

Tutorial de prólogo

La gramática Prolog o **PRO** en **LOG** ics es un lenguaje de programación lógico y declarativo. Es un ejemplo importante del lenguaje de cuarta generación que admite el paradigma de programación declarativa. Esto es particularmente adecuado para programas que involucran computación **simbólica** o **no numérica** .

Audiencia

Esta referencia ha sido preparada para principiantes para ayudarlos a comprender los conceptos básicos de Prolog.

Requisitos previos

Para este tutorial, se supone que el lector tiene conocimientos previos de codificación.



Prólogo - Introducción

Prolog, como su propio nombre indica, es la forma abreviada de PROGRAMACIÓN LÓGICA. Es un lenguaje de programación lógico y declarativo. Antes de profundizar en los conceptos de Prolog, primero comprendamos qué es exactamente la programación lógica.

La programación lógica es uno de los paradigmas de programación informática, en el que las declaraciones del programa expresan los hechos y reglas sobre diferentes problemas dentro de un sistema de lógica formal. Aquí, las reglas están escritas en forma de cláusulas lógicas, donde el núcleo y el cuerpo están presentes. Por ejemplo, H es la cabeza y B1, B2, B3 son los elementos del cuerpo. Ahora bien, si afirmamos que "H es verdadera, cuando B1, B2, B3 son todas verdaderas", esta es una regla. Por otro lado, los hechos son como las reglas, pero sin cuerpo. Entonces, un ejemplo de hecho es "H es verdadero".

Algunos lenguajes de programación lógica como Datalog o ASP (Programación de conjuntos de respuestas) se conocen como lenguajes puramente declarativos. Estos lenguajes permiten declaraciones sobre lo que el programa debería lograr. No existen instrucciones paso a paso sobre cómo realizar la tarea. Sin embargo, otros lenguajes como Prolog tienen propiedades declarativas y también imperativas. Esto también puede incluir declaraciones de procedimiento como "Para resolver el problema H, realice B1, B2 y B3".

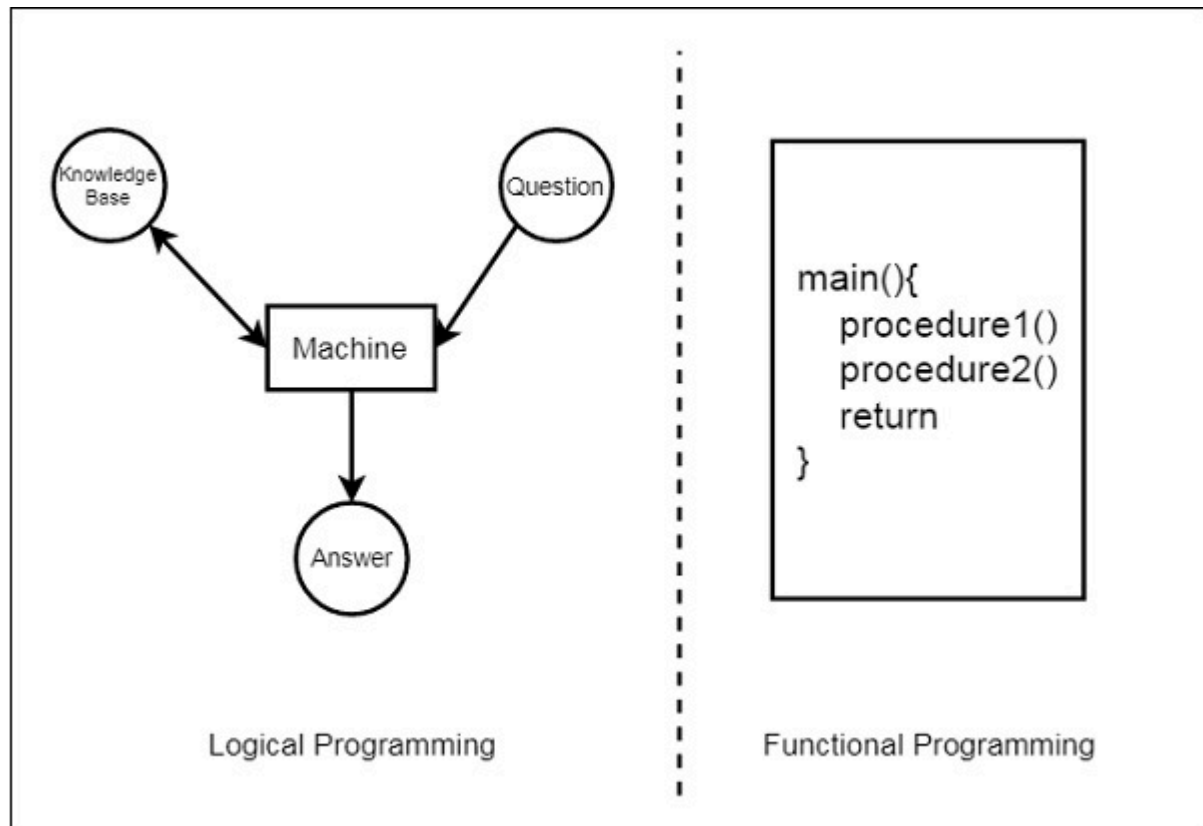
Algunos lenguajes de programación lógica se detallan a continuación:

- ALF (lenguaje de programación funcional de lógica algebraica).
- ASP (Programación del conjunto de respuestas)
- ciclo
- Registro de datos
- FuzzyCLIPS
- Jano
- Parlogar
- Prólogo
- Prólogo++
- ROPA



Programación Lógica y Funcional

Discutiremos sobre las diferencias entre la programación lógica y los lenguajes de programación funcionales tradicionales. Podemos ilustrar estos dos usando el siguiente diagrama:



En esta ilustración, podemos ver que en la Programación Funcional, tenemos que definir los procedimientos y las reglas sobre cómo funcionan. Estos procedimientos funcionan paso a paso para resolver un problema específico basado en el algoritmo. Por otro lado, para la Programación Lógica, proporcionaremos una base de conocimientos. Utilizando esta base de conocimientos, la máquina puede encontrar respuestas a las preguntas planteadas, lo cual es totalmente diferente de la programación funcional.

En programación funcional, tenemos que mencionar cómo se puede resolver un problema, pero en programación lógica tenemos que especificar para qué problema realmente queremos la solución. Luego la programación lógica encuentra automáticamente una solución adecuada que nos ayudará a resolver ese problema específico.

Ahora veamos algunas diferencias más a continuación:

Programación funcional

Programación lógica

La programación funcional sigue la arquitectura de Von-Neumann o utiliza pasos secuenciales.	La programación lógica utiliza modelos abstractos o trata con objetos y sus relaciones.
La sintaxis es en realidad la secuencia de declaraciones como (a, s, I).	La sintaxis es básicamente la fórmula lógica (Cláusulas de Horn).
El cálculo se realiza ejecutando las declaraciones de forma secuencial.	Se calcula restando las cláusulas.
La lógica y los controles se mezclan.	La lógica y los controles se pueden separar.

¿Qué es Prólogo?

La gramática Prolog o **PRO** en **LOG** ics es un lenguaje de programación lógico y declarativo. Es un ejemplo importante del lenguaje de cuarta generación que admite el paradigma de programación declarativa. Esto es particularmente adecuado para programas que involucran computación **simbólica** o **no numérica**. Esta es la razón principal para utilizar Prolog como lenguaje de programación en **Inteligencia Artificial**, donde **la manipulación de símbolos** y **la manipulación de inferencias** son las tareas fundamentales.

En Prolog, no necesitamos mencionar la forma en que se puede resolver un problema, solo necesitamos mencionar cuál es el problema, para que Prolog lo resuelva automáticamente. Sin embargo, en Prolog se supone que debemos dar pistas como **método de solución**.

El lenguaje Prolog tiene básicamente tres elementos diferentes:

Hechos : el hecho es un predicado que es verdadero, por ejemplo, si decimos "Tom es el hijo de Jack", entonces es un hecho.

Reglas : las reglas son extinciones de hechos que contienen cláusulas condicionales. Para satisfacer una regla se deben cumplir estas condiciones. Por ejemplo, si definimos una regla como:

```
grandfather(X, Y) :- father(X, Z), parent(Z, Y)
```

Esto implica que para que X sea abuelo de Y, Z debería ser padre de Y y X debería ser padre de Z.

Preguntas : y para ejecutar un programa de prólogo, necesitamos algunas preguntas, y esas preguntas pueden responderse mediante los hechos y reglas dados.

Historia del prólogo

La herencia de prolog incluye la investigación sobre demostradores de teoremas y algunos otros sistemas de deducción automatizados que se desarrollaron en las décadas de 1960 y 1970. El mecanismo de inferencia del Prólogo se basa en el principio de resolución de Robinson, propuesto en 1965, y en el mecanismo de extracción de respuestas de Green (1968). Estas ideas se unieron con fuerza con la llegada de los procedimientos de resolución lineal.

Los procedimientos explícitos de resolución lineal dirigidos a objetivos dieron impulso al desarrollo de un sistema de programación lógica de propósito general. El **primer** Prolog fue el **Marseille Prolog** basado en el trabajo de **Colmerauer** en el año 1970. El manual de este intérprete de Marseille Prolog (Roussel, 1975) fue la primera descripción detallada del lenguaje Prolog.

Prolog también se considera un lenguaje de programación de cuarta generación que admite el paradigma de programación declarativa. El conocido proyecto informático japonés de quinta generación, anunciado en 1981, adoptó Prolog como lenguaje de desarrollo y, por lo tanto, atrajo considerable atención sobre el lenguaje y sus capacidades.

Algunas aplicaciones de Prolog

Prolog se utiliza en varios dominios. Desempeña un papel vital en el sistema de automatización. A continuación se muestran algunos otros campos importantes donde se utiliza Prolog:

- Recuperación inteligente de bases de datos
- Comprensión del lenguaje natural
- Idioma de especificación
- Aprendizaje automático
- Planificación de robots
- Sistema de automatización
- Resolución de problemas

Prólogo: configuración del entorno

En este capítulo, discutiremos cómo instalar Prolog en nuestro sistema.

Versión prólogo

En este tutorial, estamos usando GNU Prolog, Versión: 1.4.5

Página web oficial

Este es el sitio web oficial de GNU Prolog donde podemos ver todos los detalles necesarios sobre GNU Prolog y también obtener el enlace de descarga.

<http://www.gprolog.org/>

Enlace de descarga directa

A continuación se muestran los enlaces de descarga directa de GNU Prolog para Windows. Para otros sistemas operativos como Mac o Linux, puede obtener los enlaces de descarga visitando el sitio web oficial (el enlace se proporciona arriba):

<http://www.gprolog.org/#download> (sistema de 32 bits)

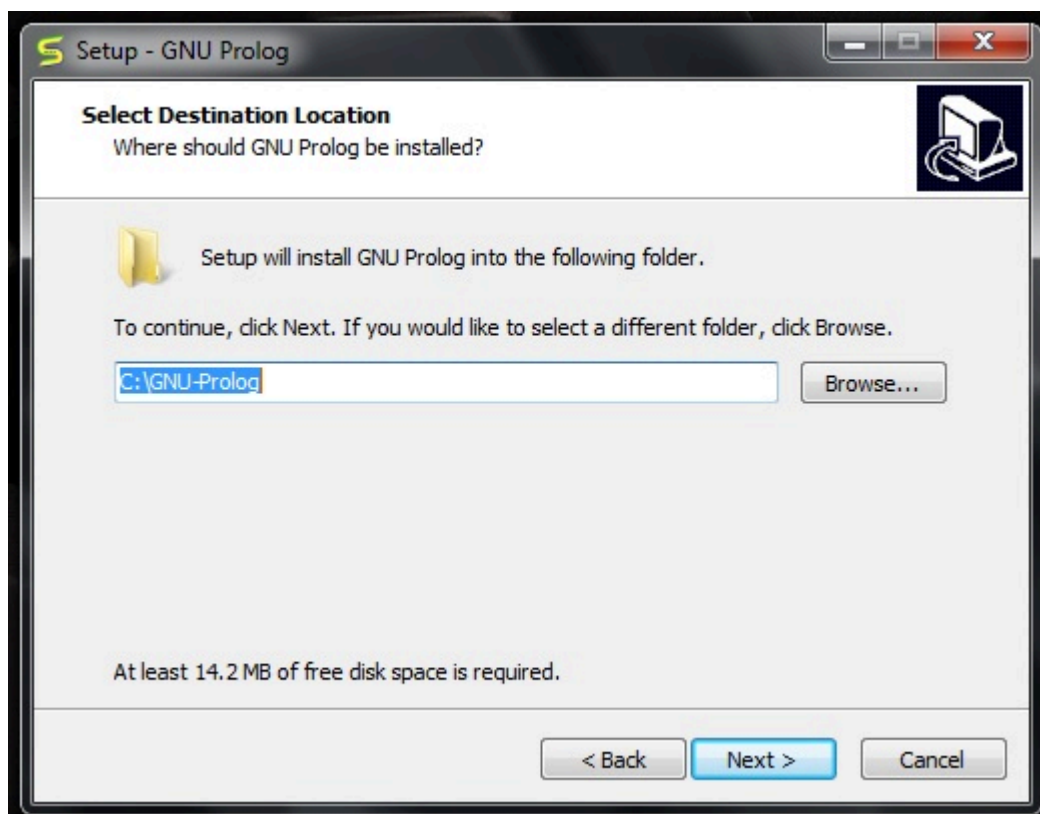
Guía de instalación

- Descargue el archivo exe y ejecútelo.
- Verá la ventana como se muestra a continuación, luego haga clic en **Siguiente** :

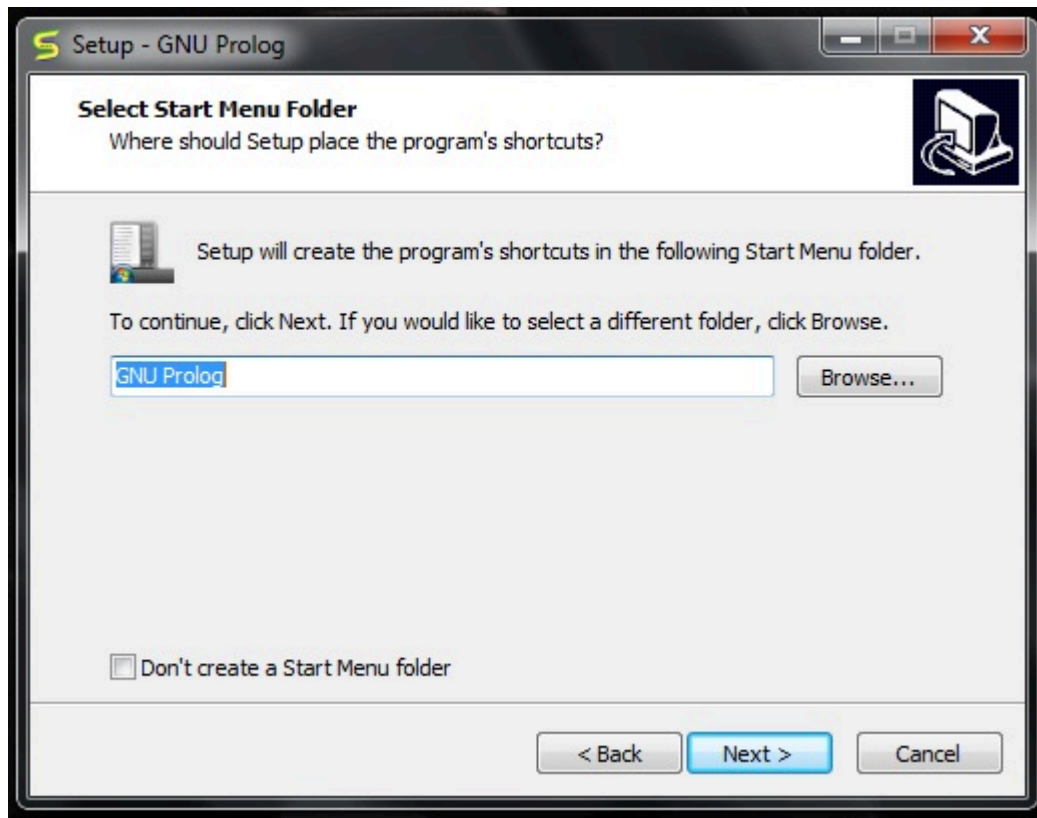




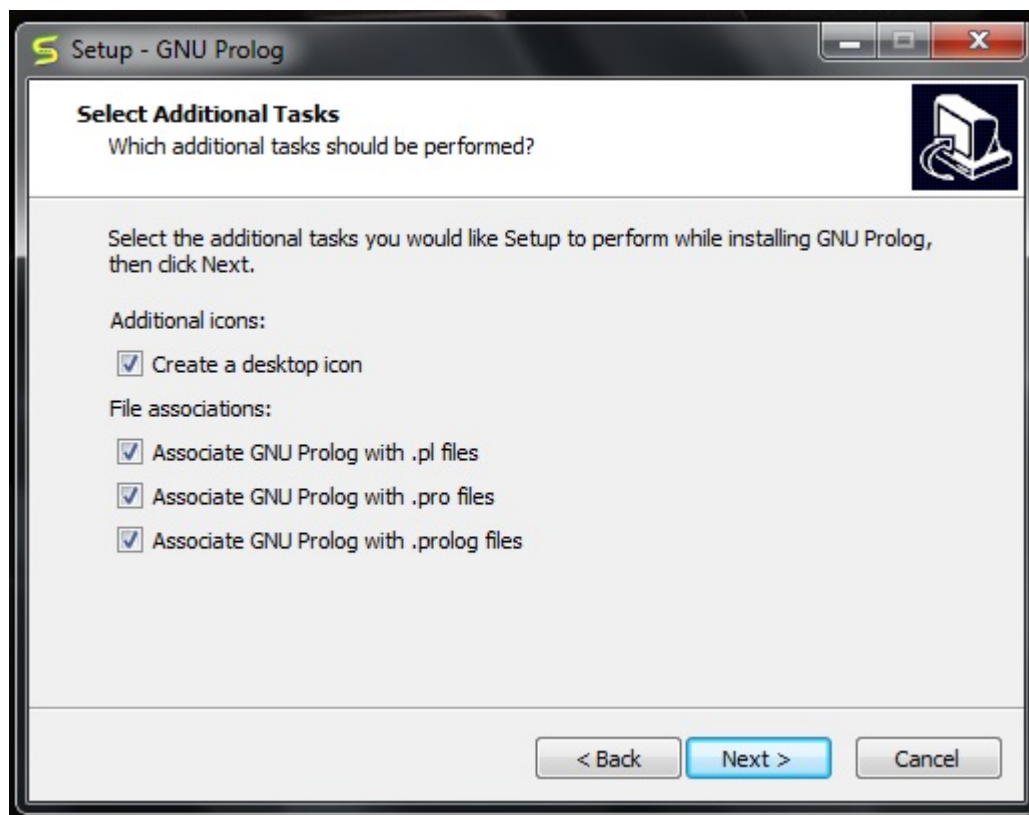
Seleccione **el directorio** adecuado donde desea instalar el software; de lo contrario, déjelo instalar en el directorio predeterminado. Luego haga clic en **siguiente** .



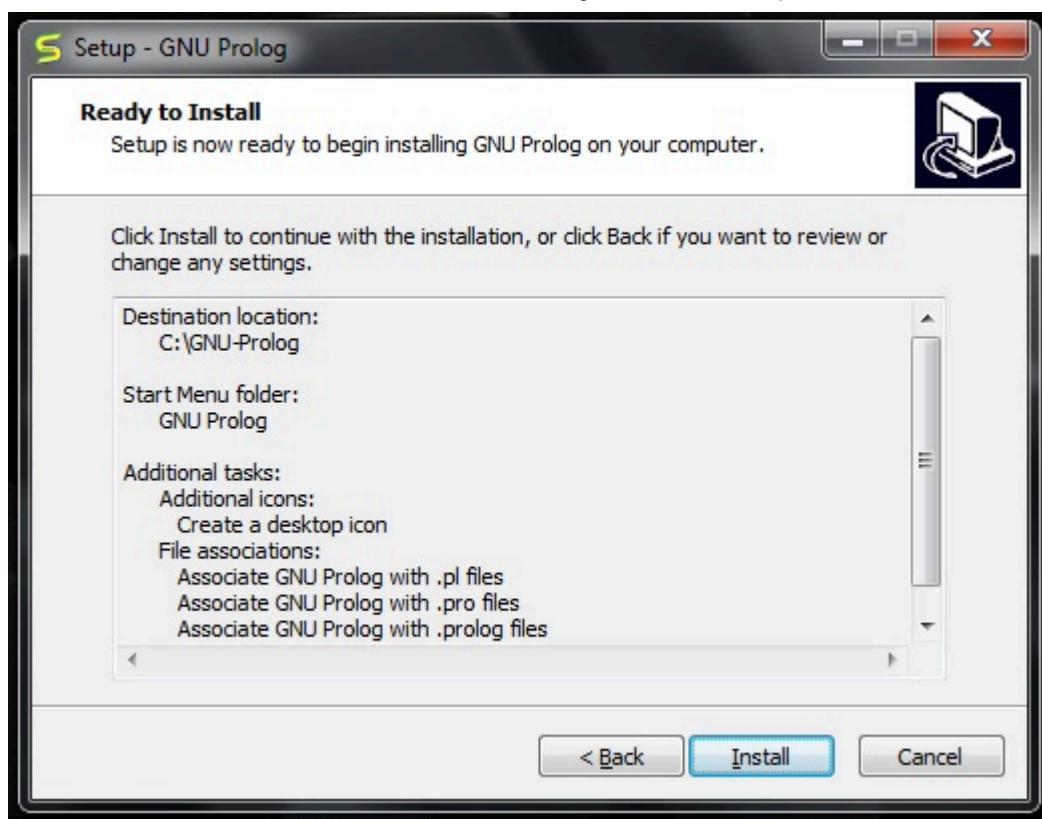
Obtendrá la siguiente pantalla, simplemente vaya a **siguiente** .



