



Programación Lógica

Agustín Torres
Franco Laborda



Conceptos de Lenguajes de Programación
Edición 2024

Introducción

La programación lógica es un tipo de **paradigma** de programación dentro de la programación declarativa.

Una de sus principales características es que se encuentra basada fuertemente en el **cálculo de predicados de primer orden**, mediante la **declaración de hechos y reglas**.

Su enfoque, justamente al ser lógica, es la **resolución de problemas mediante la inferencia**, a través de la **resolución y unificación**.

Como una fuerte parte de la computación está basada en lógica (cómo en el diseño de circuitos digitales booleanos, por ejemplo), tiene sentido que haya surgido la necesidad de un paradigma de tal estilo.

Fundamentos

¿Por qué existe este paradigma?

- Necesidad de un enfoque **declarativo**
En problemas donde es más natural especificar *qué es verdad* en vez de *cómo calcularlo*, la programación lógica permite expresar relaciones y reglas sin detallar procedimientos algorítmicos
- Estudio de la semánticas de los lenguajes, específicamente en la semántica axiomática.
- Este enfoque es especialmente útil en aplicaciones como sistemas expertos, bases de datos, y en el procesamiento de lenguajes naturales, donde la capacidad de razonar y extraer conclusiones es esencial.

Aplicaciones generales

- **Inteligencia artificial**

La manipulación de conocimiento y el razonamiento simbólico son pilares fundamentales de las IAs. La programación lógica facilita la representación y procesamiento de este tipo de información.

- **Procesamiento de lenguaje natural (PLN)**

Facilita el análisis y comprensión del lenguaje humano mediante reglas y estructuras lógicas.

- **Bases de datos y Sistemas de consulta**

El lenguaje SQL comparte principios con la programación lógica, permitiendo consultas declarativas sobre datos.

Conceptos importantes

Sistemas de programación lógica

Los sistemas de programación lógica permiten al programador declarar una colección de axiomas (generando una base de conocimiento), a partir de dónde se pueden probar teoremas.

Luego, el usuario declara un teorema, o un objetivo, a lo que la implementación del lenguaje intenta encontrar una colección de axiomas y pasos de inferencia para que en conjunto, estos impliquen el objetivo, y por tanto lo prueben.

Conceptos importantes

Axiomas y Teoremas

Los *axiomas* son declaraciones fundamentales asumidas como verdaderas, pueden ser tanto hechos simples como reglas que relacionan hechos.

Los *teoremas* (u objetivos) son consultas que se desean probar o satisfacer utilizando los axiomas disponibles.

Conceptos importantes

Cláusulas de Horn

En la inmensa mayoría de lenguajes de programación, los axiomas son escritos en forma de cláusulas de Horn.

Estas consisten en un “Cabezal” (o término H), y un cuerpo de términos B_i de la forma:

$$H \leftarrow B_1, B_2, \dots, B_n$$

Dónde esto significa que si los términos B_i son verdaderos, entonces H también lo es, es decir, H se puede probar en base a los términos B_i .

Conceptos importantes

Resolución

La *resolución* es el mecanismo principal de inferencia en programación lógica.

Combina las cláusulas existentes para derivar nuevas conclusiones y **cancela términos comunes** para simplificar y también deducir información.

Ejemplo:

$C \leftarrow A, B$

$D \leftarrow C$

$D \leftarrow A, B$

Si A y B deducen C y C deduce D, entonces A y B deducen D.

Conceptos importantes

Unificación

La *unificación* es el proceso de hacer que diferentes términos lógicos sean **idénticos**, encontrando sustituciones importantes para las variables.

Esta tiene un papel importante en la resolución, pues las variables libres pueden adquirir valores que satisfagan términos coincidentes.

Ejemplo:

wet(X) <- rainy(X)

rainy(Rochester)

wet(Rochester)

Prolog

Un poco de historia...

*"Prolog fue el hijo exitoso de un matrimonio entre el **procesamiento de lenguajes naturales** y la **demostración automatizada de teoremas**."*

*"El objetivo era tener una herramienta para el **análisis sintáctico y semántico de lenguajes naturales**, usando la **lógica de primer orden** tanto como **lenguaje de programación** como para la **representación de conocimiento**."*

Alain Colmerauer - The birth of Prolog *



* <http://alain.colmerauer.free.fr/alcol/ArchivesPublications/PrologHistory/19november92.pdf>

Prolog

¿A qué debe su popularidad?

