificación del programa:
  + assert(Clausula)
  + retract(Clausula)

La idea es de la forma:

*solve(Obj) 🡨 se verifica el objetivo Obj dado el programa Prolog puro indicado por clause/2*



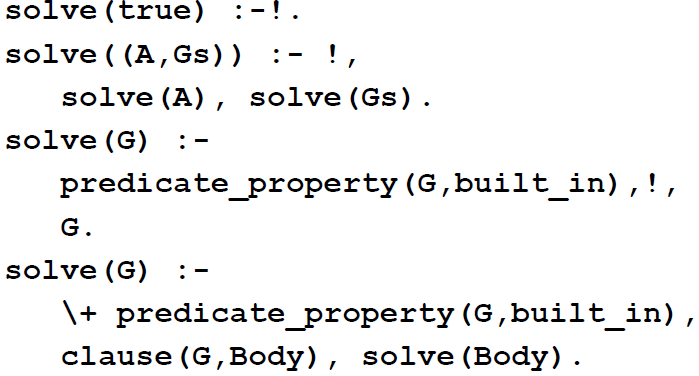
Se suele agregar el cut, resultando:



Hay un problema que es que el predicado clause no se puede utilizar con muchos predicados del sistema, por ejemplo =, <, is, var. A estos predicados se les llama **built in**. Para estos se utiliza un predicado especial que permite saber si es un predicado built in:

*predicate\_property(G,built\_in)*

Agregando este control resulta:



# ESTRUCTURAS INCOMPLETAS

## LISTAS DE DIFERENCIAS

Aaaaa

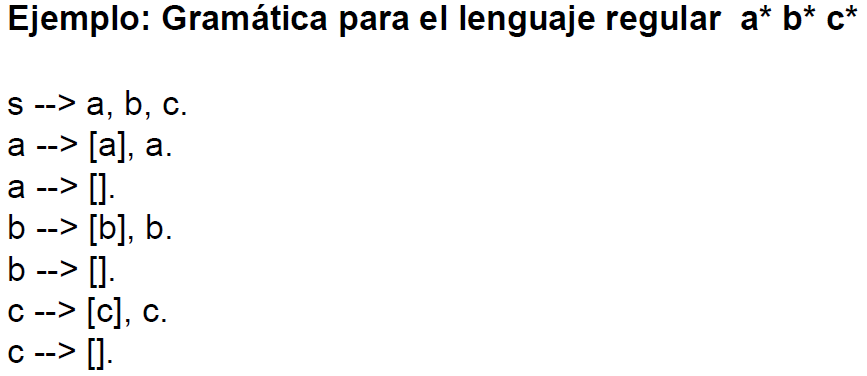
# GRAMÁTICAS LÓGICAS

## DEFINITE CLAUSE GRAMMARS (DCG)

**Definición DCG:** Gramáticas escritas con una sintaxis similar a la usual (en Gramáticas Independientes de Contexto) que se traducen a cláusulas definidas y, por lo tanto, a predicados Prolog.

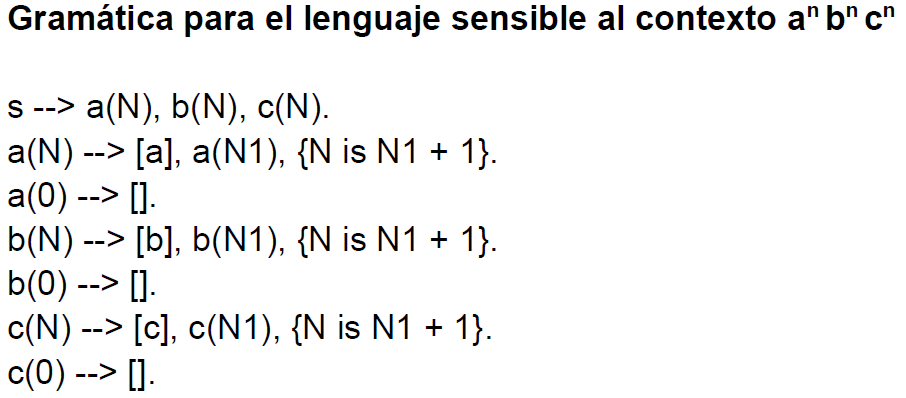
**Sintaxis:**

* Los símbolos no terminales se representan usando átomos Prolog.
* Los símbolos terminales se encierran entre [ ].
* Los símbolos de la parte derecha de la regla van separados por comas.
* Cada regla termina con un punto.



Las DCGs permiten incluir argumentos para los símbolos no terminales y llamadas a predicados dentro de las reglas.

Estas extensiones permiten representar lenguajes sensibles al contexto con gramáticas muy similares a las anteriores.



## TRADUCCIÓN A PREDICADOS

* El intérprete traduce las DCGs (las reconoce por sintaxis) a predicados Prolog “comunes”.
* Asumimos que la gramática trabaja sobre una oración representada como lista de palabras.
* Las palabras son términos, ya que no se analiza su estructura interna.
* Cada símbolo no terminal va a generar un predicado.
* Para mejorar la eficiencia se suelen utilizar listas de diferencias.

# PROGRAMACIÓN LÓGICA INDUCTIVA

## APRENDIZAJE AUTOMÁTICO

## PROGRAMACIÓN LÓGICA INDUCTIVA (ILP)

# RESOLUCIÓN DE JUEGOS

## MINIMAX

Método para determinar la mejor jugada posible a realizar por parte de un jugador (minimizar la pérdida máxima esperada).

Aplica para juegos de 2 jugadores, por turnos, de suma cero e información perfecta.

* El jugador que debe mover es un nodo Max, que quiere maximizar su ganancia.
* La mejor movida para Max es la que maximiza su ganancia, considerando todas las movidas posibles para el otro jugador (Min).
* Suponemos que cada jugador va a hacer el mejor juego posible:
  + Max va a maximizar su ganancia
  + Min va a tratar de que la ganacia de Max sea mínima
* Los nodos del árbol son Max o Min (juega Max o juega Min) alternadamente por niveles.

El valor de un nodo:

* El valor de cada nodo se determina por una función de evaluación o a partir de los valores de sus hijos
* Para un nodo MAX, el valor es el máximo de los valores de los hijos
* Para un nodo MIN, el valor es el mínimo de los valores de los hijos
* Si el nodo de es una hoja (por ser un estado final o por la profundidad definida) se evalúa con la función de evaluación

## PODA CON ALPHA-BETA

Se puede optimizar el algoritmo minimax podando ciertas ramas.

* Alpha: Valor de mi mejor jugada hasta el momento
* Beta: Valor de la mejor jugada de mi oponente hasta el momento
* Si beta ≤ alpha, la mejor movida del oponente obtendría un resultado peor que el que tengo hasta el momento ⇒ podo

Reglas para poda con alpha-beta:

* Alpha Pruning: Descartar la búsqueda debajo de cualquier nodo MIN con beta menor o igual al valor alpha de cualquier de sus ancestros MAX
* Beta Pruning: Descartar la búsqueda debajo de cualquier nodo MAX con alpha mayor o igual al valor beta de cualquiera de sus ancestros MIN