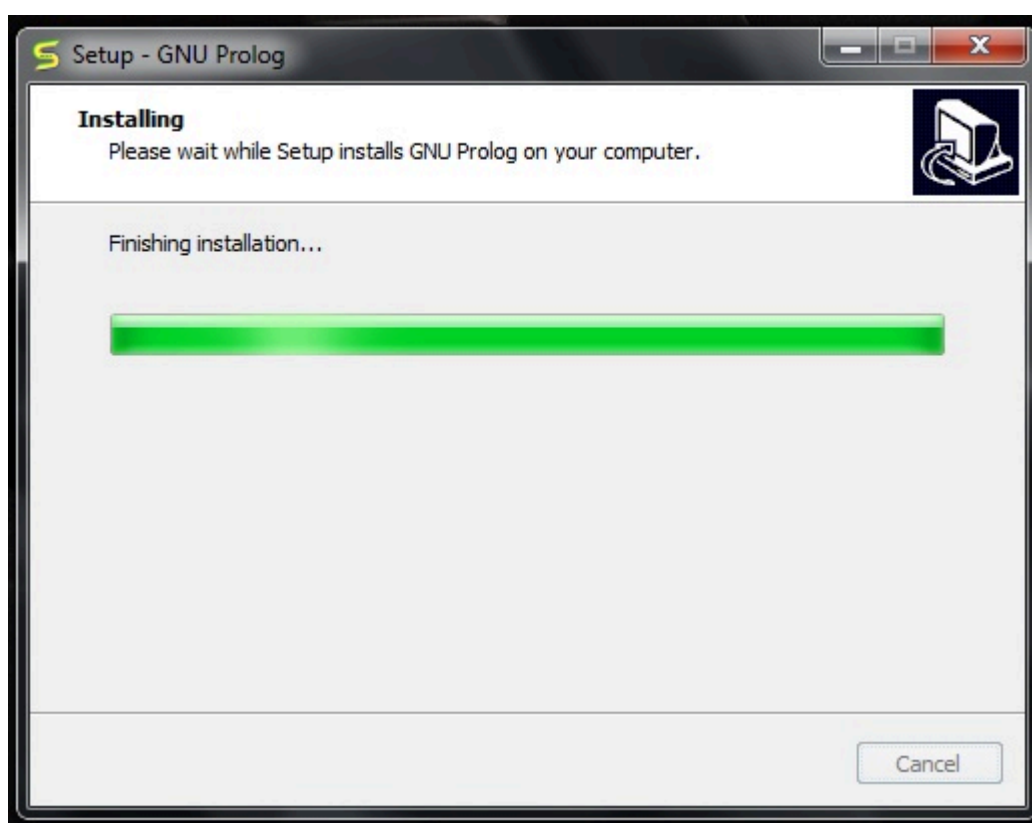
Puede verificar la siguiente pantalla y **marcar/desmarcar** las casillas correspondientes; de lo contrario, puede dejarla como predeterminada. Luego haga clic en **siguiente** .



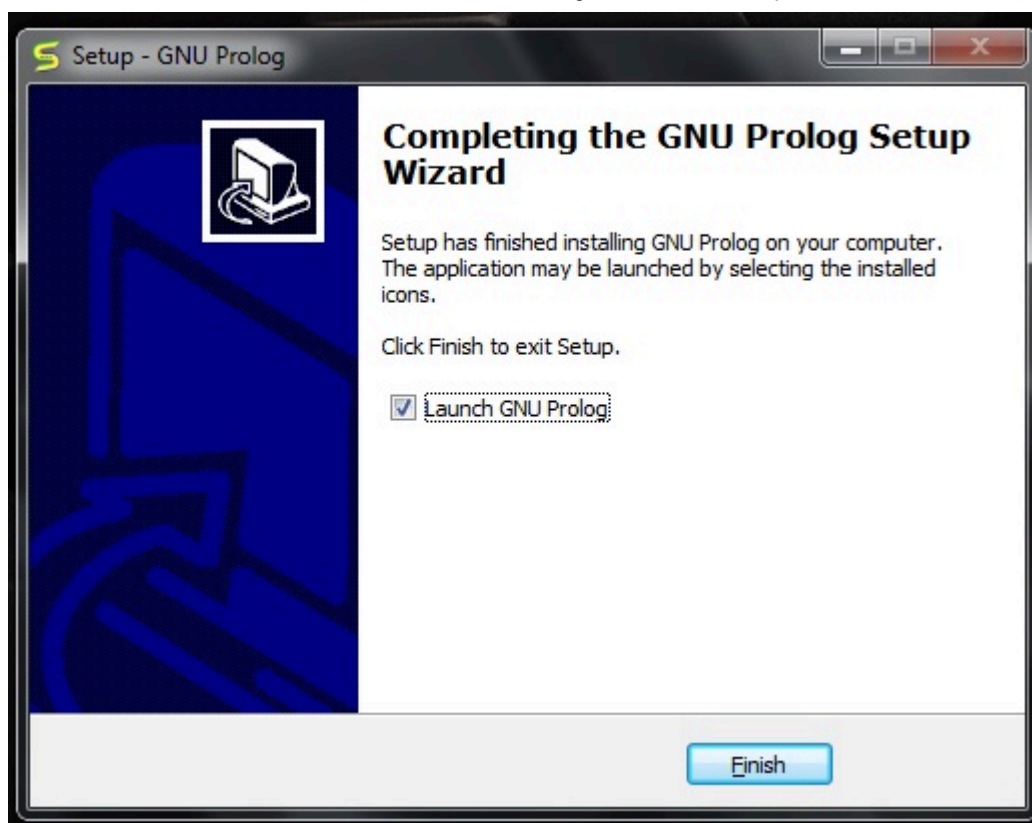
En el siguiente paso, verá la siguiente pantalla, luego haga clic en **Instalar** .



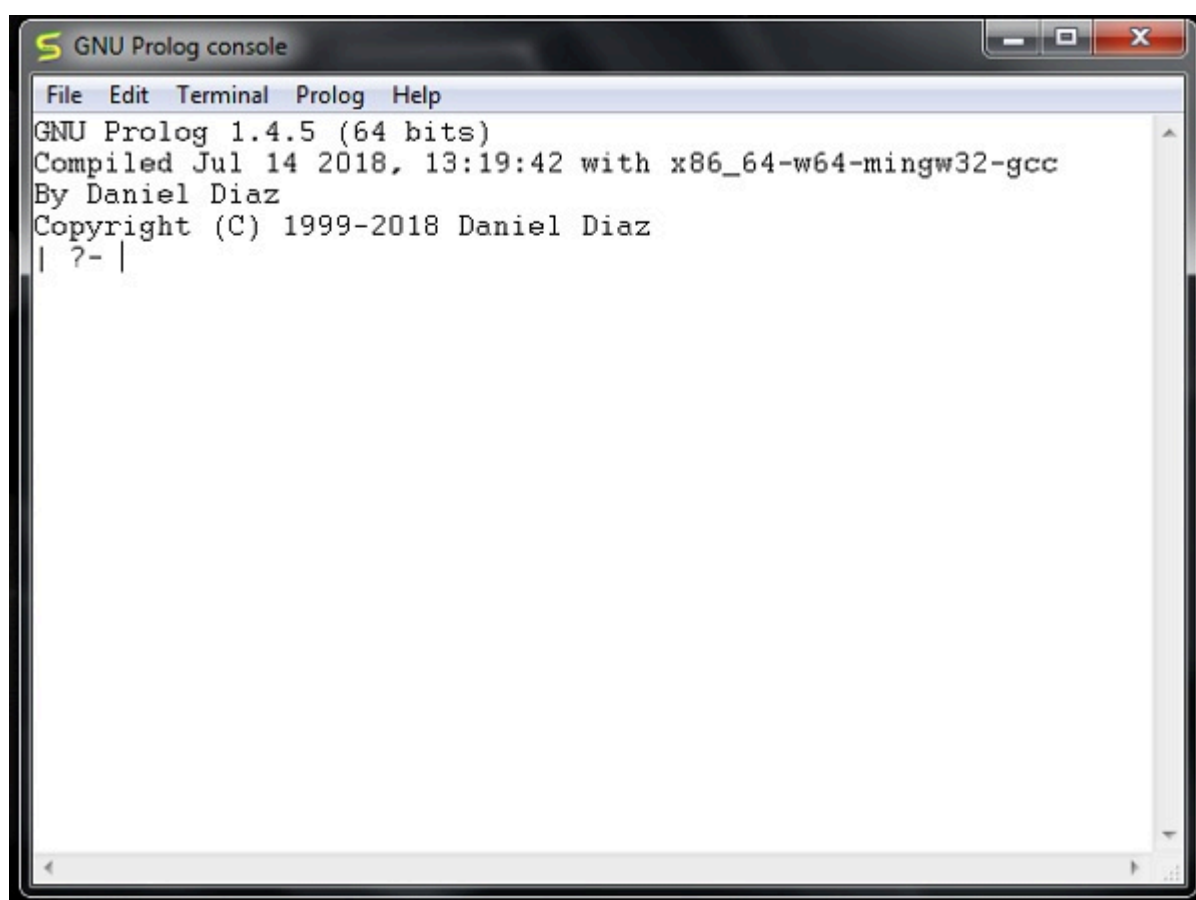
Luego espere a que finalice el proceso de instalación.



Finalmente haga clic en **Finalizar** para iniciar GNU Prolog.



El prólogo de GNU se instaló correctamente como se muestra a continuación:



Prólogo - Hola mundo

En la sección anterior, hemos visto cómo instalar GNU Prolog. Ahora veremos cómo escribir un programa **Hello World** sencillo en nuestro entorno Prolog.

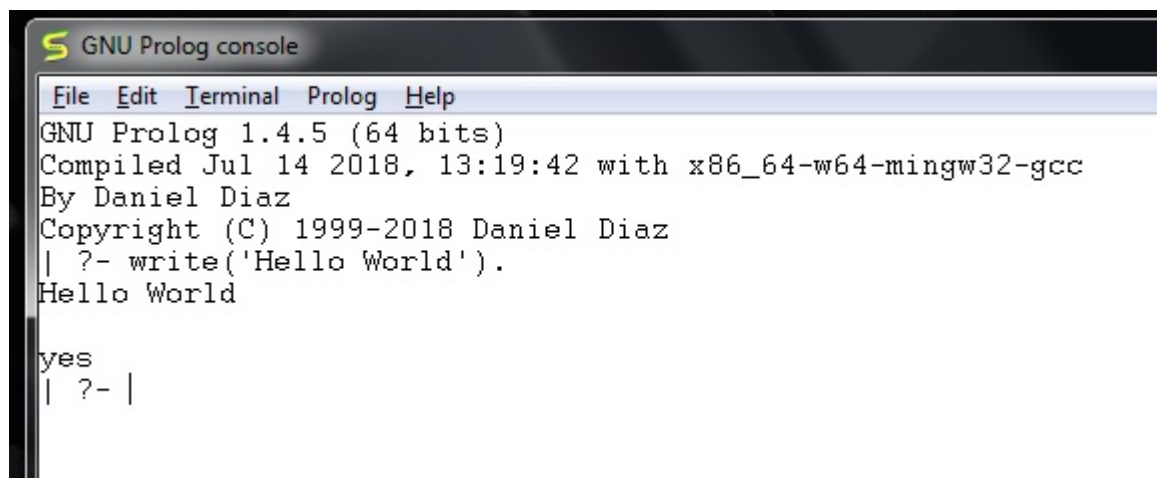
Programa Hola Mundo

Después de ejecutar el prólogo de GNU, podemos escribir el programa hola mundo directamente desde la consola. Para hacerlo, debemos escribir el comando de la siguiente manera:

```
write('Hello World').
```

Nota : Después de cada línea, debe utilizar un símbolo de punto (.) para mostrar que la línea ha finalizado.

El resultado correspondiente será el que se muestra a continuación:



```
GNU Prolog console
File Edit Terminal Prolog Help
GNU Prolog 1.4.5 (64 bits)
Compiled Jul 14 2018, 13:19:42 with x86_64-w64-mingw32-gcc
By Daniel Diaz
Copyright (C) 1999-2018 Daniel Diaz
| ?- write('Hello World').
Hello World
yes
| ?- |
```

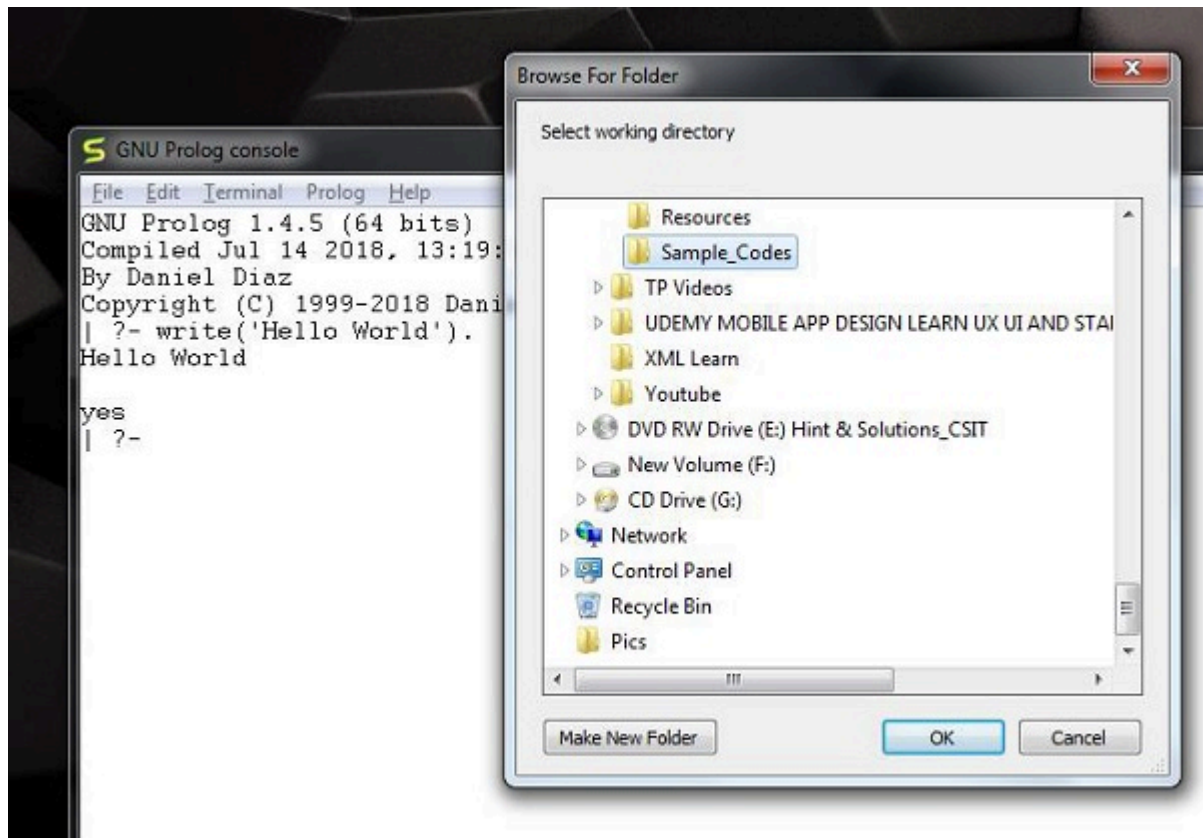
Ahora veamos cómo ejecutar el archivo de script de Prolog (la extensión es *.pl) en la consola de Prolog.

Antes de ejecutar el archivo *.pl, debemos almacenar el archivo en el directorio al que apunta la consola GNU Prolog; de lo contrario, simplemente cambie el directorio siguiendo los siguientes pasos:

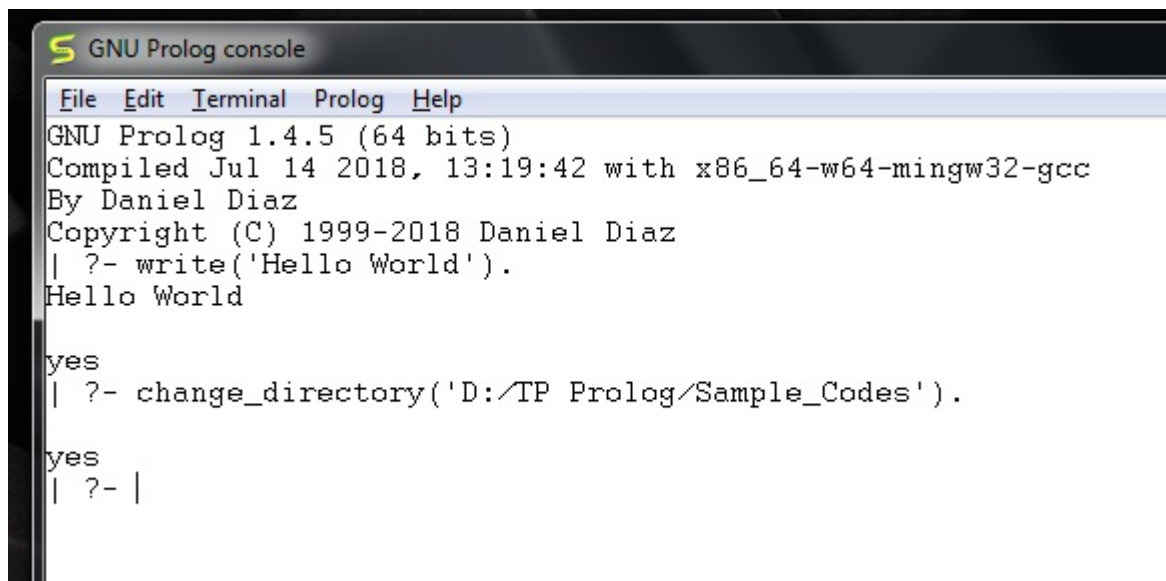
Paso 1 : desde la consola de Prolog, vaya a Archivo > Cambiar directorio y luego haga clic en ese menú.

Paso 2 : seleccione la carpeta adecuada y presione **Aceptar** .





Ahora podemos ver en la consola de prólogo que hemos cambiado el directorio con éxito.