Es el lenguaje de programación lógica más utilizado, dentro de algunas de las características que lo hacen a prolog tan popular dentro de los lenguajes de programación lógica se encuentran:

- Antigüedad y Madurez: Prolog fue desarrollado en la década de 1970, lo que le implica una larga historia y una madurez que muchos lenguajes más recientes no tienen.
- Ecosistema y Herramientas: Existen múltiples implementaciones de Prolog, como SWI-Prolog y GNU Prolog, que ofrecen herramientas, bibliotecas y entornos de desarrollo.
- Facilidades de uso: Prolog incorpora características como listas, aritmética y control de flujo imperativo, ampliando su aplicabilidad, así como predicados metalógicos que pueden ser útiles a distintas aplicaciones.

Prolog

Conceptos de Prolog

Un intérprete de Prolog corre en un contexto de una base de datos de cláusulas (cláusulas de Horn) que se asumen como verdaderas.

Cada una de estas cláusulas está compuesta por **términos**, estos pueden ser *constantes*, *variables* o *estructuras*.

- **Constantes:** Son átomos (un identificador que comienza con una letra minúscula, una secuencia de caracteres de "puntuación" o un string) o números.
Ejemplos: foo, my_Const, +, 'Hi, mom'.
- **Variables:** Son un identificador que comienza con una letra mayúscula. Éstas pueden ser instanciadas a valores arbitrarios en tiempo de ejecución como un resultado de unificación, y el alcance de una variable está limitado a la cláusula en la que aparece.
- **Estructuras y Predicados:** Consisten de un átomo llamado *functor* y una lista de argumento. Estos argumentos pueden ser cualquier tipo de términos.

Prolog

Hechos y Reglas

Las cláusulas en un programa de Prolog pueden ser clasificadas cómo hechos o reglas, cada una de ellas es terminada por un punto (.).

Un hecho es una cláusula que no tiene lado derecho, y tiene el aspecto de un término simple.

```
lluvioso(Inglaterra).
```

Una regla por otro lado tiene tanto lado derecho cómo izquierdo.

```
nieva(X) :- llueve(X), mucho_frio(X).
```

El token “:-” es el símbolo de implicación (si llueve y hace frío, entonces nieva), y la coma indica la conjunción (el and).

Prolog

Consultas

En la mayoría de las implementaciones de Prolog, las consultas se indican con el carácter especial `?-`, este es el símbolo de implicación.

Por ejemplo, si tenemos el programa:

```
rainy(seattle)  
rainy(rochester)  
?- rainy(X)
```

El intérprete Prolog va a responder `X = seattle`, esto debido a que la cláusula de `seattle` está primera en la base de datos.

Para ver las demás soluciones debemos ingresar el carácter `“;”` una vez que se nos muestra la primer solución.

Cuando el intérprete ya no encuentre más soluciones, devuelve ***false***.

Prolog

Resolución y Unificación

El principio de resolución nos menciona que si C1 y C2 son clausulas de Horn, y el cabezal de C1 “matchea” con uno de los terminos en el cuerpo de C2, entonces podemos reemplazar el termino de C2 con el cuerpo de C1.

Observemos el siguiente ejemplo:

asiste(pedro, matemática).

asiste(pedro, ciencia).

asiste(juan, arte).

asiste(juan, ciencia).

compañeros_de_clase(X, Y) :- asiste(X, Z), asiste(Y, Z).

Si definimos a X cómo pedro, y Z cómo ciencia, podemos reemplazar el primer término del lado derecho (asiste(X,Z)) con la calusula vacía, ya que al ser un hecho de nuestro conocimiento sabemos que se cumple. Por lo que la nueva regla generada es:

compañeros_de_clase(pedro, Y) :- asiste(Y, ciencia). En otras palabras, reducimos el problema a definir que Y es compañero de pedro si asiste a la clase de ciencia.