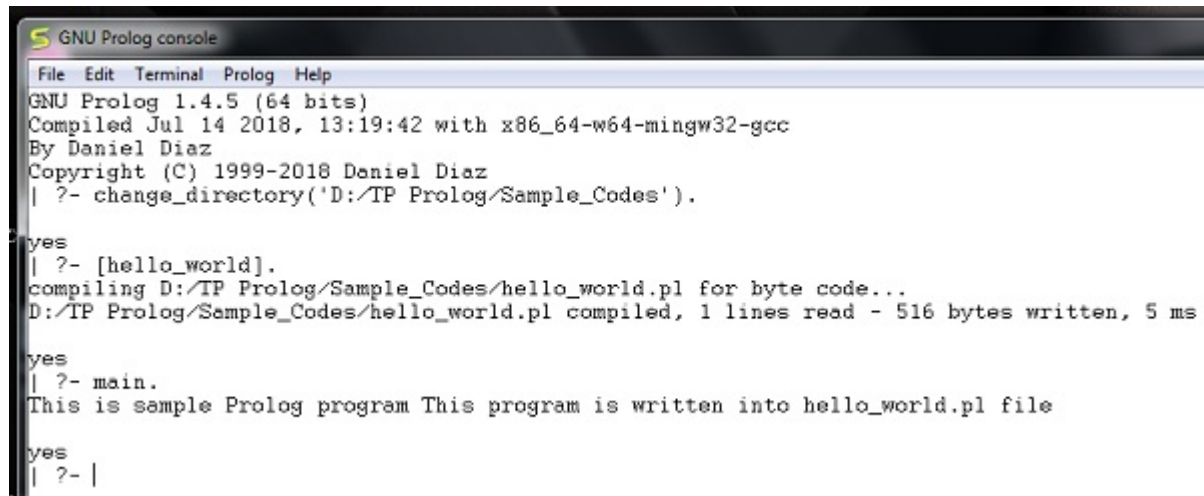
Paso 3 : ahora cree un archivo (la extensión es *.pl) y escriba el código de la siguiente manera:

```
main :- write('This is sample Prolog program'),  
        write(' This program is written into hello_world.pl file').
```

Ahora ejecutemos el código. Para ejecutarlo, debemos escribir el nombre del archivo de la siguiente manera:

[hello_world]

El resultado es el siguiente:



```
GNU Prolog console
File Edit Terminal Prolog Help
GNU Prolog 1.4.5 (64 bits)
Compiled Jul 14 2018, 13:19:42 with x86_64-w64-mingw32-gcc
By Daniel Diaz
Copyright (C) 1999-2018 Daniel Diaz
| ?- change_directory('D:/TP Prolog/Sample_Codes').
yes
| ?- [hello_world].
compiling D:/TP Prolog/Sample_Codes/hello_world.pl for byte code...
D:/TP Prolog/Sample_Codes/hello_world.pl compiled, 1 lines read - 516 bytes written, 5 ms
yes
| ?- main.
This is sample Prolog program This program is written into hello_world.pl file
yes
| ?- |
```

Prólogo - Conceptos básicos

En este capítulo, obtendremos algunos conocimientos básicos sobre Prolog. Entonces pasaremos al primer paso de nuestra programación de Prolog.

Los diferentes temas que se cubrirán en este capítulo son:

Base de conocimientos : esta es una de las partes fundamentales de la programación lógica. Veremos en detalle sobre la Base de Conocimiento y cómo ayuda en la programación lógica.

Hechos, reglas y consultas : estos son los componentes básicos de la programación lógica. Obtendremos un conocimiento detallado sobre hechos y reglas, y también veremos algunos tipos de consultas que se utilizarán en la programación lógica.

Aquí, discutiremos sobre los componentes esenciales de la programación lógica. Estos componentes básicos son **los hechos, las reglas y las consultas** .

Hechos

Podemos definir el hecho como una relación explícita entre objetos y las propiedades que estos objetos podrían tener. De modo que los hechos son incondicionalmente verdaderos por naturaleza. Supongamos que tenemos algunos hechos como se detallan a continuación:

- tom es un gato
- A Kunal le encanta comer pasta.
- el pelo es negro
- A Nawaz le encanta jugar.
- Pratyusha es vaga.

Estos son algunos hechos que son incondicionalmente ciertos. En realidad, estas son afirmaciones que debemos considerar como verdaderas.

A continuación se presentan algunas pautas para escribir hechos:

- Los nombres de propiedades/relaciones comienzan con letras minúsculas



- El nombre de la relación aparece como primer término.
- Los objetos aparecen como argumentos separados por comas entre paréntesis.
- Un período "." debe poner fin a un hecho.
- Los objetos también comienzan con letras minúsculas. También pueden comenzar con dígitos (como 1234) y pueden ser cadenas de caracteres entre comillas, por ejemplo, color (penink, 'red').
- foneno(agnibha, 1122334455). También se le llama predicado o cláusula.

Sintaxis

La sintaxis de los hechos es la siguiente:

```
relation(object1,object2...).
```

Ejemplo

El siguiente es un ejemplo del concepto anterior:

```
cat(tom).  
loves_to_eat(kunal,pasta).  
of_color(hair,black).  
loves_to_play_games(nawaz).  
lazy(pratyusha).
```

Normas

Podemos definir la regla como una relación implícita entre objetos. Entonces los hechos son condicionalmente ciertos. Entonces, cuando una condición asociada es verdadera, entonces el predicado también es verdadero. Supongamos que tenemos algunas reglas como las que se detallan a continuación:

- Lili es feliz si baila.
- Tom tiene hambre si busca comida.
- Jack y Bili son amigos si a ambos les encanta jugar al cricket.
- Irá a jugar si la escuela está cerrada y él está libre.



Estas son algunas reglas que son **condicionalmente** verdaderas, por lo que cuando el lado derecho es verdadero, entonces el lado izquierdo también es verdadero.

Aquí el símbolo (:-) se pronunciará como "Si" o "está implícito en". Esto también se conoce como símbolo del cuello, el lado izquierdo de este símbolo se llama Cabeza y el lado derecho se llama Cuerpo. Aquí podemos usar coma (,) que se conoce como conjunción, y también podemos usar punto y coma, que se conoce como disyunción.

Sintaxis

```
rule_name(object1, object2, ...) :- fact/rule(object1,  
object2, ...)
```

Suppose a clause is like :

P :- Q;R.

This can also be written as

P :- Q.

P :- R.

If one clause is like :

P :- Q,R;S,T,U.

Is understood as

P :- (Q,R);(S,T,U).

Or can also be written as:

P :- Q,R.

P :- S,T,U.

Ejemplo

```
happy(lili) :- dances(lili).
```

```
hungry(tom) :- search_for_food(tom).
```

```
friends(jack, bili) :- lovesCricket(jack), lovesCricket(bili).
```

```
goToPlay(ryan) :- isClosed(school), free(ryan).
```

Consultas

Las consultas son algunas preguntas sobre las relaciones entre objetos y propiedades de los objetos. Entonces, la pregunta puede ser cualquier cosa, como se indica a continuación:

- ¿Tom es un gato?
- ¿A Kunal le encanta comer pasta?
- ¿Lili está feliz?
- ¿Irá Ryan a jugar?

Entonces, de acuerdo con estas consultas, el lenguaje de programación Logic puede encontrar la respuesta y devolverlas.

Base de conocimientos en programación lógica

En esta sección veremos qué es la base de conocimientos en programación lógica.

Bueno, como sabemos, hay tres componentes principales en la programación lógica: **hechos, reglas y consultas**. Entre estos tres, si recopilamos los hechos y las reglas en su conjunto, se forma una **base de conocimientos**. Entonces podemos decir que la **base de conocimientos** es una **colección de hechos y reglas**.

Ahora veremos cómo escribir algunas bases de conocimiento. Supongamos que tenemos nuestra primera base de conocimientos llamada KB1. Aquí en KB1 tenemos algunos datos. Los hechos se utilizan para enunciar cosas que son incondicionalmente ciertas en el dominio de interés.

Base de conocimientos 1

Supongamos que tenemos algún conocimiento de que Priya, Tiyasha y Jaya son tres niñas, y entre ellas, Priya sabe cocinar. Intentemos escribir estos hechos de una manera más genérica como se muestra a continuación:

```
girl(priya).  
girl(tiyasha).  
girl(jaya).  
can_cook(priya).
```

Nota : aquí hemos escrito el nombre en letras minúsculas, porque en Prolog, una cadena que comienza con una letra mayúscula indica una **variable**.

Ahora podemos utilizar esta base de conocimientos planteando algunas consultas. "¿Priya es una niña?", responderá "sí", "¿jamini es una niña?" entonces responderá "No", porque no sabe quién es jamini. Nuestra siguiente pregunta es "¿Puede Priya cocinar?", dirá "sí", pero si le hacemos la misma pregunta a Jaya, dirá "No".

Producción

GNU Prolog 1.4.5 (64 bits)

Compiled Jul 14 2018, 13:19:42 with x86_64-w64-mingw32-gcc

By Daniel Diaz

Copyright (C) 1999-2018 Daniel Diaz

| ?- change_directory('D:/TP Prolog/Sample_Codes').

yes

| ?- [kb1]

.

compiling D:/TP Prolog/Sample_Codes/kb1.pl for byte code...

D:/TP Prolog/Sample_Codes/kb1.pl compiled, 3 lines read - 489 bytes written, 10 ms

yes

| ?- girl(priya)

.

yes

| ?- girl(jamini).

no

| ?- can_cook(priya).

yes

| ?- can_cook(jaya).

no

| ?-

Veamos otra base de conocimiento, donde tenemos algunas reglas. Las reglas contienen información que es condicionalmente cierta sobre el dominio de interés. Supongamos que nuestra base de conocimientos es la siguiente:

sing_a_song(ananya).

listens_to_music(rohit).

listens_to_music(ananya) :- sing_a_song(ananya).

happy(ananya) :- sing_a_song(ananya).



```
happy(rohit) :- listens_to_music(rohit).  
playes_guitar(rohit) :- listens_to_music(rohit).
```

Entonces, hay algunos hechos y reglas dados anteriormente. Los dos primeros son hechos, pero el resto son reglas. Como sabemos que Ananya canta una canción, esto implica que también escucha música. Entonces, si preguntamos "¿Ananya escucha música?", la respuesta será verdadera. De manera similar, "¿Rohit está feliz?", esto también será cierto porque escucha música. Pero si nuestra pregunta es "¿Ananya toca la guitarra?", entonces, según la base de conocimientos, responderá "No". Estos son algunos ejemplos de consultas basadas en esta base de conocimiento.

Producción

```
| ?- [kb2].  
compiling D:/TP Prolog/Sample_Codes/kb2.pl for byte code...  
D:/TP Prolog/Sample_Codes/kb2.pl compiled, 6 lines read - 1066 bytes written, 15 ms  
  
yes  
| ?- happy(rohit).  
  
yes  
| ?- sing_a_song(rohit).  
  
no  
| ?- sing_a_song(ananya).  
  
yes  
| ?- playes_guitar(rohit).  
  
yes  
| ?- playes_guitar(ananya).  
  
no  
| ?- listens_to_music(ananya).  
  
yes  
| ?-
```

Base de conocimientos 3

Los hechos y reglas de Knowledge Base 3 son los siguientes:

```
can_cook(priya).
can_cook(jaya).
can_cook(tiyasha).

likes(priya,jaya) :- can_cook(jaya).
likes(priya,tiyasha) :- can_cook(tiyasha).
```

Supongamos que queremos ver los miembros que saben cocinar, podemos usar una **variable** en nuestra consulta. Las variables deben comenzar con letras mayúsculas. En el resultado, se mostrará uno por uno. Si presionamos enter, saldrá; de lo contrario, si presionamos punto y coma (;), mostrará el siguiente resultado.

Veamos una demostración práctica para entender cómo funciona.

Producción

```
| ?- [kb3].
compiling D:/TP Prolog/Sample_Codes/kb3.pl for byte code...
D:/TP Prolog/Sample_Codes/kb3.pl compiled, 5 lines read - 737 bytes written, 22 ms
warning: D:/TP Prolog/Sample_Codes/kb3.pl:1: redefining procedure can_cook/1
        D:/TP Prolog/Sample_Codes/kb1.pl:4: previous definition

yes
| ?- can_cook(X).

X = priya ? ;

X = jaya ? ;

X = tiyasha

yes
| ?- likes(priya,X).

X = jaya ? ;

X = tiyasha

yes
| ?-
```


Prólogo - Relaciones

La relación es una de las características principales que debemos mencionar adecuadamente en Prolog. Estas relaciones pueden expresarse como hechos y reglas. Después de eso veremos sobre las relaciones familiares, cómo podemos expresar las relaciones familiares en Prolog y también veremos las relaciones recursivas de la familia.

Crearemos la base de conocimientos creando hechos y reglas, y realizaremos consultas sobre ellos.

Relaciones en prólogo

En los programas Prolog, especifica la relación entre los objetos y las propiedades de los objetos.

Supongamos que hay una declaración, "Amit tiene una bicicleta", entonces en realidad estamos declarando la relación de propiedad entre dos objetos: uno es Amit y el otro es una bicicleta.

Si hacemos la pregunta: "¿Tiene Amit una bicicleta?", en realidad estamos tratando de averiguar sobre una relación.

Hay varios tipos de relaciones, algunas de las cuales también pueden ser reglas. Una regla puede descubrir una relación incluso si la relación no está definida explícitamente como un hecho.

Podemos definir una relación de hermano de la siguiente manera:

Dos personas son hermanos, si,

- Ambos son hombres.
- Tienen el mismo padre.

Ahora considere que tenemos las siguientes frases:

- padre(sudip, piyus).
- padre (sudip, raj).
- macho(piyus).



- hombre (raj).
- hermano(X,Y) :- padre(Z,X), padre(Z,Y),masculino(X), masculino(Y)

Estas cláusulas pueden darnos la respuesta de que piyus y raj son hermanos, pero aquí obtendremos tres pares de resultados. Ellos son: (piyus, piyus), (piyus, raj), (raj, raj). Para estos pares, las condiciones dadas son verdaderas, pero para los pares (piyus, piyus), (raj, raj), en realidad no son hermanos, son las mismas personas. Entonces tenemos que crear las cláusulas adecuadamente para formar una relación.

La relación revisada puede ser la siguiente:

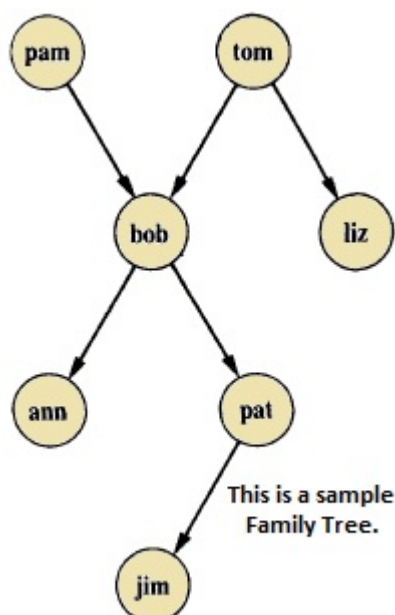
A y B son hermanos si:

- A y B, ambos son hombres.
- tienen el mismo padre
- tienen la misma madre
- A y B no son iguales

Relación familiar en Prolog

Aquí veremos la relación familiar. Este es un ejemplo de relación compleja que se puede formar usando Prolog. Queremos hacer un árbol genealógico, y eso se mapeará en hechos y reglas, luego podremos realizar algunas consultas sobre ellos.

Supongamos que el árbol genealógico es el siguiente:



Aquí desde este árbol podemos entender que existen pocas relaciones. Aquí Bob es hijo de Pam y Tom, y Bob también tiene dos hijos: Ann y Pat. Bob tiene un hermano, Liz, cuyo padre también es Tom. Entonces queremos hacer predicados de la siguiente manera:

Predicados

- padre (pam, bob).
- padre (tom, bob).
- padre (tom, liz).
- padre (bob, ann).
- padre (bob, palmadita).
- padre (pat, jim).
- padre (bob, peter).
- padre (peter, jim).

Nuestro ejemplo ha ayudado a ilustrar algunos puntos importantes:

- Hemos definido la relación entre padres indicando las n-tuplas de objetos según la información proporcionada en el árbol genealógico.
- El usuario puede consultar fácilmente al sistema Prolog sobre las relaciones definidas en el programa.
- Un programa Prolog consta de cláusulas terminadas con un punto.
- Los argumentos de las relaciones pueden (entre otras cosas) ser: objetos concretos o constantes (como pat y jim), u objetos generales como X e Y. Los objetos del primer tipo en nuestro programa se denominan átomos. Los objetos del segundo tipo se llaman variables.
- Las preguntas al sistema constan de uno o más objetivos.

Algunos datos se pueden escribir de dos maneras diferentes, como el sexo de los miembros de la familia, que se puede escribir en cualquiera de las formas:

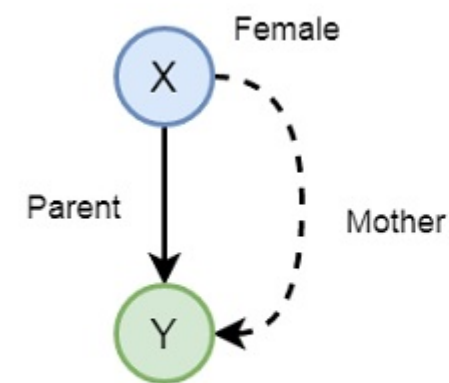
- mujer (pam).
- macho(tom).
- macho (bobo).
- mujer (liz).

- hembra(palmadita).
- mujer (ana).
- hombre(jim).

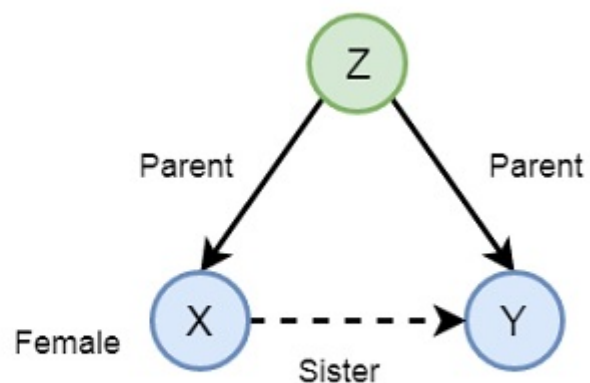
O en el siguiente formulario:

- sexo(pam, femenino).
- sexo(tom, masculino).
- sexo(bob, masculino).
- ... etcétera.

Ahora, si queremos establecer una relación entre madre y hermana, podemos escribir como se indica a continuación:



Mother Relationship



Sister Relationship

En la sintaxis de Prolog, podemos escribir:

- madre(X,Y) :- padre(X,Y), mujer(X).
- hermana(X,Y) :- padre(Z,X), padre(Z,Y), mujer(X), X \== Y.

Ahora veamos la demostración práctica:

Base de conocimientos (familia.pl)

```
female(pam).
female(liz).
```

```

female(pat).
female(ann).
male(jim).
male(bob).
male(tom).
male(peter).
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
parent(bob,peter).
parent(peter,jim).
mother(X,Y):- parent(X,Y),female(X).
father(X,Y):- parent(X,Y),male(X).
haschild(X):- parent(X,_).
sister(X,Y):- parent(Z,X),parent(Z,Y),female(X),X\==Y.
brother(X,Y):-parent(Z,X),parent(Z,Y),male(X),X\==Y.

```

Producción

```

| ?- [family].
compiling D:/TP Prolog/Sample_Codes/family.pl for byte code...
D:/TP Prolog/Sample_Codes/family.pl compiled, 23 lines read - 3088 bytes written, 9 ms

yes
| ?- parent(X,jim).

X = pat ? ;

X = peter

yes
| ?-
mother(X,Y).

X = pam
Y = bob ? ;

X = pat

```

Y = jim ? ;

no

| ?- haschild(X).

X = pam ? ;

X = tom ? ;

X = tom ? ;

X = bob ? ;

X = bob ? ;

X = pat ? ;

X = bob ? ;

X = peter

yes

| ?- sister(X,Y).

X = liz

Y = bob ? ;

X = ann

Y = pat ? ;

X = ann

Y = peter ? ;

X = pat

Y = ann ? ;

X = pat

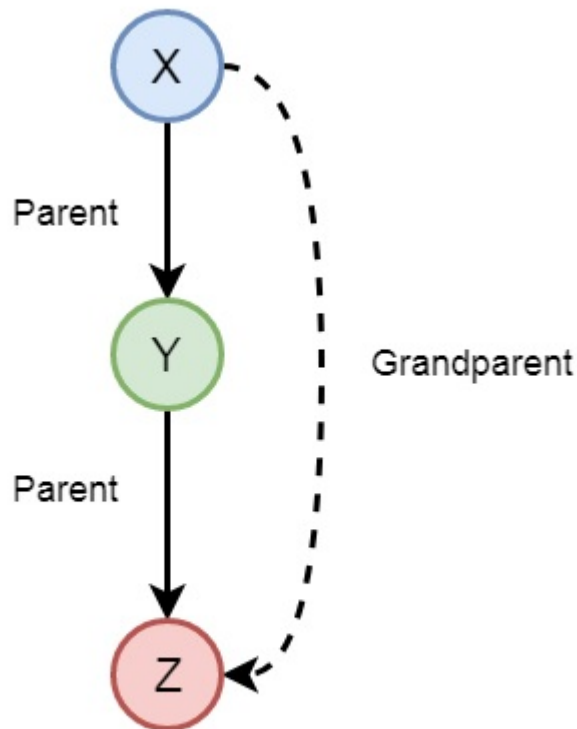
Y = peter ? ;

(16 ms) no

| ?-



Ahora veamos algunas relaciones más que podemos establecer a partir de las relaciones anteriores de una familia. Entonces, si queremos establecer una relación de abuelos, se puede formar de la siguiente manera:



Grandparent Relationship

También podemos crear otras relaciones como esposa, tío, etc. Podemos escribir las relaciones como se indica a continuación:

- `abuelo(X,Y) :- padre(X,Z), padre(Z,Y).`
- `abuela(X,Z):- madre(X,Y), padre(Y,Z).`
- `abuelo(X,Z) :- padre(X,Y), padre(Y,Z).`
- `esposa (X, Y): - padre (X, Z), padre (Y, Z), mujer (X), hombre (Y).`
- `tío (X,Z): - hermano (X,Y), padre (Y,Z).`

Así que escribamos un programa prólogo para ver esto en acción. Aquí también veremos el trace para rastrear la ejecución.

Base de conocimientos (family_ext.pl)

```
female(pam).  
female(liz).  
female(pat).
```

```
female(ann).

male(jim).
male(bob).
male(tom).
male(peter).

parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).

parent(bob,pat).
parent(pat,jim).
parent(bob,peter).
parent(peter,jim).

mother(X,Y):- parent(X,Y),female(X).
father(X,Y):-parent(X,Y),male(X).
sister(X,Y):-parent(Z,X),parent(Z,Y),female(X),X\==Y.
brother(X,Y):-parent(Z,X),parent(Z,Y),male(X),X\==Y.
grandparent(X,Y):-parent(X,Z),parent(Z,Y).
grandmother(X,Z):-mother(X,Y),parent(Y,Z).
grandfather(X,Z):-father(X,Y),parent(Y,Z).
wife(X,Y):-parent(X,Z),parent(Y,Z),female(X),male(Y).
uncle(X,Z):-brother(X,Y),parent(Y,Z).
```

Producción

```
| ?- [family_ext].
compiling D:/TP Prolog/Sample_Codes/family_ext.pl for byte code...
D:/TP Prolog/Sample_Codes/family_ext.pl compiled, 27 lines read - 4646 bytes written, :

| ?- uncle(X,Y).

X = peter
Y = jim ? ;

no
| ?- grandparent(X,Y).
```

```
X = pam
Y = ann ? ;

X = pam
Y = pat ? ;

X = pam
Y = peter ? ;

X = tom
Y = ann ? ;

X = tom
Y = pat ? ;

X = tom
Y = peter ? ;

X = bob
Y = jim ? ;

X = bob
Y = jim ? ;

no
| ?- wife(X,Y).

X = pam
Y = tom ? ;

X = pat
Y = peter ? ;

(15 ms) no
| ?-
```

Seguimiento de la salida

En Prolog podemos rastrear la ejecución. Para rastrear la salida, debe ingresar al modo de rastreo escribiendo "trace". Luego, en el resultado podemos ver que solo estamos rastreando "¿pam es madre de quién?". Vea el resultado del rastreo

tomando X = pam e Y como variable, Y será bob como respuesta. Para salir del modo de rastreo presione "notrace".

Programa

```
| ?- [family_ext].
compiling D:/TP Prolog/Sample_Codes/family_ext.pl for byte code...
D:/TP Prolog/Sample_Codes/family_ext.pl compiled, 27 lines read - 4646 bytes

(16 ms) yes
| ?- mother(X,Y).

X = pam
Y = bob ? ;

X = pat
Y = jim ? ;

no
| ?- trace.
The debugger will first creep -- showing everything (trace)

yes
{trace}
| ?- mother(pam,Y).
  1 1 Call: mother(pam,_23) ?
  2 2 Call: parent(pam,_23) ?
  2 2 Exit: parent(pam,bob) ?
  3 2 Call: female(pam) ?
  3 2 Exit: female(pam) ?
  1 1 Exit: mother(pam,bob) ?

Y = bob

(16 ms) yes
{trace}
| ?- notrace.
The debugger is switched off
```

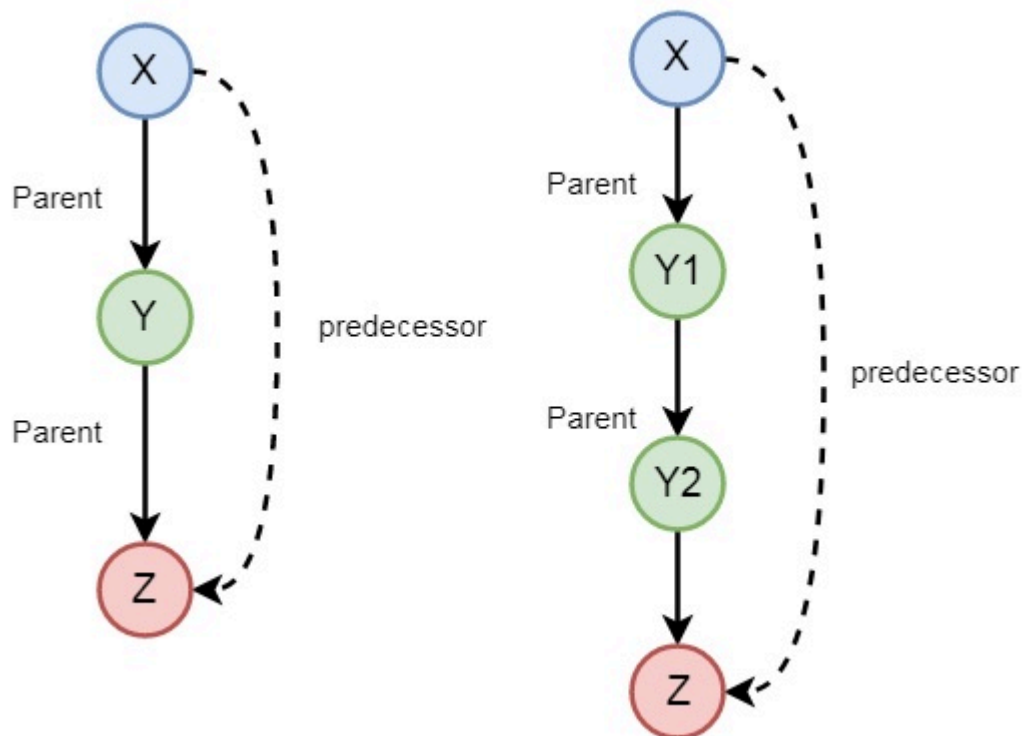


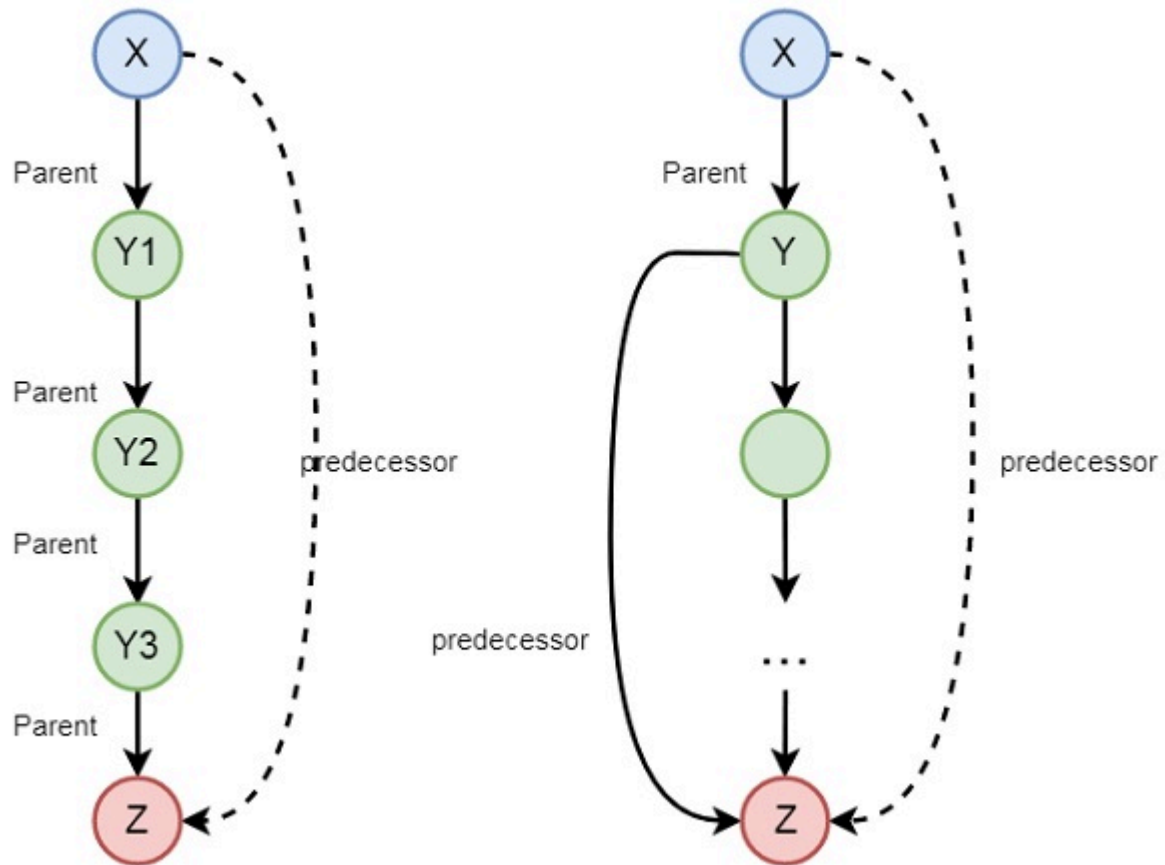
yes

| ?-

Recursión en la relación familiar

En el apartado anterior hemos visto que podemos definir algunas relaciones familiares. Estas relaciones son de naturaleza estática. También podemos crear algunas relaciones recursivas que se pueden expresar a partir de la siguiente ilustración:





Entonces podemos entender que la relación predecesora es recursiva. Podemos expresar esta relación usando la siguiente sintaxis:

```
predecessor(X, Z) :- parent(X, Z).
predecessor(X, Z) :- parent(X, Y),predecessor(Y, Z).
```

Ahora veamos la demostración práctica.

Base de conocimientos (family_rec.pl)

```
female(pam).
female(liz).
female(pat).
female(ann).

male(jim).
male(bob).
male(tom).
male(peter).

parent(pam,bob).
```

```

parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
parent(bob,peter).
parent(peter,jim).

predecessor(X, Z) :- parent(X, Z).
predecessor(X, Z) :- parent(X, Y),predecessor(Y, Z).

```

Producción

```

| ?- [family_rec].
compiling D:/TP Prolog/Sample_Codes/family_rec.pl for byte code...
D:/TP Prolog/Sample_Codes/family_rec.pl compiled, 21 lines read - 1851 bytes written, 1

yes
| ?- predecessor(peter,X).

X = jim ? ;

no
| ?- trace.
The debugger will first creep -- showing everything (trace)

yes
{trace}
| ?- predecessor(bob,X).
  1 1 Call: predecessor(bob,_23) ?
  2 2 Call: parent(bob,_23) ?
  2 2 Exit: parent(bob,ann) ?
  1 1 Exit: predecessor(bob,ann) ?

X = ann ? ;
  1 1 Redo: predecessor(bob,ann) ?
  2 2 Redo: parent(bob,ann) ?
  2 2 Exit: parent(bob,pat) ?
  1 1 Exit: predecessor(bob,pat) ?

X = pat ? ;

```



```
1 1 Redo: predecessor(bob,pat) ?
2 2 Redo: parent(bob,pat) ?
2 2 Exit: parent(bob,peter) ?
1 1 Exit: predecessor(bob,peter) ?
```

X = peter ? ;

```
1 1 Redo: predecessor(bob,peter) ?
2 2 Call: parent(bob,_92) ?
2 2 Exit: parent(bob,ann) ?
3 2 Call: predecessor(ann,_23) ?
4 3 Call: parent(ann,_23) ?
4 3 Fail: parent(ann,_23) ?
4 3 Call: parent(ann,_141) ?
4 3 Fail: parent(ann,_129) ?
3 2 Fail: predecessor(ann,_23) ?
2 2 Redo: parent(bob,ann) ?
2 2 Exit: parent(bob,pat) ?
3 2 Call: predecessor(pat,_23) ?
4 3 Call: parent(pat,_23) ?
4 3 Exit: parent(pat,jim) ?
3 2 Exit: predecessor(pat,jim) ?
1 1 Exit: predecessor(bob,jim) ?
```

X = jim ? ;

```
1 1 Redo: predecessor(bob,jim) ?
3 2 Redo: predecessor(pat,jim) ?
4 3 Call: parent(pat,_141) ?
4 3 Exit: parent(pat,jim) ?
5 3 Call: predecessor(jim,_23) ?
6 4 Call: parent(jim,_23) ?
6 4 Fail: parent(jim,_23) ?
6 4 Call: parent(jim,_190) ?
6 4 Fail: parent(jim,_178) ?
5 3 Fail: predecessor(jim,_23) ?
3 2 Fail: predecessor(pat,_23) ?
2 2 Redo: parent(bob,pat) ?
2 2 Exit: parent(bob,peter) ?
3 2 Call: predecessor(peter,_23) ?
4 3 Call: parent(peter,_23) ?
4 3 Exit: parent(peter,jim) ?
3 2 Exit: predecessor(peter,jim) ?
1 1 Exit: predecessor(bob,jim) ?
```



X = jim ?

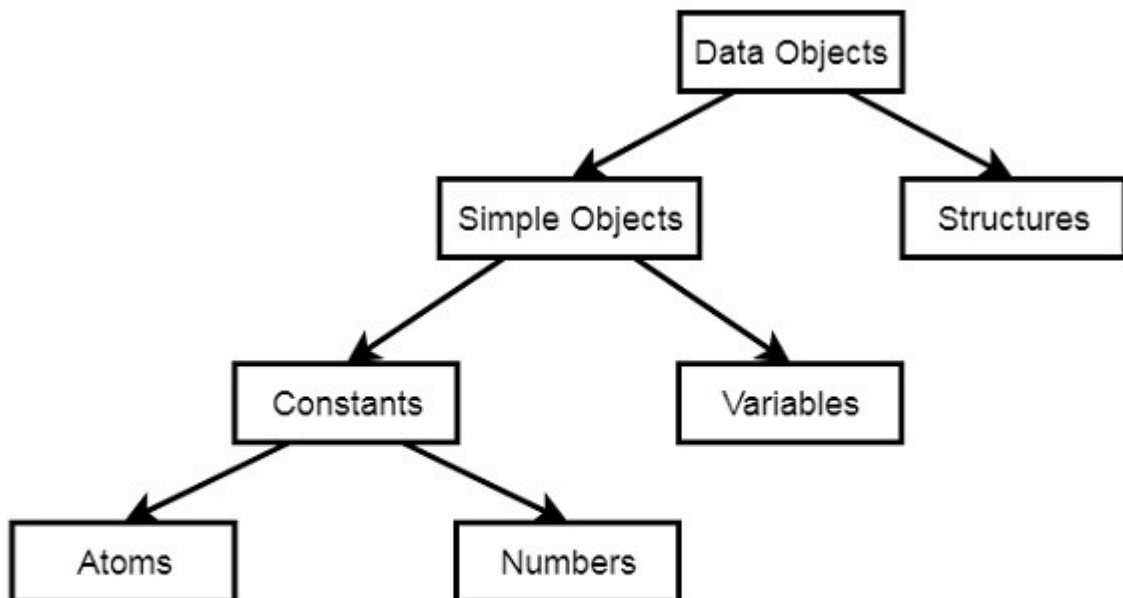
(78 ms) yes

{trace}

| ?-

Prólogo - Objetos de datos

En este capítulo, aprenderemos sobre los objetos de datos en Prolog. Se pueden dividir en algunas categorías diferentes como se muestra a continuación:



Data objects in Prolog

A continuación se muestran algunos ejemplos de diferentes tipos de objetos de datos:

- Átomos: tom, pat, x100, x_45
- Números – 100, 1235, 2000,45
- Variables – X, Y, Xval, _X
- Estructuras: día (9 de junio de 2017), punto (10, 25)

Átomos y variables

En esta sección, discutiremos los átomos, los números y las variables de Prolog.

átomos



Los átomos son una variación de las constantes. Pueden ser cualquier nombre u objeto. Hay algunas reglas que se deben seguir cuando intentamos utilizar Atoms, como se indica a continuación:

Cadenas de letras, dígitos y el carácter de subrayado, '_', comenzando con una letra minúscula. Por ejemplo -

- azahar
- b59
- b_59
- b_59AB
- b_x25
- antara_sarkar

Cadenas de caracteres especiales

Debemos tener en cuenta que cuando utilizamos átomos de esta forma, es necesario tener cierto cuidado ya que algunas cadenas de caracteres especiales ya tienen un significado predefinido; Por ejemplo ':-'.

- <--->
- =====>
- ...
- .∴.
- ::=

Cadenas de caracteres entre comillas simples.

Esto es útil si queremos tener un átomo que comience con mayúscula. Al encerrarlo entre comillas, lo distinguimos de las variables:

- 'Rubai'
- 'Arindam_Chatterjee'
- 'Sumit Mitra'

Números

Otra variación de las constantes son los Números. Entonces, los números enteros se pueden representar como 100, 4, -81, 1202. En Prolog, el rango normal de números enteros es de -16383 a 16383.

Prolog también admite números reales, pero normalmente el caso de uso de números de punto flotante es mucho menor en los programas Prolog, porque Prolog es para cálculos simbólicos y no numéricos. El tratamiento de los números reales depende de la implementación de Prolog. Ejemplos de números reales son 3,14159, -0,00062, 450,18, etc.

Las variables se encuentran en la sección **Objetos simples** . Las variables se pueden utilizar en muchos de estos casos en nuestro programa Prolog, que hemos visto anteriormente. Entonces existen algunas reglas para definir variables en Prolog.

Podemos definir variables de Prolog, de modo que las variables sean cadenas de letras, dígitos y caracteres de subrayado. Comienzan **con** una letra **mayúscula** o un **carácter de subrayado** . Algunos ejemplos de variables son:

- X
- Suma
- nombre_memer
- lista_estudiantes
- Lista de la compra
- _a50
- _15

Variables anónimas en Prolog

Las variables anónimas no tienen nombre. Las variables anónimas en el prólogo se escriben con un único carácter de subrayado '_'. Y una cosa importante es que cada variable anónima individual se trata como **diferente** . No son iguales.

Ahora la pregunta es, ¿dónde deberíamos utilizar estas variables anónimas?

Supongamos que en nuestra base de conocimientos tenemos algunos datos: "jim odia a tom", "pat odia a bob". Entonces, si Tom quiere saber quién lo odia, puede usar variables. Sin embargo, si quiere comprobar si hay alguien que lo odia, podemos utilizar variables anónimas. Entonces, cuando queremos usar la variable, pero no queremos revelar el valor de la variable, podemos usar variables anónimas.

Entonces, veamos su implementación práctica:

Base de conocimientos (var_anonymous.pl)

```
hates(jim,tom).  
hates(pat,bob).  
hates(dog,fox).  
hates(peter,tom).
```

Producción

```
| ?- [var_anonymous].  
compiling D:/TP Prolog/Sample_Codes/var_anonymous.pl for byte code...  
D:/TP Prolog/Sample_Codes/var_anonymous.pl compiled, 3 lines read - 536 bytes written  
  
yes  
| ?- hates(X,tom).  
  
X = jim ? ;  
  
X = peter  
  
yes  
| ?- hates(_,tom).  
  
true ? ;  
  
(16 ms) yes  
| ?- hates(_,pat).  
  
no  
| ?- hates(_,fox).  
  
true ? ;  
  
no  
| ?-
```

Prólogo - Operadores

En las siguientes secciones veremos cuáles son los diferentes tipos de operadores en Prolog. Tipos de operadores de comparación y operadores aritméticos.

También veremos en qué se diferencian de cualquier otro operador de lenguaje de alto nivel, en qué se diferencian sintácticamente y en qué se diferencian en su trabajo. También veremos alguna demostración práctica para entender el uso de diferentes operadores.

Operadores de comparación

Los operadores de comparación se utilizan para comparar dos ecuaciones o estados. A continuación se muestran diferentes operadores de comparación:

Operador	Significado
$X > Y$	X es mayor que Y
$X < Y$	X es menor que Y
$X \geq Y$	X es mayor o igual que Y
$X \leq Y$	X es menor o igual que Y
$X =: Y$	los valores de X e Y son iguales
$X \neq Y$	los valores de X e Y no son iguales

Puede ver que el operador ' \leq ', el operador ' $=:$ ' y los operadores ' \neq ' son sintácticamente diferentes de otros lenguajes. Veamos alguna demostración práctica de esto.

Ejemplo

```
| ?- 1+2=:2+1.
```

yes

```
| ?- 1+2=2+1.
```



no

| ?- 1+A=B+2.

A = 2

B = 1

yes

| ?- 5<10.

yes

| ?- 5>10.

no

| ?- 10=\=100.

yes

Aquí podemos ver que $1+2=:=2+1$ devuelve verdadero, pero $1+2=2+1$ devuelve falso. Esto se debe a que, en el primer caso se verifica si el valor de $1 + 2$ es igual que $2 + 1$ o no, y en el otro se verifica si dos patrones ' $1+2$ ' y ' $2+1$ ' son iguales o no. Como no son iguales, devuelve no (falso). En el caso de $1+A=B+2$, A y B son dos variables y se les asignan automáticamente algunos valores que coincidirán con el patrón.