Prolog

Resolución y Unificación

En el ejemplo anterior, el proceso de asociar términos con variables es conocido como unificación, las variables que reciben un valor por resultado de la unificación son llamadas variables instanciadas. Mientras que la resolución es la transformación de reglas según su contexto.

Prolog

Listas

Si bien Prolog permite definir listas como en otros lenguajes (por ejemplo `[a,b,c,d,e,f]`), admite una notación extra con una barra vertical que delimita el cabezal del resto de la lista.

Por ejemplo, `[a | [b,c,d,e,f]]` es equivalente a `[a,b,c,d,e,f]`, pero también lo son `[a,b,c,d | [e,f]]` y `[a,b,c,d,e,f | []]`.

El uso de esta notación es de interés cuando el cabezal de la lista es una variable.

Por ejemplo, si queremos ver si un elemento es parte de una lista:

```
% El caracter _ indica que la variable no es necesaria en la cláusula  
member(X, [X | _]). % Si X es el cabezal de la lista retornar True.
```

```
% En este punto sabemos que el cabezal es distinto de X, por lo que hacemos recursión  
member(X, [_ | T]) :- member(X, T).
```

Prolog

Argumentos de entrada y salida

Algo interesante de Prolog es que no distingue entre atributos de entrada y salida (con algunas excepciones), sino que dependiendo de la instanciación de los argumentos que recibe trata de buscar todas las soluciones posibles.

Por ejemplo, dada la siguiente función que añade un elemento a una lista:

```
append([], A, A).
```

```
append([H | T], A, [H | L]) :- append(T, A, L).
```

Cada llamada tendrá una salida diferente:

```
?- append([a,b,c], [d,e], L).
```

```
?- append(X, [d,e], [a,b,c,d,e]).
```

```
?- append([a,b,c], Y, [a,b,c,d,e]).
```

Prolog

Argumentos de entrada y salida

Algo interesante de Prolog es que no distingue entre atributos de entrada y salida (con algunas excepciones), sino que dependiendo de la instanciación de los argumentos que recibe trata de buscar todas las soluciones posibles.

Por ejemplo, dada la siguiente función que añade un elemento a una lista:

```
append([], A, A).
```

```
append([H | T], A, [H | L]) :- append(T, A, L).
```

Cada llamada tendrá una salida diferente:

```
?- append([a,b,c], [d,e], L).
```

```
L = [a,b,c,d,e].
```

```
?- append(X, [d,e], [a,b,c,d,e]).
```

```
X = [a,b,c].
```

```
?- append([a,b,c], Y, [a,b,c,d,e]).
```

```
Y = [d,e].
```

Prolog

Aritmética

En Prolog podemos encontrar los distintos operadores aritméticos, pero con el rol de predicados y no de funciones.

Es por ello que $+(2,3)$, que también se puede escribir cómo $2 + 3$, es una estructura de 2 argumentos, no un llamado a una función. En particular, esta expresión no unifica con 5, justamente por esto último, $(2 + 3) = 5$ es falso.

Para resolver este problema, Prolog cuenta con el uso del predicado `is`, encargado de unificar el primer argumento con el valor aritmético del segundo argumento.

```
?- is(X, 1+2).  
X = 3.  
?- X is 1+2.  
X = 3.  
?- 1+2 is 4-1.  
false.  
?- X is Y.  
ERROR  
?- Y is 1+2, X is Y.  
Y = X, X = 3.
```

¿Por qué cada una retorna el valor indicado?

Prolog

Aritmética

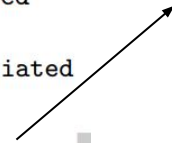
En Prolog podemos encontrar los distintos operadores aritméticos, pero con el rol de predicados y no de funciones.

Es por ello que $+(2,3)$, que también se puede escribir cómo $2 + 3$, es una estructura de 2 argumentos, no un llamado a una función. En particular, esta expresión no unifica con 5, justamente por esto último, $(2 + 3) = 5$ es falso.