Operadores aritméticos en Prolog

Los operadores aritméticos se utilizan para realizar operaciones aritméticas. Existen algunos tipos diferentes de operadores aritméticos de la siguiente manera:

Operador	Significado
+	Suma
-	Sustracción
*	Multiplicación
/	División
**	Fuerza
//	División entera

modificación

Módulo

Veamos un código práctico para comprender el uso de estos operadores.

Programa

```
calc :- X is 100 + 200,write('100 + 200 is '),write(X),nl,  
        Y is 400 - 150,write('400 - 150 is '),write(Y),nl,  
        Z is 10 * 300,write('10 * 300 is '),write(Z),nl,  
        A is 100 / 30,write('100 / 30 is '),write(A),nl,  
        B is 100 // 30,write('100 // 30 is '),write(B),nl,  
        C is 100 ** 2,write('100 ** 2 is '),write(C),nl,  
        D is 100 mod 30,write('100 mod 30 is '),write(D),nl.
```

Nota : nl se utiliza para crear una nueva línea.

Producción

```
| ?- change_directory('D:/TP Prolog/Sample_Codes').  
  
yes  
| ?- [op_arith].  
compiling D:/TP Prolog/Sample_Codes/op_arith.pl for byte code...  
D:/TP Prolog/Sample_Codes/op_arith.pl compiled, 6 lines read - 2390 bytes written, 11 r  
  
yes  
| ?- calc.  
100 + 200 is 300  
400 - 150 is 250  
10 * 300 is 3000  
100 / 30 is 3.3333333333333335  
100 // 30 is 3  
100 ** 2 is 10000.0  
100 mod 30 is 10  
  
yes  
| ?-
```

Prólogo: bucle y toma de decisiones

En este capítulo, discutiremos los bucles y la toma de decisiones en Prolog.

Bucles

Las declaraciones de bucle se utilizan para ejecutar el bloque de código varias veces. En general, for, while y do- while son construcciones de bucle en lenguajes de programación (como Java, C, C++).

El bloque de código se ejecuta varias veces utilizando lógica de predicados recursiva. No hay bucles directos en algunos otros lenguajes, pero podemos simular bucles con algunas técnicas diferentes.

Programa

```
count_to_10(10) :- write(10),nl.  
count_to_10(X) :-  
    write(X),nl,  
    Y is X + 1,  
    count_to_10(Y).
```

Producción

```
| ?- [loop].  
compiling D:/TP Prolog/Sample_Codes/loop.pl for byte code...  
D:/TP Prolog/Sample_Codes/loop.pl compiled, 4 lines read - 751 bytes written, 16 ms  
  
(16 ms) yes  
| ?- count_to_10(3).  
3  
4  
5  
6  
7  
8  
9  
10
```



```
true ?  
yes  
| ?-
```

Ahora cree un bucle que tome los valores más bajo y más alto. Entonces, podemos usar `Between()` para simular bucles.

Programa

Veamos un programa de ejemplo:

```
count_down(L, H) :-  
    between(L, H, Y),  
    Z is H - Y,  
    write(Z), nl.  
  
count_up(L, H) :-  
    between(L, H, Y),  
    Z is L + Y,  
    write(Z), nl.
```

Producción

```
| ?- [loop].  
compiling D:/TP Prolog/Sample_Codes/loop.pl for byte code...  
D:/TP Prolog/Sample_Codes/loop.pl compiled, 14 lines read - 1700 bytes written, 16 ms  
  
yes  
| ?- count_down(12,17).  
5  
  
true ? ;  
4  
  
true ? ;  
3  
  
true ? ;  
2
```

```
true ? ;  
1  
  
true ? ;  
0  
  
yes  
| ?- count_up(5,12).  
10  
  
true ? ;  
11  
  
true ? ;  
12  
  
true ? ;  
13  
  
true ? ;  
14  
  
true ? ;  
15  
  
true ? ;  
16  
  
true ? ;  
17  
  
yes  
| ?-
```

Toma de decisiones

Las declaraciones de decisión son declaraciones If-Then-Else. Entonces, cuando intentamos cumplir alguna condición y realizar alguna tarea, utilizamos las declaraciones de toma de decisiones. El uso básico es el siguiente:

If <condition> is true, Then <do this>, Else

En algunos lenguajes de programación diferentes, existen declaraciones If-Else, pero en Prolog tenemos que definir nuestras declaraciones de alguna otra manera. A continuación se muestra un ejemplo de toma de decisiones en Prolog.

Programa

```
% If-Then-Else statement

gt(X,Y) :- X >= Y,write('X is greater or equal').
gt(X,Y) :- X < Y,write('X is smaller').

% If-Elif-Else statement

gte(X,Y) :- X > Y,write('X is greater').
gte(X,Y) :- X == Y,write('X and Y are same').
gte(X,Y) :- X < Y,write('X is smaller').
```

Producción

```
| ?- [test].
compiling D:/TP Prolog/Sample_Codes/test.pl for byte code...
D:/TP Prolog/Sample_Codes/test.pl compiled, 3 lines read - 529 bytes written, 15 ms

yes
| ?- gt(10,100).
X is smaller

yes
| ?- gt(150,100).
X is greater or equal

true ?

yes
| ?- gte(10,20).
X is smaller

(15 ms) yes
| ?- gte(100,20).
X is greater
```


true ?

yes

| ?- gte(100,100).

X and Y are same

true ?

yes

| ?-

Prólogo - Conjunciones y disyunciones

En este capítulo, discutiremos las propiedades de conjunción y disyunción. Estas propiedades se utilizan en otros lenguajes de programación que utilizan lógicas AND y OR. Prolog también usa la misma lógica en su sintaxis.

Conjunción

La conjunción (lógica Y) se puede implementar utilizando el operador coma (,). Entonces, dos predicados separados por coma se unen con la declaración AND. Supongamos que tenemos un predicado, **parent(jhon, bob)**, que significa "Jhon es padre de Bob", y otro predicado, **male(jhon)**, que significa "Jhon es hombre". Entonces podemos hacer otro predicado que sea **padre(jhon,bob)**, que significa "Jhon es padre de Bob". Podemos definir el predicado **padre** cuando es padre **Y** es hombre.

Disyunción

La disyunción (lógica OR) se puede implementar utilizando el operador de punto y coma (;). Entonces, dos predicados separados por punto y coma se unen con una declaración OR. Supongamos que tenemos un predicado, **padre(jhon, bob)**. Este dice que "Jhon es padre de Bob", y otro predicado, **madre(lili,bob)**, este dice que "lili es madre de bob". Si creamos otro predicado como **child()**, esto será cierto cuando **padre(jhon, bob)** sea verdadero **O** **madre(lili,bob)** sea verdadero.

Programa

```
parent(jhon,bob).
parent(lili,bob).

male(jhon).
female(lili).

% Conjunction Logic
father(X,Y) :- parent(X,Y),male(X).
mother(X,Y) :- parent(X,Y),female(X).
```



% Disjunction Logic

```
child_of(X,Y) :- father(X,Y);mother(X,Y).
```

Producción

```
| ?- [conj_disj].
```

```
compiling D:/TP Prolog/Sample_Codes/conj_disj.pl for byte code...
```

```
D:/TP Prolog/Sample_Codes/conj_disj.pl compiled, 11 lines read - 1513 bytes written, 24
```

```
yes
```

```
| ?- father(jhon,bob).
```

```
yes
```

```
| ?- child_of(jhon,bob).
```

```
true ?
```

```
yes
```

```
| ?- child_of(lili,bob).
```

```
yes
```

```
| ?-
```

Prólogo - Listas

En este capítulo, analizaremos uno de los conceptos importantes de Prolog, las listas. Es una estructura de datos que se puede utilizar en diferentes casos para programación no numérica. Las listas se utilizan para almacenar los átomos como una colección.

En las secciones siguientes, discutiremos los siguientes temas:

- Representación de listas en Prolog
- Operaciones básicas en el prólogo como insertar, eliminar, actualizar, agregar.
- Operadores de reposicionamiento como permutación, combinación, etc.
- Establecer operaciones como establecer unión, establecer intersección, etc.

Representación de listas

La lista es una estructura de datos simple que se usa ampliamente en programación no numérica. La lista consta de cualquier número de elementos, por ejemplo, rojo, verde, azul, blanco y oscuro. Se representará como [rojo, verde, azul, blanco, oscuro]. La lista de elementos irá entre **corchetes** .

Una lista puede estar **vacía** o **no vacía** . En el primer caso, la lista se escribe simplemente como un átomo de Prolog, []. En el segundo caso, la lista consta de dos cosas como se detalla a continuación:

- El primer elemento, denominado **cabeza** de lista;
- La parte restante de la lista, llamada **cola** .

Supongamos que tenemos una lista como: [rojo, verde, azul, blanco, oscuro]. Aquí la cabeza es roja y la cola [verde, azul, blanca, oscura]. Entonces la cola es otra lista.

Ahora, consideremos que tenemos una lista, $L = [a, b, c]$. Si escribimos $\text{Tail} = [b, c]$ entonces también podemos escribir la lista L como $L = [a \mid \text{Cola}]$. Aquí la barra vertical (|) separa las partes de la cabeza y la cola.

Entonces las siguientes representaciones de lista también son válidas:



- $[a, b, c] = [x \mid [\text{antes de Cristo}]]$
- $[a, b, c] = [a, b \mid [C]]$
- $[a, b, c] = [a, b, c \mid []]$

Para estas propiedades podemos definir la lista como:

Una estructura de datos que está vacía o consta de dos partes: una cabeza y una cola. La cola en sí tiene que ser una lista.

Operaciones básicas en listas

La siguiente tabla contiene varias operaciones en listas de prólogo:

Operaciones	Definición
Comprobación de membresía	Durante esta operación, podemos verificar si un elemento determinado es miembro de una lista especificada o no.
Cálculo de longitud	Con esta operación podemos encontrar la longitud de una lista.
Concatenación	La concatenación es una operación que se utiliza para unir/agregar dos listas.
Eliminar objetos	Esta operación elimina el elemento especificado de una lista.
Agregar elementos	La operación de agregar agrega una lista a otra (como un elemento).
Insertar elementos	Esta operación inserta un elemento determinado en una lista.

Operación de membresía

Durante esta operación, podemos verificar si un miembro X está presente en la lista L o no. Entonces, ¿cómo comprobar esto? Bueno, tenemos que definir un predicado para hacerlo. Supongamos que el nombre del predicado es **list_member(X,L)**. El objetivo de este predicado es comprobar si X está presente en L o no.

Para diseñar este predicado, podemos seguir estas observaciones. X es miembro de L si:

- X es cabeza de L, o
- X es miembro de la cola de L.

Programa

```
list_member(X,[X|_]).  
list_member(X,[_|TAIL]) :- list_member(X,TAIL).
```

Producción

```
| ?- [list_basics].  
compiling D:/TP Prolog/Sample_Codes/list_basics.pl for byte code...  
D:/TP Prolog/Sample_Codes/list_basics.pl compiled, 1 lines read - 467 bytes written, 13  
  
yes  
| ?- list_member(b,[a,b,c]).  
  
true ?  
  
yes  
| ?- list_member(b,[a,[b,c]]).  
  
no  
| ?- list_member([b,c],[a,[b,c]]).  
  
true ?  
  
yes  
| ?- list_member(d,[a,b,c]).  
  
no  
| ?- list_member(d,[a,b,c]).
```

Cálculo de longitud

Esto se utiliza para encontrar la longitud de la lista L. Definiremos un predicado para realizar esta tarea. Supongamos que el nombre del predicado es **list_length(L,N)** . Esto toma L y N como argumento de entrada. Esto contará los elementos en una lista L y creará una instancia de N a su número. Como fue el caso con nuestras relaciones anteriores que involucraban listas, es útil considerar dos casos:

- Si la lista está vacía, entonces la longitud es 0.
- Si la lista no está vacía, entonces $L = [\text{Cabeza}|\text{Cola}]$, entonces su longitud es $1 + \text{longitud de la cola}$.

Programa

```
list_length([],0).  
list_length([_|TAIL],N) :- list_length(TAIL,N1), N is N1 + 1.
```

Producción

```
| ?- [list_basics].  
compiling D:/TP Prolog/Sample_Codes/list_basics.pl for byte code...  
D:/TP Prolog/Sample_Codes/list_basics.pl compiled, 4 lines read - 985 bytes written, 23  
  
yes  
| ?- list_length([a,b,c,d,e,f,g,h,i,j],Len).  
  
Len = 10  
  
yes  
| ?- list_length([],Len).  
  
Len = 0  
  
yes  
| ?- list_length([[a,b],[c,d],[e,f]],Len).  
  
Len = 3  
  
yes  
| ?-
```

Concatenación

La concatenación de dos listas significa agregar los elementos de la segunda lista después de la primera. Entonces, si dos listas son $[a,b,c]$ y $[1,2]$, entonces la lista final será $[a,b,c,1,2]$. Entonces, para realizar esta tarea crearemos un predicado llamado `list_concat()`, que tomará la primera lista $L1$, la segunda lista $L2$ y la $L3$ como lista resultante. Hay dos observaciones aquí.

- Si la primera lista está vacía y la segunda lista es L , entonces la lista resultante será L .
- Si la primera lista no está vacía, escriba esto como $[Cabeza|Cola]$, concatene $Cola$ con $L2$ de forma recursiva y guárdela en una nueva lista con el formato $[Cabeza|Nueva\ lista]$.

Programa

```
list_concat([],L,L).
list_concat([X1|L1],L2,[X1|L3]) :- list_concat(L1,L2,L3).
```

Producción

```
| ?- [list_basics].
compiling D:/TP Prolog/Sample_Codes/list_basics.pl for byte code...
D:/TP Prolog/Sample_Codes/list_basics.pl compiled, 7 lines read - 1367 bytes written, 19
yes
| ?- list_concat([1,2],[a,b,c],NewList).

NewList = [1,2,a,b,c]

yes
| ?- list_concat([], [a,b,c], NewList).

NewList = [a,b,c]

yes
| ?- list_concat([[1,2,3],[p,q,r]], [a,b,c], NewList).

NewList = [[1,2,3],[p,q,r],a,b,c]
```


yes

| ?-

Eliminar de la lista

Supongamos que tenemos una lista L y un elemento X, tenemos que eliminar X de L. Entonces hay tres casos:

- Si X es el único elemento, después de eliminarlo, devolverá una lista vacía.
- Si X es la cabeza de L, la lista resultante será la parte de la cola.
- Si X está presente en la parte Tail, elimínalo desde allí de forma recursiva.

Programa

```
list_delete(X, [X], []).
list_delete(X, [X|L1], L1).
list_delete(X, [Y|L2], [Y|L1]) :- list_delete(X, L2, L1).
```

Producción

```
| ?- [list_basics].
compiling D:/TP Prolog/Sample_Codes/list_basics.pl for byte code...
D:/TP Prolog/Sample_Codes/list_basics.pl compiled, 11 lines read - 1923 bytes written, 2

yes
| ?- list_delete(a,[a,e,i,o,u],NewList).

NewList = [e,i,o,u] ?

yes
| ?- list_delete(a,[a],NewList).

NewList = [] ?

yes
| ?- list_delete(X,[a,e,i,o,u],[a,e,o,u]).
```

```
X = i ? ;
```

```
no
```

```
| ?-
```

Agregar a la lista

Agregar dos listas significa agregar dos listas o agregar una lista como un elemento. Ahora bien, si el elemento está presente en la lista, la función de agregar no funcionará. Entonces crearemos un predicado, a saber, `list_append(L1, L2, L3)`. Las siguientes son algunas observaciones:

- Sea A un elemento, L1 es una lista, la salida también será L1, cuando L1 ya tenga A.
- De lo contrario, la nueva lista será $L2 = [A|L1]$.

Programa

```
list_member(X,[X|_]).
list_member(X,[_|TAIL]) :- list_member(X,TAIL).

list_append(A,T,T) :- list_member(A,T),!.
list_append(A,T,[A|T]).
```

En este caso hemos utilizado el símbolo (!), que se conoce como corte. Entonces, cuando la primera línea se ejecuta con éxito, la cortamos para que no ejecute la siguiente operación.

Producción

```
| ?- [list_basics].
compiling D:/TP Prolog/Sample_Codes/list_basics.pl for byte code...
D:/TP Prolog/Sample_Codes/list_basics.pl compiled, 14 lines read - 2334 bytes written, 2

(16 ms) yes
| ?- list_append(a,[e,i,o,u],NewList).

NewList = [a,e,i,o,u]
```

```
yes
| ?- list_append(e,[e,i,o,u],NewList).
```

```
NewList = [e,i,o,u]
```

```
yes
| ?- list_append([a,b],[e,i,o,u],NewList).
```

```
NewList = [[a,b],e,i,o,u]
```

```
yes
| ?-
```

Insertar en la lista

Este método se utiliza para insertar un elemento X en la lista L, y la lista resultante será R. Por lo tanto, el predicado tendrá esta forma `list_insert(X, L, R)`. Entonces esto puede insertar X en L en todas las posiciones posibles. Si miramos más de cerca, entonces hay algunas observaciones.

- Si realizamos `list_insert(X,L,R)`, podemos usar `list_delete(X,R,L)`, así que elimine X de R y cree una nueva lista L.

Programa

```
list_delete(X, [X], []).
list_delete(X,[X|L1], L1).
list_delete(X, [Y|L2], [Y|L1]) :- list_delete(X,L2,L1).

list_insert(X,L,R) :- list_delete(X,R,L).
```

Producción

```
| ?- [list_basics].
compiling D:/TP Prolog/Sample_Codes/list_basics.pl for byte code...
D:/TP Prolog/Sample_Codes/list_basics.pl compiled, 16 lines read - 2558 bytes written, 2

(16 ms) yes
| ?- list_insert(a,[e,i,o,u],NewList).
```

NewList = [a,e,i,o,u] ? a

NewList = [e,a,i,o,u]

NewList = [e,i,a,o,u]

NewList = [e,i,o,a,u]

NewList = [e,i,o,u,a]

NewList = [e,i,o,u,a]

(15 ms) no

| ?-

Operaciones de reposicionamiento de elementos de la lista.

Las siguientes son operaciones de reposicionamiento:

Operaciones de reposicionamiento	Definición
Permutación	Esta operación cambiará las posiciones de los elementos de la lista y generará todos los resultados posibles.
Artículos inversos	Esta operación organiza los elementos de una lista en orden inverso.
Elementos de turno	Esta operación desplazará un elemento de una lista hacia la izquierda de forma rotatoria.
Encargar artículos	Esta operación verifica si la lista dada está ordenada o no.

Operación de permutación

Esta operación cambiará las posiciones de los elementos de la lista y generará todos los resultados posibles. Entonces crearemos un predicado como `list_perm(L1,L2)`.

Esto generará todas las permutaciones de L1 y las almacenará en L2. Para hacer esto necesitamos la cláusula `list_delete()` para ayudar.

Para diseñar este predicado, podemos seguir algunas observaciones como se detalla a continuación:

X es miembro de L si:

- Si la primera lista está vacía, la segunda lista también debe estar vacía.
- Si la primera lista no está vacía, entonces tiene la forma `[X | L]`, y se puede construir una permutación de dicha lista como, primero permutar L obteniendo L1 y luego insertar X en cualquier posición en L1.

Programa

```
list_delete(X,[X|L1], L1).
list_delete(X, [Y|L2], [Y|L1]) :- list_delete(X,L2,L1).

list_perm([],[]).
list_perm(L,[X|P]) :- list_delete(X,L,L1),list_perm(L1,P).
```

Producción

```
| ?- [list_repos].
compiling D:/TP Prolog/Sample_Codes/list_repos.pl for byte code...
D:/TP Prolog/Sample_Codes/list_repos.pl compiled, 4 lines read - 1060 bytes written, 17
(15 ms) yes
| ?- list_perm([a,b,c,d],X).

X = [a,b,c,d] ? a

X = [a,b,d,c]

X = [a,c,b,d]

X = [a,c,d,b]

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

X = [a,d,c,b]

X = [b,a,c,d]

X = [b,a,d,c]

X = [b,c,a,d]

X = [b,c,d,a]

X = [b,d,a,c]

X = [b,d,c,a]

X = [c,a,b,d]

X = [c,a,d,b]

X = [c,b,a,d]

X = [c,b,d,a]

X = [c,d,a,b]

X = [c,d,b,a]

X = [d,a,b,c]

X = [d,a,c,b]

X = [d,b,a,c]

X = [d,b,c,a]

X = [d,c,a,b]

X = [d,c,b,a]

(31 ms) no

| ?-

Operación inversa

Supongamos que tenemos una lista $L = [a,b,c,d,e]$ y queremos invertir los elementos, por lo que la salida será $[e,d,c,b,a]$. Para hacer esto, crearemos una cláusula, `list_reverse(List, ReversedList)`. A continuación se presentan algunas observaciones:

- Si la lista está vacía, la lista resultante también estará vacía.
- De lo contrario, coloque los elementos de la lista, a saber, `[Cabeza|Cola]`, invierta los elementos de la cola de forma recursiva y concatene con el encabezado.
- De lo contrario, coloque los elementos de la lista, a saber, `[Cabeza|Cola]`, invierta los elementos de la cola de forma recursiva y concatene con el encabezado.

Programa

```
list_concat([],L,L).
list_concat([X1|L1],L2,[X1|L3]) :- list_concat(L1,L2,L3).

list_rev([],[]).
list_rev([Head|Tail],Reversed) :-
    list_rev(Tail, RevTail),list_concat(RevTail, [Head],Reversed).
```

Producción

```
| ?- [list_repos].
compiling D:/TP Prolog/Sample_Codes/list_repos.pl for byte code...
D:/TP Prolog/Sample_Codes/list_repos.pl compiled, 10 lines read - 1977 bytes written, 1

yes
| ?- list_rev([a,b,c,d,e],NewList).