Para resolver este problema, Prolog cuenta con el uso del predicado `is`, encargado de unificar el primer argumento con el valor aritmético del segundo argumento.

```
?- is(X, 1+2).  
X = 3.  
?- X is 1+2.  
X = 3.                % infix is also ok  
?- 1+2 is 4-1.  
false.               % 1st argument (1+2) is already instantiated  
?- X is Y.  
ERROR                % 2nd argument (Y) must already be instantiated  
?- Y is 1+2, X is Y.  
Y = X, X = 3.       % Y is instantiated before it is needed
```

Prolog “lee” de
izquierda a
derecha



Prolog

Orden de Búsqueda/Ejecución

Dentro de la lógica formal, hay dos estrategias de búsquedas:

1. **Encadenamiento hacia adelante** (*forward chaining*): Se parte de hechos existentes y se busca derivar el objetivo.
2. **Encadenamiento hacia atrás** (*backward chaining*): Se parte del objetivo y se busca descomponerlo en reglas previas.

En casos donde la cantidad de reglas es muy grande y el número de hechos es pequeño, es posible que *forward chaining* pueda hallar la solución más rápido que *backward chaining*.

Sin embargo, en la mayoría de las circunstancias *backward chaining* es más eficiente.

Prolog utiliza **búsqueda hacia atrás**.

Prolog

Orden de Búsqueda/Ejecución

Prolog sigue un orden de búsqueda en profundidad:

- Exploración de resoluciones posibles desde **izquierda a derecha**.
- **Retroceso** (*Backtracking*): Si una resolución falla, Prolog retrocede al último punto donde puede probar una alternativa.

Para comprender mejor el Backtracking veamos un ejemplo (Ejercicio 3 de la prueba final del curso Programación Lógica 2024):

```
f(a,b).  
f(c,c).
```

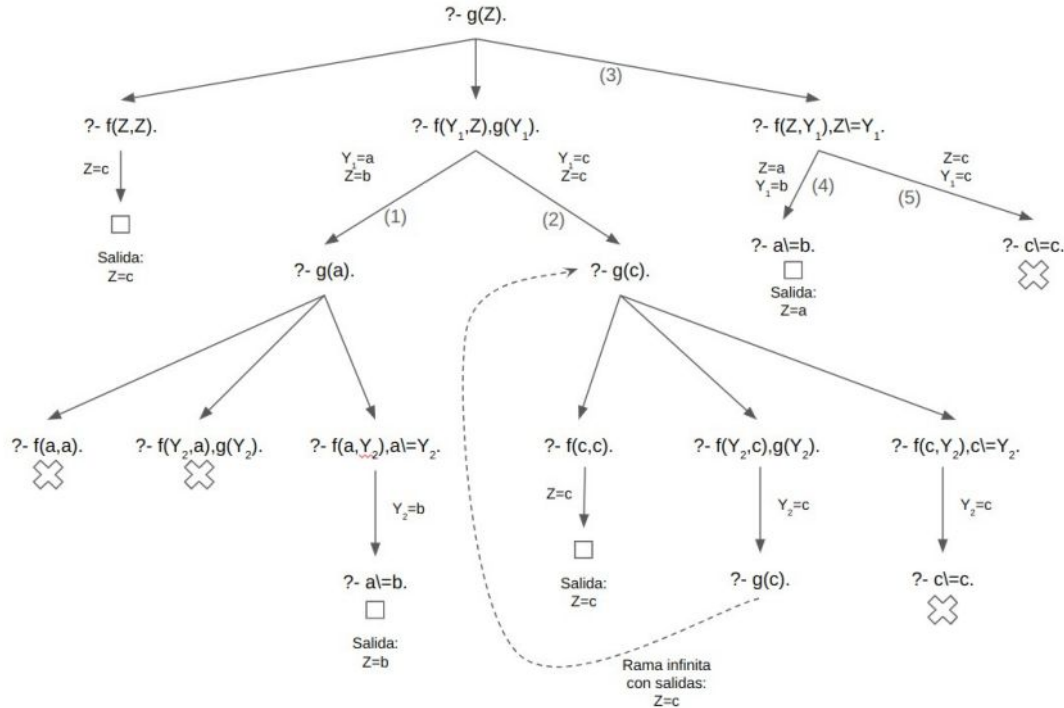
```
g(X) :- f(X,X).  
g(X) :- f(Y,X), g(Y).  
g(X) :- f(X,Y), X\=Y.
```

- a) Dibuje el árbol SLD correspondiente a la consulta **?- g(Z)** suponiendo que la regla de computación toma el átomo de más a la izquierda para la resolución.

Prolog

Orden de Búsqueda/Ejecución

Solución:



Prolog

Control de flujo imperativo - Uso de "Cut"

El cut es un predicado meta-lógico utilizado para modificar el flujo del programa, específicamente, cómo su nombre dice, es utilizado para cortar parte del flujo, específicamente cortar ramas del árbol de unificación generado.

Supongamos que tenemos el siguiente ejemplo:

```
member(X, [X | _]).  
member(X, [_ | T]) :- member(X, T).
```

Cual es el problema de esto a nivel computacional? Suponiendo que utilizamos al predicado member para solamente retornar true o false dependiendo de si encuentra un elemento, sin importar la cantidad de veces.

Prolog

Control de flujo imperativo - Uso de "Cut"

El cut es un predicado meta-lógico utilizado para modificar el flujo del programa, específicamente, cómo su nombre dice, es utilizado para cortar parte del flujo, específicamente cortar ramas del árbol de unificación generado.

Supongamos que tenemos el siguiente ejemplo:

```
member(X, [X | _]).  
member(X, [_ | T]) :- member(X, T).
```

Para el ejemplo `member(5,[1,2,3,4,5,6,7,8,9,10...,5,999999])`, el programa luego de unificar `member(5,[5 | _])`, continúa recorriendo la lista completa hasta llegar a 999999 buscando más soluciones que satisfagan la consulta. Esto puede ser contraproducente en caso de estar utilizando backtracking ya que repetiremos varias veces las mismas ramas de decisión del árbol.

Prolog

Cut

Si un átomo específico aparece en la lista L n veces, entonces el objetivo `?- member(a,L)` puede ser unificado de n formas diferentes. Esto puede no ser ideal dependiendo del caso ya que podría dar lugar a computación desperdiciada, especialmente en listas largas cuando el predicado `member` es seguido por un objetivo que pueda fallar.

```
candidato_primo(X) :- member(X,Candidatos), primo(X).
```

Supongamos que `primo(X)` es “caro” computacionalmente. Para determinar si a es un candidato primo, entonces debemos chequear si es un miembro de `Candidatos` (la lista), y luego chequear si es primo. Si `primo(a)` falla, entonces prolog hará backtracking y volverá a intentar satisfacer `member(a,Candidatos)` nuevamente. Si a se encuentra varias veces en esta lista, este proceso se repetirá varias veces más cuando podría ser evitable.

El siguiente código utiliza `cut`, y no se hará backtracking una vez que se ejecute sobre el predicado padre.

```
member(X, [X | _]) :- !.  
member(X, [_ | T]) :- member(X, T).
```