NewList = [e,d,c,b,a]

yes
| ?- list_rev([a,b,c,d,e],[e,d,c,b,a]).
```



yes

| ?-

Operación de turno

Usando la operación Shift, podemos desplazar un elemento de una lista hacia la izquierda de forma rotatoria. Entonces, si los elementos de la lista son [a,b,c,d], luego de cambiar, serán [b,c,d,a]. Entonces crearemos una cláusula list_shift(L1, L2).

- Expresaremos la lista como [Cabeza|Cola], luego concatenaremos recursivamente Cabeza después de Cola, de modo que como resultado podamos sentir que los elementos están desplazados.
- Esto también se puede utilizar para comprobar si las dos listas están desplazadas en una posición o no.

Programa

```
list_concat([],L,L).
list_concat([X1|L1],L2,[X1|L3]) :- list_concat(L1,L2,L3).

list_shift([Head|Tail],Shifted) :- list_concat(Tail, [Head],Shifted).
```

Producción

```
| ?- [list_repos].
compiling D:/TP Prolog/Sample_Codes/list_repos.pl for byte code...
D:/TP Prolog/Sample_Codes/list_repos.pl compiled, 12 lines read - 2287 bytes written, 1

yes
| ?- list_shift([a,b,c,d,e],L2).

L2 = [b,c,d,e,a]

(16 ms) yes
| ?- list_shift([a,b,c,d,e],[b,c,d,e,a]).
```



yes

| ?-

Operación de pedido

Aquí definiremos un predicado `list_order(L)` que verifica si `L` está ordenado o no. Entonces, si `L = [1,2,3,4,5,6]`, entonces el resultado será verdadero.

- Si solo hay un elemento, ese ya está ordenado.
- De lo contrario, tome los dos primeros elementos `X` e `Y` como Cabeza y descansen como Cola. Si `X =< Y`, entonces llame a la cláusula nuevamente con el parámetro `[Y|Tail]`, de modo que esto verificará de forma recursiva desde el siguiente elemento.

Programa

```
list_order([X, Y | Tail]) :- X =< Y, list_order([Y|Tail]).
list_order([X]).
```

Producción

```
| ?- [list_repos].
```

```
compiling D:/TP Prolog/Sample_Codes/list_repos.pl for byte code...
```

```
D:/TP Prolog/Sample_Codes/list_repos.pl:15: warning: singleton variables [X] for list_order
```

```
D:/TP Prolog/Sample_Codes/list_repos.pl compiled, 15 lines read - 2805 bytes written, 1
```

yes

```
| ?- list_order([1,2,3,4,5,6,6,7,7,8]).
```

true ?

yes

```
| ?- list_order([1,4,2,3,6,5]).
```

no

| ?-

Establecer operaciones en listas

Intentaremos escribir una cláusula que obtenga todos los subconjuntos posibles de un conjunto dado. Entonces, si el conjunto es $[a,b]$, entonces el resultado será $[], [a], [b], [a,b]$. Para hacerlo, crearemos una cláusula, `list_subset(L, X)`. Tomará `L` y devolverá cada subconjunto a `X`. Entonces procederemos de la siguiente manera:

- Si la lista está vacía, el subconjunto también está vacío.
- Encuentre el subconjunto de forma recursiva reteniendo la cabeza, y
- Haz otra llamada recursiva donde eliminaremos Head.

Programa

```
list_subset([], []).
list_subset([Head|Tail],[Head|Subset]) :- list_subset(Tail,Subset).
list_subset([Head|Tail],Subset) :- list_subset(Tail,Subset).
```

Producción

```
| ?- [list_set].
compiling D:/TP Prolog/Sample_Codes/list_set.pl for byte code...
D:/TP Prolog/Sample_Codes/list_set.pl:3: warning: singleton variables [Head] for list_subset/2
D:/TP Prolog/Sample_Codes/list_set.pl compiled, 2 lines read - 653 bytes written, 7 ms

yes
| ?- list_subset([a,b],X).

X = [a,b] ? ;

X = [a] ? ;

X = [b] ? ;

X = []

(15 ms) yes
| ?- list_subset([x,y,z],X).
```

```
X = [x,y,z] ? a
```

```
X = [x,y]
```

```
X = [x,z]
```

```
X = [x]
```

```
X = [y,z]
```

```
X = [y]
```

```
X = [z]
```

```
X = []
```

```
yes
```

```
| ?-
```

Operación sindical

Definamos una cláusula llamada `list_union(L1,L2,L3)`. Entonces esto tomará L1 y L2, realizará una unión en ellos y almacenará el resultado en L3. Como sabes, si dos listas tienen el mismo elemento dos veces, después de la unión, solo habrá uno. Entonces necesitamos otra cláusula auxiliar para verificar la membresía.

Programa

```
list_member(X,[X|_]).
list_member(X,[_|TAIL]) :- list_member(X,TAIL).

list_union([X|Y],Z,W) :- list_member(X,Z),list_union(Y,Z,W).
list_union([X|Y],Z,[X|W]) :- \+ list_member(X,Z), list_union(Y,Z,W).
list_union([],Z,Z).
```

Nota : en el programa, hemos utilizado el operador (`\+`), este operador se usa para **NOT**.

Producción

```
| ?- [list_set].
```

```
compiling D:/TP Prolog/Sample_Codes/list_set.pl for byte code...
```

```
D:/TP Prolog/Sample_Codes/list_set.pl:6: warning: singleton variables [Head] for list_su
```

```
D:/TP Prolog/Sample_Codes/list_set.pl compiled, 9 lines read - 2004 bytes written, 18 m
```

```
yes
```

```
| ?- list_union([a,b,c,d,e],[a,e,i,o,u],L3).
```

```
L3 = [b,c,d,a,e,i,o,u] ?
```

```
(16 ms) yes
```

```
| ?- list_union([a,b,c,d,e],[1,2],L3).
```

```
L3 = [a,b,c,d,e,1,2]
```

```
yes
```

Operación de intersección

Definamos una cláusula llamada `list_intersection(L1,L2,L3)`, por lo que tomará `L1` y `L2`, realizará la operación de intersección y almacenará el resultado en `L3`.

Intersection devolverá aquellos elementos que están presentes en ambas listas.

Entonces `L1 = [a,b,c,d,e]`, `L2 = [a,e,i,o,u]`, luego `L3 = [a,e]`. Aquí, usaremos la cláusula `list_member()` para verificar si un elemento está presente en una lista o no.

Programa

```
list_member(X,[X|_]).
```

```
list_member(X,[_|TAIL]) :- list_member(X,TAIL).
```

```
list_intersect([X|Y],Z,[X|W]) :-
```

```
    list_member(X,Z), list_intersect(Y,Z,W).
```

```
list_intersect([X|Y],Z,W) :-
```

```
    \+ list_member(X,Z), list_intersect(Y,Z,W).
```

```
list_intersect([],Z,[]).
```

Producción

```
| ?- [list_set].
compiling D:/TP Prolog/Sample_Codes/list_set.pl for byte code...
D:/TP Prolog/Sample_Codes/list_set.pl compiled, 13 lines read - 3054 bytes written, 9 m
(15 ms) yes
| ?- list_intersect([a,b,c,d,e],[a,e,i,o,u],L3).

L3 = [a,e] ?

yes
| ?- list_intersect([a,b,c,d,e],[],L3).

L3 = []

yes
| ?-
```

Operaciones varias en listas

A continuación se muestran algunas operaciones diversas que se pueden realizar en listas:

Operaciones varias	Definición
Hallazgo de longitudes pares e impares	Verifica si la lista tiene un número par o impar de elementos.
Dividir	Divide una lista en dos listas, y estas listas tienen aproximadamente la misma longitud.
máx.	Recupera el elemento con valor máximo de la lista dada.
Suma	Devuelve la suma de elementos de la lista dada.
Combinar ordenar	Organiza los elementos de una lista determinada en orden (utilizando el algoritmo Merge Sort).

Operación de longitud par e impar

En este ejemplo, veremos dos operaciones mediante las cuales podemos verificar si la lista tiene un número impar de elementos o un número par de elementos. Definiremos predicados, a saber, `list_even_len(L)` y `list_odd_len(L)`.

- Si la lista no tiene elementos, entonces es una lista de longitud par.
- De lo contrario, lo tomamos como `[Cabeza | Cola]`, luego, si la cola tiene una longitud impar, entonces la lista total es una cadena de longitud par.
- De manera similar, si la lista tiene un solo elemento, entonces esa es una lista de longitud impar.
- Al tomarlo como `[Cabeza | Cola]` y Cola es una cadena de longitud par, entonces la lista completa es una lista de longitud impar.

Programa

```
list_even_len([]).
list_even_len([Head|Tail]) :- list_odd_len(Tail).

list_odd_len([]).
list_odd_len([Head|Tail]) :- list_even_len(Tail).
```

Producción

```
| ?- [list_misc].
compiling D:/TP Prolog/Sample_Codes/list_misc.pl for byte code...
D:/TP Prolog/Sample_Codes/list_misc.pl:2: warning: singleton variables [Head] for list_e
D:/TP Prolog/Sample_Codes/list_misc.pl:5: warning: singleton variables [Head] for list_o
D:/TP Prolog/Sample_Codes/list_misc.pl compiled, 4 lines read - 726 bytes written, 20 m

yes
| ?- list_odd_len([a,2,b,3,c]).

true ?

yes
| ?- list_odd_len([a,2,b,3]).

no
| ?- list_even_len([a,2,b,3]).
```

```
true ?
```

```
yes
```

```
| ?- list_even_len([a,2,b,3,c]).
```

```
no
```

```
| ?-
```

Operación de lista dividida

Esta operación divide una lista en dos listas, y estas listas tienen aproximadamente la misma longitud. Entonces, si la lista dada es [a,b,c,d,e], entonces el resultado será [a,c,e],[b,d]. Esto colocará todos los elementos colocados impares en una lista y todos los elementos colocados pares en otra lista. Definiremos un predicado, `list_divide(L1,L2,L3)` para resolver esta tarea.

- Si la lista dada está vacía, devolverá listas vacías.
- Si solo hay un elemento, entonces la primera lista será una lista con ese elemento y la segunda lista estará vacía.
- Supongamos que X, Y son dos elementos de la cabeza y el resto son la cola. Entonces haga dos listas [X | Lista1], [Y | Lista2], estas Lista1 y Lista2 se separan dividiendo la cola.

Programa

```
list_divide([],[],[]).
list_divide([X],[X],[]).
list_divide([X,Y|Tail], [X|List1],[Y|List2]) :-
    list_divide(Tail,List1,List2).
```

Producción

```
| ?- [list_misc].
```

```
compiling D:/TP Prolog/Sample_Codes/list_misc.pl for byte code...
```

```
D:/TP Prolog/Sample_Codes/list_misc.pl:2: warning: singleton variables [Head] for list_e
```

```
D:/TP Prolog/Sample_Codes/list_misc.pl:5: warning: singleton variables [Head] for list_o
```

```
D:/TP Prolog/Sample_Codes/list_misc.pl compiled, 8 lines read - 1432 bytes written, 8 m
```

```
yes
| ?- list_divide([a,1,b,2,c,3,d,5,e],L1,L2).
```

```
L1 = [a,b,c,d,e]
L2 = [1,2,3,5] ?
```

```
yes
| ?- list_divide([a,b,c,d],L1,L2).
```

```
L1 = [a,c]
L2 = [b,d]
```

```
yes
| ?-
```

Operación máxima de artículos

Esta operación se utiliza para encontrar el elemento máximo de una lista. Definiremos un predicado, **list_max_elem(List, Max)** , luego encontrará el elemento Max de la lista y regresará.

- Si solo hay un elemento, será el elemento máximo.
- Divida la lista como [X,Y|Cola]. Ahora encuentre recursivamente el máximo de [Y|Tail] y guárdelo en MaxRest, y almacene el máximo de X y MaxRest, luego guárdelo en Max.

Programa

```
max_of_two(X,Y,X) :- X >= Y.
max_of_two(X,Y,Y) :- X < Y.
list_max_elem([X],X).
list_max_elem([X,Y|Rest],Max) :-
    list_max_elem([Y|Rest],MaxRest),
    max_of_two(X,MaxRest,Max).
```

Producción


```
| ?- [list_misc].
compiling D:/TP Prolog/Sample_Codes/list_misc.pl for byte code...
D:/TP Prolog/Sample_Codes/list_misc.pl:2: warning: singleton variables [Head] for list_e
D:/TP Prolog/Sample_Codes/list_misc.pl:5: warning: singleton variables [Head] for list_o
D:/TP Prolog/Sample_Codes/list_misc.pl compiled, 16 lines read - 2385 bytes written, 16

yes
| ?- list_max_elem([8,5,3,4,7,9,6,1],Max).

Max = 9 ?

yes
| ?- list_max_elem([5,12,69,112,48,4],Max).

Max = 112 ?

yes
| ?-
```

Operación de suma de listas

En este ejemplo, definiremos una cláusula, `list_sum(List, Sum)`, esto devolverá la suma de los elementos de la lista.

- Si la lista está vacía, la suma será 0.
- Representa la lista como `[Cabeza|Cola]`, encuentra la suma de la cola de forma recursiva y guárdala en `SumTemp`, luego establece `Suma = Cabeza + SumTemp`.

Programa

```
list_sum([],0).
list_sum([Head|Tail], Sum) :-
    list_sum(Tail,SumTemp),
    Sum is Head + SumTemp.
```

Producción

yes

| ?- [list_misc].

compiling D:/TP Prolog/Sample_Codes/list_misc.pl for byte code...

D:/TP Prolog/Sample_Codes/list_misc.pl:2: warning: singleton variables [Head] for list_e

D:/TP Prolog/Sample_Codes/list_misc.pl:5: warning: singleton variables [Head] for list_o

D:/TP Prolog/Sample_Codes/list_misc.pl compiled, 21 lines read - 2897 bytes written, 21

(32 ms) yes

| ?- list_sum([5,12,69,112,48,4],Sum).

Sum = 250

yes

| ?- list_sum([8,5,3,4,7,9,6,1],Sum).

Sum = 43

yes

| ?-

Combinar ordenar en una lista

Si la lista es [4,5,3,7,8,1,2], entonces el resultado será [1,2,3,4,5,7,8]. Los pasos para realizar la ordenación por combinación se muestran a continuación:

- Tome la lista y divídala en dos sublistas. Esta división se realizará de forma recursiva.
- Fusiona cada división en orden.
- De esta forma se ordenará toda la lista.

Definiremos un predicado llamado mergesort(L, SL), tomará L y devolverá el resultado a SL.

Programa

```
mergesort([], []).    /* covers special case */
mergesort([A],[A]).
mergesort([A,B|R],S) :-
```

```

split([A,B|R],L1,L2),
mergesort(L1,S1),
mergesort(L2,S2),
merge(S1,S2,S).

split([],[],[]).
split([A],[A],[]).
split([A,B|R],[A|Ra],[B|Rb]) :-
    split(R,Ra,Rb).
merge(A,[],A).
merge([],B,B).
merge([A|Ra],[B|Rb],[A|M]) :-
    A <= B, merge(Ra,[B|Rb],M).
merge([A|Ra],[B|Rb],[B|M]) :-
    A > B, merge([A|Ra],Rb,M).

```

Producción

```

| ?- [merge_sort].
compiling D:/TP Prolog/Sample_Codes/merge_sort.pl for byte code...
D:/TP Prolog/Sample_Codes/merge_sort.pl compiled, 17 lines read - 3048 bytes written,

yes
| ?- mergesort([4,5,3,7,8,1,2],L).

L = [1,2,3,4,5,7,8] ?

yes
| ?- mergesort([8,5,3,4,7,9,6,1],L).

L = [1,3,4,5,6,7,8,9] ?