Prolog

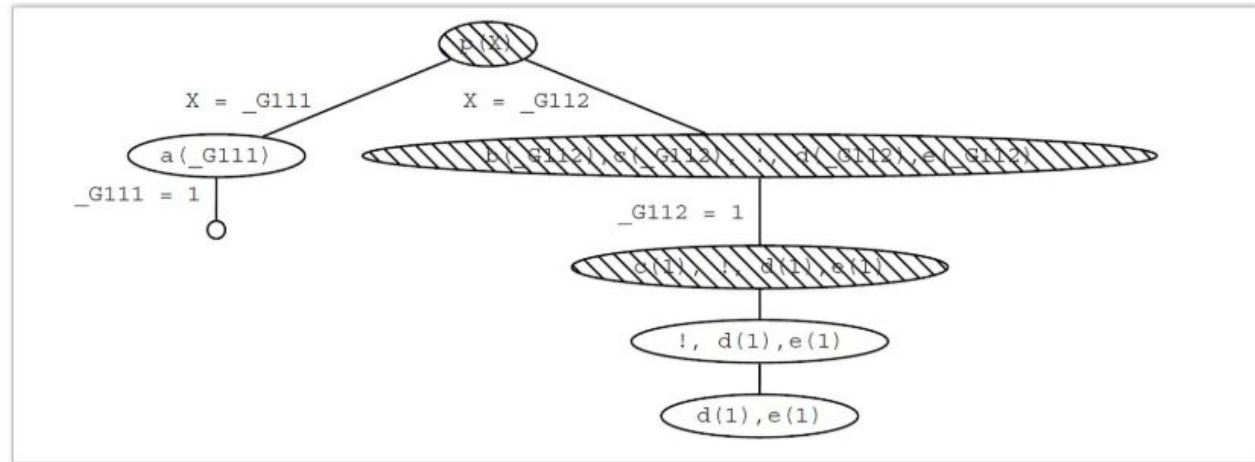
Cut

Ejemplo aplicado a un árbol:

Ejemplo

Ejemplo teórico

```
p(X) :- a(X).  
p(X) :- b(X),c(X),!,d(X),e(X).  
p(X) :- f(X).  
a(1).  
b(1).  
c(1).  
b(2).  
c(2).  
d(2).  
e(2).  
f(3).
```



Prolog

Cut

Cuts Rojos y Verdes

- Como se puede apreciar en los ejemplos anteriores, el operador de corte puede podar soluciones de la consulta realizada.
- **Terminología:**
 - Denominaremos cuts rojos a los que poden nuevas soluciones. \longrightarrow o no funcione de la forma esperada
 - Denominaremos cuts verdes a los demás, es decir a aquellos que no poden soluciones, o que a lo sumo podan soluciones repetidas.

Advertencia

- El operador de corte debe ser empleado con cuidado y analizando las consecuencias en cada situación.
- Si se emplea mal puede causar comportamiento poco predecible.

Prolog

Cut Verde

MAX/3 sin cut:

`max(X,Y,Y) :- X =< Y.`

`max(X,Y,X) :- X > Y.`

MAX/3 con cut:

`max(X,Y,Y) :- X =< Y, !.`

`max(X,Y,X) :- X > Y.`



Prolog

Cut Rojo

MAX/3 :

`max(X,Y,Y) :- X =< Y, !.`

`max(X,Y,X).`

¿Qué pasa con el ejemplo
`max(100,200,100)?`



Prolog

Un ejemplo más...

Veamos como los cuts pueden cambiar el funcionamiento del programa de forma total (Ejercicio 3 de la prueba final del curso Programación Lógica 2024):

Ejercicio 3 [25 puntos]

Considere el siguiente programa Prolog:

```
f(a,b).  
f(c,c).  
  
g(X) :- f(X,X).  
g(X) :- f(Y,X), g(Y).  
g(X) :- f(X,Y), X\=Y.
```

- Dibuje el árbol SLD correspondiente a la consulta **?- g(Z)** suponiendo que la regla de computación toma el átomo de más a la izquierda para la resolución.
- ¿Qué respuestas dará el intérprete Prolog para el objetivo anterior?
- ¿Cuáles son las respuestas si cambiamos la segunda cláusula del predicado `g/1` por la siguiente? Justifique.

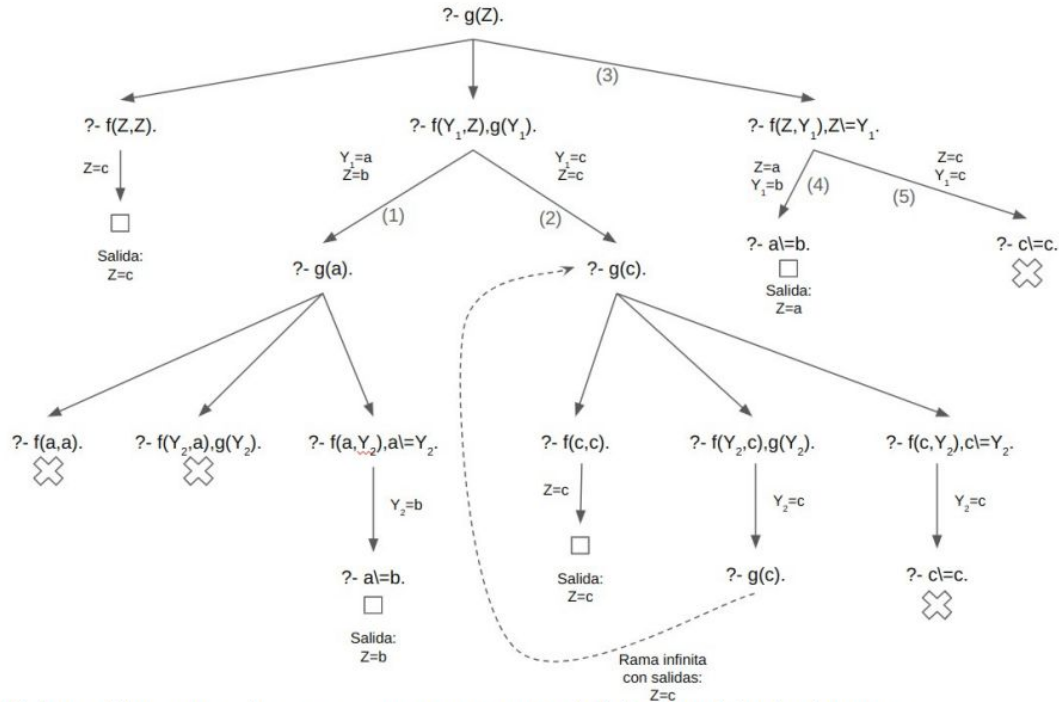
```
g(X) :- f(Y,X), !, g(Y).
```

- ¿Cuáles son las respuestas si cambiamos la primera cláusula del predicado `f/2` por la siguiente? Justifique.

```
f(a,b) :- !.
```


Solución

a)



b) Las salidas son: c, b, c, c, c... cae por una rama infinita devolviendo siempre c.

c) Este cut se activa luego de pasar por el arco marcado con (1), y corta las ramas (2) y (3). Por lo tanto las salidas que quedan son: c, b.

d) Este cut se activa luego de pasar por el arco marcado con (1), pero en este caso corta solo la rama (2). Luego se activa otra vez en el arco (4), y corta la (5). Por lo tanto las salidas que quedan son: c, b, a.

Prolog

Predicados metalógicos

Prolog mantiene en su implementación distintos predicados metalógicos que pueden ayudar a tener programas más dinámicos, potentes y complejos que utilizando únicamente predicados lógicos.

Metapredicados

`atom(+T)` - T es un átomo

`integer(+T)` - T es un entero

`float(+T)` - T es flotante

`number(+T)` - T es entero o flotante

`atomic(+T)` - T es una constante

`var(+T)` - T es una variable **libre**

`compound(+T)` - T es una estructura

Predicados de acceso a términos

- `functor(?T, ?Nombre, ?Aridad)` - Verdadero cuando T es un término
Nombre/Aridad

- `arg(?Arg, +T, ?Valor)` - T es un término, Arg es un valor entre 1 y la aridad de T,
Valor se unifica con el Arg-ésimo argumento de T.

- `?T =.. ?L` - L es una lista cuyo primer elemento es el functor de T, y el resto de sus
elementos son los argumentos de T.

Prolog

Manipulación de Bases de Datos

El contenido de la base de datos se puede alterar de manera dinámica durante la ejecución:

- *assert*: Añade nuevos hechos o reglas a la base de datos
- *retract*: Elimina hechos o reglas de la base de datos.
- *clause*: Permite inspeccionar las reglas en la base de datos, devolviendo el cuerpo de una cláusula coincidente.

Prolog permite modificar su base de datos, esto es útil para sistemas que requieren un conocimiento dinámico.

Prolog

Conclusiones

En este capítulo, se explicó el modelo lógico de la computación comparado con los modelos imperativos y funcionales. Mientras que los programas imperativos usan iteración y efectos secundarios, y los funcionales trabajan mediante la sustitución de parámetros, los programas lógicos resuelven afirmaciones lógicas mediante la unificación de variables y términos.

Se analizó Prolog como el principal lenguaje de programación lógica, abordando conceptos como cláusulas, términos, secuencia de ejecución, manipulación de listas y predicados de orden superior.

En resumen, todos los sistemas informáticos tienen ventajas y limitaciones dependiendo del tipo de situación y necesidades específicas.

¡Muchas Gracias!