yes
| ?-

```

Prólogo - Recursión y Estructuras

Este capítulo cubre la recursividad y las estructuras.

recursividad

La recursividad es una técnica en la que un predicado se utiliza a sí mismo (puede ser con otros predicados) para encontrar el valor de verdad.

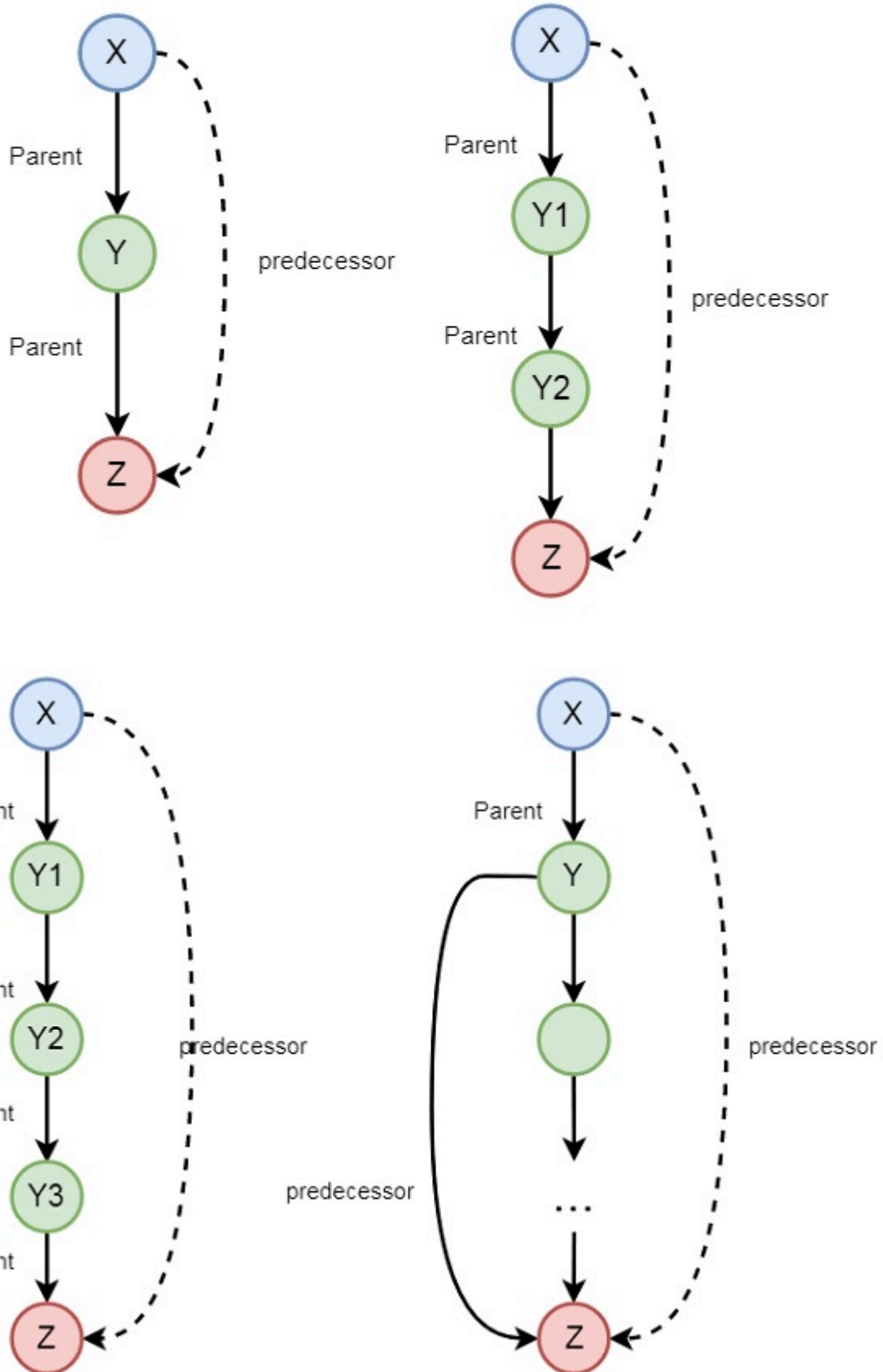
Entendamos esta definición con la ayuda de un ejemplo:

- `is_digesting(X,Y) :- just_ate(X,Y).`
- `está_digestando(X,Y) :- just_ate(X,Z), está_digestando(Z,Y).`

Entonces este predicado es de naturaleza recursiva. Supongamos que decimos que `just_ate(venado, hierba)`, significa `is_digesting(venado, hierba)` es verdadero. Ahora bien, si decimos `is_digesting(tigre, pasto)`, esto será cierto si `is_digesting(tigre, pasto) :- just_ate(tigre, venado), is_digesting(venado, pasto)`, entonces la afirmación `is_digesting(tigre, pasto)` también es verdadera .

Puede haber otros ejemplos también, así que veamos un ejemplo familiar. Entonces, si queremos expresar la lógica predecesora, podemos expresarla usando el siguiente diagrama:





Entonces podemos entender que la relación predecesora es recursiva. Podemos expresar esta relación usando la siguiente sintaxis:

- `predecesor (X, Z): - padre (X, Z).`

- predecesor (X, Z): - padre (X, Y), predecesor (Y, Z).

Estructuras

Las estructuras son objetos de datos que contienen múltiples componentes.

Por ejemplo, la fecha puede verse como una estructura con tres componentes: día, mes y año. Entonces la fecha 9 de abril de 2020 se puede escribir como: fecha (9 de abril de 2020).

Nota : la estructura, a su vez, puede tener otra estructura como componente.

Entonces podemos ver las vistas como estructura de árbol y **Prolog Functors** .



Ahora veamos un ejemplo de estructuras en Prolog. Definiremos una estructura de puntos, Segmentos y Triángulo como estructuras.

Para representar un punto, un segmento de línea y un triángulo usando la estructura en Prolog, podemos considerar las siguientes declaraciones:

- p1 – punto(1, 1)
- p2 – punto(2,3)
- S – seg(P1, P2): seg(punto(1,1), punto(2,3))
- T - triángulo (punto (4, Z), punto (6,4), punto (7,1))

Nota : las estructuras se pueden representar naturalmente como árboles. Prolog puede verse como un lenguaje para procesar árboles.

Coincidencia en el prólogo

La coincidencia se utiliza para comprobar si dos términos dados son iguales (idénticos) o si las variables en ambos términos pueden tener los mismos objetos después de crear una instancia. Veamos un ejemplo.

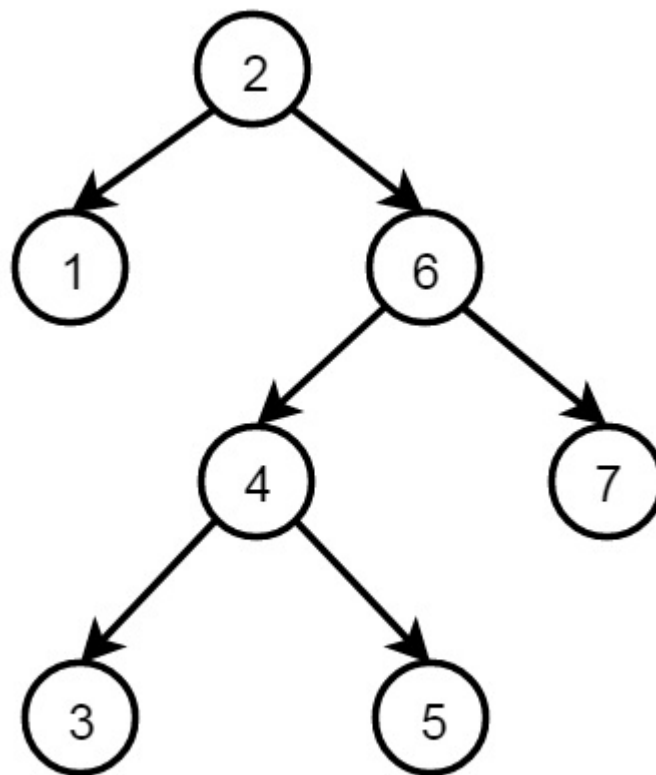
Supongamos que la estructura de fechas se define como $\text{fecha}(D, M, 2020) = \text{fecha}(D1, \text{abr}, Y1)$, esto indica que $D = D1$, $M = \text{feb}$ y $Y1 = 2020$.

Se utilizarán las siguientes reglas para comprobar si dos términos S y T coinciden:

- Si S y T son constantes, $S=T$ si ambos son objetos iguales.
- Si S es una variable y T es cualquier cosa, $T=S$.
- Si T es variable y S es cualquier cosa, $S=T$.
- Si S y T son estructuras, $S=T$ si –
 - S y T tienen el mismo functor.
 - Todos sus componentes de argumentos correspondientes deben coincidir.

árboles binarios

La siguiente es la estructura del árbol binario utilizando estructuras recursivas:



La definición de la estructura es la siguiente:

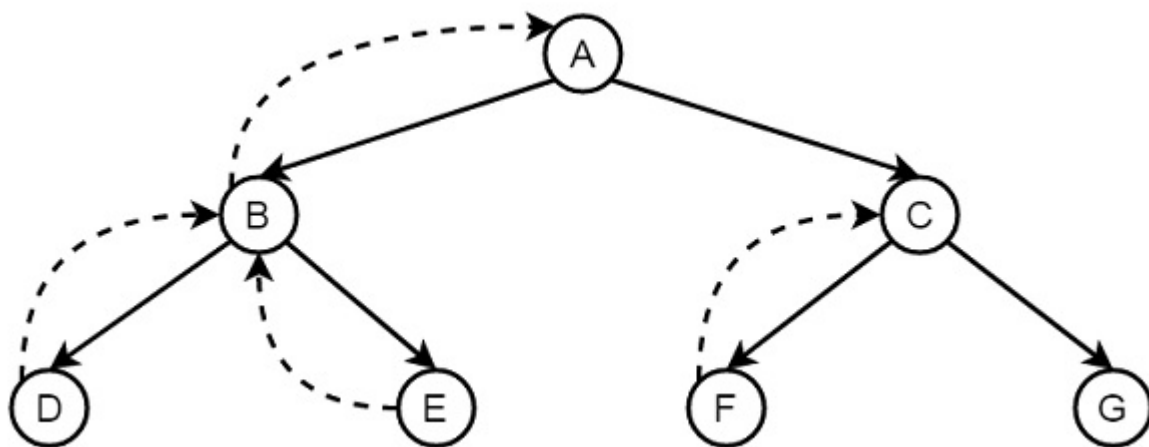
```
node(2, node(1,nil,nil), node(6, node(4,node(3,nil,nil), node(5,nil,nil)), nc
```

Cada nodo tiene tres campos, datos y dos nodos. Un nodo sin estructura secundaria (nodo hoja) se escribe como **nodo(valor, nil, nil)** , el nodo con un solo hijo izquierdo se escribe como **nodo(valor, left_node, nil)** , el nodo con solo un hijo derecho se escribe como **nodo (value, nil; right_node)** , y el nodo con ambos hijos tiene **node(value, left_node, right_node)** .

Prólogo - Retroceso

En este capítulo, discutiremos el retroceso en Prolog. El retroceso es un procedimiento en el que el prólogo busca el valor de verdad de diferentes predicados comprobando si son correctos o no. El término retroceso es bastante común en el diseño de algoritmos y en diferentes entornos de programación. En Prolog, hasta que llega al destino correcto, intenta retroceder. Cuando se encuentra el destino, **se detiene**.

Veamos cómo se realiza el retroceso utilizando una estructura similar a un árbol:



Supongamos que de A a G hay algunas reglas y hechos. Comenzamos desde A y queremos llegar a G. El camino correcto será ACG, pero al principio irá de A a B, luego de B a D. Cuando descubre que D no es el destino, retrocede hasta B, luego va a E y retrocede nuevamente a B, ya que no hay otro hijo de B, luego retrocede a A, busca G y finalmente encuentra G en el camino ACG. (Las líneas discontinuas indican el retroceso). Entonces, cuando encuentra G, se detiene.

¿Cómo funciona el retroceso?

Ahora sabemos qué es el retroceso en Prolog. Veamos un ejemplo,

Nota : mientras ejecutamos algún código de prólogo, durante el retroceso puede haber varias respuestas, podemos presionar **punto y coma (;)** para obtener las siguientes respuestas una por una, lo que ayuda a retroceder. De lo contrario, cuando obtengamos un resultado, se detendrá.

Ahora, considere una situación en la que dos personas, X e Y, pueden pagarse entre sí, pero la condición es que un niño pueda pagarle a una niña, por lo que X se



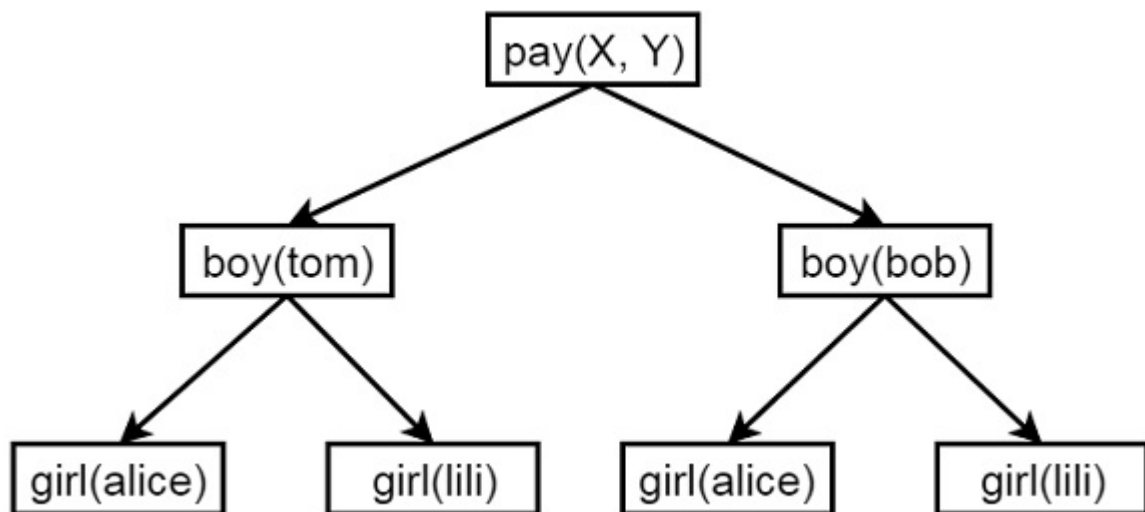
niño e Y será una niña. Entonces, para estos hemos definido algunos hechos y reglas:

Base de conocimientos

```
boy(tom).
boy(bob).
girl(alice).
girl(lili).

pay(X,Y) :- boy(X), girl(Y).
```

A continuación se muestra la ilustración del escenario anterior:



Como X será un niño, hay dos opciones, y para cada niño hay dos opciones, Alice y Lili. Ahora veamos el resultado, cómo funciona el retroceso.

Producción

```
| ?- [backtrack].
compiling D:/TP Prolog/Sample_Codes/backtrack.pl for byte code...
D:/TP Prolog/Sample_Codes/backtrack.pl compiled, 5 lines read - 703 bytes written, 22 r

yes
| ?- pay(X,Y).

X = tom
Y = alice ?
```

(15 ms) yes
| ?- pay(X,Y).

X = tom
Y = alice ? ;

X = tom
Y = lili ? ;

X = bob
Y = alice ? ;

X = bob
Y = lili

yes
| ?- trace.

The debugger will first creep -- showing everything (trace)

(16 ms) yes
{trace}
| ?- pay(X,Y).
1 1 Call: pay(_23,_24) ?
2 2 Call: boy(_23) ?
2 2 Exit: boy(tom) ?
3 2 Call: girl(_24) ?
3 2 Exit: girl(alice) ?
1 1 Exit: pay(tom,alice) ?

X = tom
Y = alice ? ;
1 1 Redo: pay(tom,alice) ?
3 2 Redo: girl(alice) ?
3 2 Exit: girl(lili) ?
1 1 Exit: pay(tom,lili) ?

X = tom
Y = lili ? ;
1 1 Redo: pay(tom,lili) ?
2 2 Redo: boy(tom) ?
2 2 Exit: boy(bob) ?



```

3 2 Call: girl(_24) ?
3 2 Exit: girl(alice) ?
1 1 Exit: pay(bob,alice) ?

```

X = bob

Y = alice ? ;

```

1 1 Redo: pay(bob,alice) ?

```

```

3 2 Redo: girl(alice) ?

```

```

3 2 Exit: girl(lili) ?

```

```

1 1 Exit: pay(bob,lili) ?

```

X = bob

Y = lili

yes

{trace}

| ?-

Prevenir el retroceso

Hasta ahora hemos visto algunos conceptos de retroceso. Veamos ahora algunos inconvenientes del retroceso. A veces escribimos los mismos predicados más de una vez cuando nuestro programa lo exige, por ejemplo para escribir reglas recursivas o para realizar algunos sistemas de toma de decisiones. En tales casos, el retroceso incontrolado puede causar ineficiencia en un programa. Para resolver esto, usaremos el **Corte** en Prolog.

Supongamos que tenemos algunas reglas de la siguiente manera:

Función de doble paso

- Regla 1 − si $X < 3$ entonces $Y = 0$
- Regla 2 − si $3 \leq X$ y $X < 6$ entonces $Y = 2$
- Regla 3 − si $6 \leq X$ entonces $Y = 4$

En la sintaxis de Prolog podemos escribir,

- `f(X,0) :- X < 3. % Regla 1`
- `f(X,2) :- 3 =< X, X < 6. % Regla 2`
- `f(X,4) :- 6 =< X. % Regla 3`

Ahora, si hacemos una pregunta como $f(1,Y), 2 < Y$.

El primer objetivo $f(1,Y)$ instancia Y a 0. El segundo objetivo se convierte en $2 < 0$, lo cual falla. Prolog intenta retroceder dos alternativas infructuosas (Regla 2 y Regla 3). Si miramos más de cerca, podemos observar que:

- Las tres reglas son mutuamente excluyentes y como máximo una de ellas tendrá éxito.
- Tan pronto como uno de ellos tiene éxito, no tiene sentido intentar utilizar los demás, ya que están destinados a fracasar.

Entonces podemos usar **cut** para resolver esto. El **corte** se puede expresar mediante el símbolo de exclamación. La sintaxis del prólogo es la siguiente:

- $f(X,0) :- X < 3, !. \%$ Regla 1
- $f(X,2) :- 3 \leq X, X < 6, !. \%$ Regla 2
- $f(X,4) :- 6 \leq X. \%$ Regla 3

Ahora, si usamos la misma pregunta, $?- f(1,Y), 2 < Y$. Prolog elige la regla 1 ya que $1 < 3$ y falla el objetivo $2 < Y$ falla. Prolog intentará retroceder, ipero no más allá del punto marcado! En el programa no se generarán la regla 2 y la regla 3.

Veamos esto en la siguiente ejecución:

Programa

```
f(X,0) :- X < 3.           % Rule 1
f(X,2) :- 3 <= X, X < 6.   % Rule 2
f(X,4) :- 6 <= X.         % Rule 3
```

Producción

```
| ?- [backtrack].
compiling D:/TP Prolog/Sample_Codes/backtrack.pl for byte code...
D:/TP Prolog/Sample_Codes/backtrack.pl compiled, 10 lines read - 1224 bytes written, 1
yes
| ?- f(1,Y), 2<Y.
```

no

| ?- trace

.

The debugger will first creep -- showing everything (trace)

yes

{trace}

| ?- f(1,Y), 2<Y.

1 1 Call: f(1,_23) ?

2 2 Call: 1<3 ?

2 2 Exit: 1<3 ?

1 1 Exit: f(1,0) ?

3 1 Call: 2<0 ?

3 1 Fail: 2<0 ?

1 1 Redo: f(1,0) ?

2 2 Call: 3=<1 ?

2 2 Fail: 3=<1 ?

2 2 Call: 6=<1 ?

2 2 Fail: 6=<1 ?

1 1 Fail: f(1,_23) ?

(46 ms) no

{trace}

| ?-

Veamos lo mismo usando cut.

Programa

```
f(X,0) :- X < 3,!.
```

```
% Rule 1
```

```
f(X,2) :- 3 =< X, X < 6,!.
```

```
% Rule 2
```

```
f(X,4) :- 6 =< X.
```

```
% Rule 3
```

Producción

| ?- [backtrack].

1 1 Call: [backtrack] ?

compiling D:/TP Prolog/Sample_Codes/backtrack.pl for byte code...

D:/TP Prolog/Sample_Codes/backtrack.pl compiled, 10 lines read - 1373 bytes written, 1

1 1 Exit: [backtrack] ?

```
(16 ms) yes
{trace}
| ?- f(1,Y), 2<Y.
  1 1 Call: f(1,_23) ?
  2 2 Call: 1<3 ?
  2 2 Exit: 1<3 ?
  1 1 Exit: f(1,0) ?
  3 1 Call: 2<0 ?
  3 1 Fail: 2<0 ?
no
{trace}
| ?-
```

La negación como fracaso

Aquí realizaremos una falla cuando la condición no se cumpla. Supongamos que tenemos una afirmación: "A María le gustan todos los animales menos las serpientes", la expresaremos en Prolog.

Sería muy fácil y directo si la afirmación fuera "A María le gustan todos los animales". En ese caso podemos escribir "A María le gusta X si X es un animal". Y en el prólogo podemos escribir esta declaración como me gusta (maría, X): = animal (X).

Nuestra declaración real se puede expresar como:

- Si X es una serpiente, entonces "A María le gusta X" no es cierto.
- De lo contrario, si X es un animal, entonces a María le gusta X.

En el prólogo podemos escribir esto como:

- Me gusta (maría, X): - serpiente (X), !, falla.
- le gusta (maría, X): - animal (X).

La declaración de 'fallo' causa el fracaso. Ahora veamos cómo funciona en Prolog.

Programa

```
animal(dog).
animal(cat).
animal(elephant).
```

```

animal(tiger).
animal(cobra).
animal(python).

snake(cobra).
snake(python).

likes(mary, X) :- snake(X), !, fail.
likes(mary, X) :- animal(X).

```

Producción

```

| ?- [negate_fail].
compiling D:/TP Prolog/Sample_Codes/negate_fail.pl for byte code...
D:/TP Prolog/Sample_Codes/negate_fail.pl compiled, 11 lines read - 1118 bytes written,

yes
| ?- likes(mary,elephant).

yes
| ?- likes(mary,tiger).

yes
| ?- likes(mary,python).

no
| ?- likes(mary,cobra).

no
| ?- trace
.
The debugger will first creep -- showing everything (trace)

yes
{trace}
| ?- likes(mary,dog).
  1 1 Call: likes(mary,dog) ?
  2 2 Call: snake(dog) ?
  2 2 Fail: snake(dog) ?
  2 2 Call: animal(dog) ?
  2 2 Exit: animal(dog) ?

```



```
1 1 Exit: likes(mary,dog) ?
```

```
yes
```

```
{trace}
```

```
| ?- likes(mary,python).
```

```
1 1 Call: likes(mary,python) ?
```

```
2 2 Call: snake(python) ?
```

```
2 2 Exit: snake(python) ?
```

```
3 2 Call: fail ?
```

```
3 2 Fail: fail ?
```

```
1 1 Fail: likes(mary,python) ?
```

```
no
```

```
{trace}
```

```
| ?-
```

Prólogo - Diferente y no

Aquí definiremos dos predicados: **diferente** y **no**. El predicado diferente comprobará si dos argumentos dados son iguales o no. Si son iguales, devolverá falso; de lo contrario, devolverá verdadero. El predicado **not** se usa para negar alguna afirmación, lo que significa que, cuando una afirmación es verdadera, entonces not(declaración) será falsa; de lo contrario, si la afirmación es falsa, entonces not(declaración) será verdadera.

Entonces, el término "diferente" se puede expresar de tres maneras diferentes, como se indica a continuación:

- X e Y no son literalmente lo mismo
- X e Y no coinciden
- Los valores de la expresión aritmética X e Y no son iguales.

Entonces, en Prolog, intentaremos expresar las declaraciones de la siguiente manera:

- Si X e Y coinciden, entonces falla diferente (X,Y),
- De lo contrario, diferente (X, Y) tiene éxito.

La sintaxis del prólogo respectivo será la siguiente:

- diferente(X, X) :- !, falla.
- diferente (X, Y).

También podemos expresarlo usando cláusulas disyuntivas como se indica a continuación:

- diferente (X, Y): - X = Y,! , falla; verdadero. % cierto es el objetivo que siempre se logra

Programa

El siguiente ejemplo muestra cómo se puede hacer esto en el prólogo:



```
different(X, X) :- !, fail.  
different(X, Y).
```

Producción

```
| ?- [diff_rel].  
compiling D:/TP Prolog/Sample_Codes/diff_rel.pl for byte code...  
D:/TP Prolog/Sample_Codes/diff_rel.pl:2: warning: singleton variables [X,Y] for different,  
D:/TP Prolog/Sample_Codes/diff_rel.pl compiled, 2 lines read - 327 bytes written, 11 ms  
  
yes  
| ?- different(100,200).  
  
yes  
| ?- different(100,100).  
  
no  
| ?- different(abc,def).  
  
yes  
| ?- different(abc,abc).  
  
no  
| ?-
```

Veamos un programa que usa las cláusulas disyuntivas:

Programa

```
different(X, Y) :- X = Y, !, fail ; true.
```

Producción

```
| ?- [diff_rel].  
compiling D:/TP Prolog/Sample_Codes/diff_rel.pl for byte code...  
D:/TP Prolog/Sample_Codes/diff_rel.pl compiled, 0 lines read - 556 bytes written, 17 ms  
  
yes
```

```
| ?- different(100,200).
```

```
yes
```

```
| ?- different(100,100).
```

```
no
```

```
| ?- different(abc,def).
```

```
yes
```

```
| ?- different(abc,abc).
```

```
no
```

```
| ?-
```

No relación en el prólogo

La relación **not** es muy útil en diferentes casos. También en nuestros lenguajes de programación tradicionales utilizamos la operación lógica not para negar alguna declaración. Entonces significa que cuando una afirmación es verdadera, entonces no (declaración) será falsa; de lo contrario, si la afirmación es falsa, entonces no (declaración) será verdadera.

En el prólogo, podemos definir esto como se muestra a continuación:

```
not(P) :- P, !, fail ; true.
```

Entonces, si P es verdadero, entonces corte y falle, esto devolverá falso; de lo contrario, será verdadero. Ahora veamos un código simple para entender este concepto.

Programa

```
not(P) :- P, !, fail ; true.
```

Producción

```
| ?- [not_rel].
```

```
compiling D:/TP Prolog/Sample_Codes/not_rel.pl for byte code...
```

```
D:/TP Prolog/Sample_Codes/not_rel.pl compiled, 0 lines read - 630 bytes written, 17 ms
```

```
yes  
| ?- not(true).
```

```
no  
| ?- not(fail).
```

```
yes  
| ?-
```

Prólogo - Entradas y Salidas

En este capítulo, veremos algunas técnicas para manejar entradas y salidas a través de prolog. Usaremos algunos predicados integrados para realizar estas tareas y también veremos técnicas de manejo de archivos.

Los siguientes temas se discutirán en detalle:

- Manejo de entradas y salidas.
- Manejo de archivos usando Prolog
- Usar algún archivo externo para leer líneas y términos
- Manipulación de caracteres para entrada y salida.
- Construyendo y descomponiendo átomos.
- Consultar archivos de prólogo en otras técnicas de programas de prólogo.

Manejo de entrada y salida

Hasta ahora hemos visto que podemos escribir un programa y la consulta en la consola para ejecutarlo. En algunos casos, imprimimos algo en la consola, que está escrito en nuestro código de prólogo. Así que aquí veremos las tareas de escritura y lectura con más detalle usando prolog. Estas serán las técnicas de manejo de entrada y salida.

El predicado write()

Para escribir la salida podemos usar el predicado **write()** . Este predicado toma el parámetro como entrada y escribe el contenido en la consola de forma predeterminada. write() también puede escribir en archivos. Veamos algunos ejemplos de la función write().

Programa

```
| ?- write(56).
```

```
56
```

```
yes
```



```
| ?- write('hello').  
hello  
  
yes  
| ?- write('hello'),nl,write('world').  
hello  
world  
  
yes  
| ?- write("ABCDE")  
.  
[65,66,67,68,69]  
  
yes
```

En el ejemplo anterior, podemos ver que el predicado `write()` puede escribir el contenido en la consola. Podemos usar `'\n'` para crear una nueva línea. Y de este ejemplo, queda claro que, si queremos imprimir alguna cadena en la consola, tenemos que usar comillas simples ('cadena'). Pero si usamos comillas dobles ("cadena"), devolverá una lista de valores ASCII.

El predicado `read()`

El predicado `read()` se utiliza para leer desde la consola. El usuario puede escribir algo en la consola, que puede tomarse como entrada y procesarlo. `read()` se usa generalmente para leer desde la consola, pero también se puede usar para leer desde archivos. Ahora veamos un ejemplo para ver cómo funciona `read()`.

Programa

```
cube :-  
    write('Write a number: '),  
    read(Number),  
    process(Number).  
process(stop) :- !.  
process(Number) :-  
    C is Number * Number * Number,  
    write('Cube of '),write(Number),write(': '),write(C),nl, cube.
```

Producción

```
| ?- [read_write].
compiling D:/TP Prolog/Sample_Codes/read_write.pl for byte code...
D:/TP Prolog/Sample_Codes/read_write.pl compiled, 9 lines read - 1226 bytes written, 1

(15 ms) yes
| ?- cube.
Write a number: 2.
Cube of 2: 8
Write a number: 10.
Cube of 10: 1000
Write a number: 12.
Cube of 12: 1728
Write a number: 8.
Cube of 8: 512
Write a number: stop
.

(31 ms) yes
| ?-
```

El predicado tab()

El tab() es un predicado adicional que se puede usar para poner algunos espacios en blanco mientras escribimos algo. Entonces toma un número como argumento e imprime esa cantidad de espacios en blanco.

Programa

```
| ?- write('hello'),tab(15),write('world').
hello          world

yes
| ?- write('We'),tab(5),write('will'),tab(5),write('use'),tab(5),write('tabs')
We    will    use    tabs

yes
| ?-
```


Lectura/escritura de archivos

En esta sección, veremos cómo podemos usar archivos para leer y escribir en los archivos. Hay algunos predicados integrados que se pueden usar para leer un archivo y escribir en él.

El decir y decir

Si queremos escribir en un archivo, excepto en la consola, podemos escribir el predicado **tell()**. Este predicado **tell()** toma el nombre del archivo como argumento. Si ese archivo no está presente, cree un archivo nuevo y escriba en él. Ese archivo se abrirá hasta que escribamos el comando **dicho**. Podemos abrir más de un archivo usando tell(). Cuando se llame, se cerrarán todos los archivos.

Comandos de prólogo

```
| ?- told('myFile.txt').
uncaught exception: error(existence_error(procedure,told/1),top_level/0)
| ?- told("myFile.txt").
uncaught exception: error(existence_error(procedure,told/1),top_level/0)
| ?- tell('myFile.txt').

yes
| ?- tell('myFile.txt').

yes
| ?- write('Hello World').

yes
| ?- write(' Writing into a file'),tab(5),write('myFile.txt'),nl.

yes
| ?- write("Write some ASCII values").

yes
| ?- told.

yes
| ?-
```

Salida (miArchivo.txt)

```
Hello World Writing into a file    myFile.txt
```

```
[87,114,105,116,101,32,115,111,109,101,32,65,83,67,73,73,32,118,97,108,117,101,111]
```

Del mismo modo, también podemos leer desde archivos. Veamos algún ejemplo de lectura de un archivo.

El ver y lo visto

Cuando queremos leer desde un archivo, no desde el teclado, tenemos que cambiar el flujo de entrada actual. Entonces podemos usar el predicado `see()`. Esto tomará el nombre del archivo como entrada. Cuando se complete la operación de lectura, usaremos el comando `visto`.

Archivo de muestra (sample_predicate.txt)

```
likes(lili, cat).
likes(jhon, dog).
```

Producción

```
| ?- see('sample_predicate.txt'),
read(X),
read(Y),
seen,
read(Z).
the_end.
```

```
X = end_of_file
Y = end_of_file
Z = the_end
```

```
yes
| ?-
```

Entonces, en este ejemplo, podemos ver que usando el predicado `see()` podemos leer el archivo. Ahora, después de usar el comando **visto**, el control se transfiere nuevamente a la consola. Finalmente recibe información de la consola.

Procesamiento de archivos de términos

Hemos visto cómo leer contenidos específicos (unas pocas líneas) de un archivo. Ahora, si queremos leer/procesar todo el contenido de un archivo, necesitamos escribir una cláusula para procesar el archivo (`process_file`), hasta llegar al final del archivo.

Programa

```
process_file :-
    read(Line),
    Line \== end_of_file, % when Line is not not end of file, call process.
    process(Line).
process_file :- !. % use cut to stop backtracking

process(Line):- %this will print the line into the console
    write(Line),nl,
    process_file.
```

Archivo de muestra (sample_predicate.txt)

```
likes(lili, cat).
likes(jhon,dog).
domestic(dog).
domestic(cat).
```

Producción

```
| ?- [process_file].
compiling D:/TP Prolog/Sample_Codes/process_file.pl for byte code...
D:/TP Prolog/Sample_Codes/process_file.pl compiled, 9 lines read - 774 bytes written, 2:

yes
| ?- see('sample_predicate.txt'), process_file, seen.
likes(lili,cat)
likes(jhon,dog)
domestic(dog)
domestic(cat)
```

```
true ?
```

```
(15 ms) yes
```

```
| ?-
```

Manipular personajes

Usando `read()` y `write()` podemos leer o escribir el valor de átomos, predicados, cadenas, etc. Ahora, en esta sección veremos cómo escribir caracteres individuales en el flujo de salida actual, o cómo leer desde el flujo de entrada actual. . Entonces existen algunos predicados predefinidos para realizar estas tareas.

Los predicados `put(C)` y `put_char(C)`

Podemos usar `put(C)` para escribir un carácter a la vez en el flujo de salida actual. El flujo de salida puede ser un archivo o la consola. Esta `C` puede ser un carácter o un código ASCII en otra versión de Prolog como SWI prolog, pero en GNU prolog solo admite el valor ASCII. Para usar el carácter en lugar de ASCII, podemos usar `put_char(C)`.

Programa

```
| ?- put(97),put(98),put(99),put(100),put(101).  
abcde
```

```
yes  
| ?- put(97),put(66),put(99),put(100),put(101).  
aBcde
```

```
(15 ms) yes  
| ?- put(65),put(66),put(99),put(100),put(101).  
ABcde
```

```
yes  
| ?-put_char('h'),put_char('e'),put_char('l'),put_char('l'),put_char('o').  
hello
```

```
yes  
| ?-
```

Los predicados get_char(C) y get_code(C)

Para leer un solo carácter del flujo de entrada actual, podemos usar el predicado get_char(C). Esto tomará el personaje. si queremos el código ASCII, podemos usar get_code(C).

Programa

```
| ?- get_char(X).
A.

X = 'A'

yes
uncaught exception: error(syntax_error('user_input:6 (char:689) expression ex
| ?- get_code(X).
A.

X = 65

yes
uncaught exception: error(syntax_error('user_input:7 (char:14) expression exp
| ?-
```

Construyendo átomos

La construcción de átomos significa que a partir de una lista de caracteres, podemos crear un átomo, o de una lista de valores ASCII también podemos crear átomos. Para hacer esto, tenemos que usar los predicados atom_chars() y atom_codes(). En ambos casos, el primer argumento será una variable y el segundo argumento será una lista. Entonces atom_chars() construye átomos a partir de caracteres, pero atom_codes() construye átomos a partir de una secuencia ASCII.

Ejemplo

```
| ?- atom_chars(X, ['t','i','g','e','r']).

X = tiger
```

```
yes
| ?- atom_chars(A, ['t','o','m']).

A = tom

yes
| ?- atom_codes(X, [97,98,99,100,101]).

X = abcde

yes
| ?- atom_codes(A, [97,98,99]).

A = abc

yes
| ?-
```

Átomos en descomposición

El átomo se descompone significa que a partir de un átomo, podemos obtener una secuencia de caracteres, o una secuencia de códigos ASCII. Para hacer esto, tenemos que usar los mismos predicados `atom_chars()` y `atom_codes()`. Pero una diferencia es que, en ambos casos, el primer argumento será un átomo y el segundo argumento será una variable. Entonces `atom_chars()` descompone el átomo en caracteres, pero `atom_codes()` descompone los átomos en una secuencia ASCII.

Ejemplo

```
| ?- atom_chars(tiger,X).

X = [t,i,g,e,r]

yes
| ?- atom_chars(tom,A).

A = [t,o,m]

yes
| ?- atom_codes(tiger,X).
```

```
X = [116,105,103,101,114]
```

```
yes
```

```
| ?- atom_codes(tom,A).
```

```
A = [116,111,109]
```

```
(16 ms) yes
```

```
| ?-
```

La consulta en Prolog

La consultoría es una técnica que se utiliza para fusionar predicados de diferentes archivos. Podemos usar el predicado consultar() y pasar el nombre del archivo para adjuntar los predicados. Veamos un programa de ejemplo para entender este concepto.

Supongamos que tenemos dos archivos, a saber, prog1.pl y prog2.pl.

Programa (prog1.pl)

```
likes(mary,cat).  
likes(joy,rabbit).  
likes(tim,duck).
```

Programa (prog2.pl)

```
likes(suman,mouse).  
likes(angshu,deer).
```

Producción

```
| ?- [prog1].
```

```
compiling D:/TP Prolog/Sample_Codes/prog1.pl for byte code...
```

```
D:/TP Prolog/Sample_Codes/prog1.pl compiled, 2 lines read - 443 bytes written, 23 ms
```

```
yes
```

```
| ?- likes(joy,rabbit).
```



```
yes
| ?- likes(suman,mouse).

no
| ?- consult('prog2.pl').
compiling D:/TP Prolog/Sample_Codes/prog2.pl for byte code...
D:/TP Prolog/Sample_Codes/prog2.pl compiled, 1 lines read - 366 bytes written, 20 ms
warning: D:/TP Prolog/Sample_Codes/prog2.pl:1: redefining procedure likes/2
        D:/TP Prolog/Sample_Codes/prog1.pl:1: previous definition

yes
| ?- likes(suman,mouse).

yes
| ?- likes(joy,rabbit).

no
| ?-
```

Ahora bien, a partir de este resultado podemos entender que esto no es tan simple como parece. Si dos archivos tienen **cláusulas completamente diferentes**, funcionará bien. Pero si hay los mismos predicados, mientras intentamos consultar el archivo, verificará los predicados del segundo archivo, cuando encuentre alguna coincidencia, simplemente eliminará todas las entradas de los mismos predicados de la base de datos local y luego cargará ellos nuevamente desde el segundo archivo.

Prólogo: predicados integrados

En Prolog, hemos visto los predicados definidos por el usuario en la mayoría de los casos, pero hay algunos predicados integrados. Hay tres tipos de predicados integrados, como se indica a continuación:

- Identificar términos
- Estructuras en descomposición
- Recopilando todas las soluciones

Esta es la lista de algunos predicados que se incluyen en el grupo de términos de identificación:

Predicado	Descripción
var(X)	tiene éxito si X es actualmente una variable sin instancias.
nueva(X)	tiene éxito si X no es una variable o ya se ha creado una instancia
átomo(X)	es cierto si X actualmente representa un átomo
número(X)	es cierto si X actualmente representa un número
entero(X)	es cierto si X actualmente representa un número entero
flotador(X)	es cierto si X actualmente representa un número real.
atómico(X)	es cierto si X actualmente representa un número o un átomo.
compuesto(X)	es cierto si X actualmente representa una estructura.
tierra(X)	tiene éxito si X no contiene ninguna variable no instanciada.

El predicado var(X)

Cuando X no se inicializa, se mostrará verdadero; de lo contrario, se mostrará falso. Entonces veamos un ejemplo.

Ejemplo



```
| ?- var(X).
```

yes

```
| ?- X = 5, var(X).
```

no

```
| ?- var([X]).
```

no

```
| ?-
```

El predicado novar(X)

Cuando X no se inicializa, se mostrará falso; de lo contrario, será verdadero. Entonces veamos un ejemplo.

Ejemplo

```
| ?- nonvar(X).
```

no

```
| ?- X = 5, nonvar(X).
```

X = 5

yes

```
| ?- nonvar([X]).
```

yes

```
| ?-
```

El predicado del átomo (X)

Esto devolverá verdadero cuando un término no variable con argumento 0 y un término no numérico se pase como X; de lo contrario, será falso.

Ejemplo

```
| ?- atom(paul).
```

yes

```
| ?- X = paul,atom(X).
```

X = paul

yes

```
| ?- atom([]).
```

yes

```
| ?- atom([a,b]).
```

no

```
| ?-
```

El predicado número(X)

Esto devolverá verdadero, X representa cualquier número; de lo contrario, será falso.

Ejemplo

```
| ?- number(X).
```

no

```
| ?- X=5,number(X).
```

X = 5

yes

```
| ?- number(5.46).
```

yes

```
| ?-
```

El predicado entero(X)

Esto devolverá verdadero, cuando X sea un valor entero positivo o negativo; de lo contrario, será falso.

Ejemplo

```
| ?- integer(5).
```

yes

```
| ?- integer(5.46).
```

no

```
| ?-
```

El predicado flotante (X)

Esto devolverá verdadero, X es un número de punto flotante; de lo contrario, será falso.

Ejemplo

```
| ?- float(5).
```

no

```
| ?- float(5.46).
```

yes

```
| ?-
```

El predicado atómico (X)

Tenemos atom(X), que es demasiado específico, devuelve falso para datos numéricos, atomic(X) es como atom(X) pero acepta números.

Ejemplo

```
| ?- atom(5).
```

no

```
| ?- atomic(5).
```

```
yes
```

```
| ?-
```

El predicado compuesto (X)

Si `atomic(X)` falla, entonces los términos son una variable no instanciada (que se puede probar con `var(X)`) o un término compuesto. El compuesto será verdadero cuando pasemos alguna estructura compuesta.

Ejemplo

```
| ?- compound([]).
```

```
no
```

```
| ?- compound([a]).
```

```
yes
```

```
| ?- compound(b(a)).
```

```
yes
```

```
| ?-
```

El predicado fundamental (X)

Esto devolverá verdadero si `X` no contiene ninguna variable no instanciada. Esto también verifica el interior de los términos compuestos; de lo contrario, devuelve falso.

Ejemplo

```
| ?- ground(X).
```

```
no
```

```
| ?- ground(a(b,X)).
```

```
no
```

```
| ?- ground(a).
```



```
yes
| ?- ground([a,b,c]).
```

```
yes
| ?-
```

Estructuras en descomposición

Ahora veremos otro grupo de predicados integrados, que son estructuras en descomposición. Hemos visto los términos identificativos antes. Entonces, cuando usamos estructuras compuestas, no podemos usar una variable para verificar o crear un functor. Devolverá error. Por tanto, el nombre del functor no puede representarse mediante una variable.

Error

```
X = tree, Y = X(maple).
Syntax error Y=X<>(maple)
```

Ahora, veamos algunos predicados incorporados que pertenecen al grupo de estructuras en descomposición.

El functor(T,F,N) Predicado

Esto devuelve verdadero si F es el functor principal de T y N es la aridad de F.

Nota : Arity significa la cantidad de atributos.

Ejemplo

```
| ?- functor(t(f(X),a,T),Func,N).

Func = t
N = 3

(15 ms) yes
| ?-
```

El predicado arg(N,Término,A)

Esto devuelve verdadero si A es el enésimo argumento del término. De lo contrario devuelve falso.

Ejemplo

```
| ?- arg(1,t(t(X),[]),A).
```

```
A = t(X)
```

```
yes
```

```
| ?- arg(2,t(t(X),[]),A).
```

```
A = []
```

```
yes
```

```
| ?-
```

Ahora veamos otro ejemplo. En este ejemplo, estamos comprobando que el primer argumento de D será 12, el segundo argumento será abr y el tercer argumento será 2020.

Ejemplo

```
| ?- functor(D,date,3), arg(1,D,12), arg(2,D,apr), arg(3,D,2020).
```

```
D = date(12,apr,2020)
```

```
yes
```

```
| ?-
```

El predicado ../2

Este es otro predicado representado como doble punto (..). Esto requiere 2 argumentos, por lo que se escribe '/2'. Entonces Term = .. L, esto es cierto si L es una lista que contiene el funtor de Term, seguido de sus argumentos.

Ejemplo

```
| ?- f(a,b) =.. L.

L = [f,a,b]

yes
| ?- T =.. [is_blue,sam,today].

T = is_blue(sam,today)

yes
| ?-
```

Al representar el componente de una estructura como una lista, se pueden procesar de forma recursiva sin conocer el nombre del funtor. Veamos otro ejemplo:

Ejemplo

```
| ?- f(2,3)=..[F,N|Y], N1 is N*3, L=..[F,N1|Y].

F = f
L = f(6,3)
N = 2
N1 = 6
Y = [3]

yes
| ?-
```

Recopilación de todas las soluciones

Ahora veamos la tercera categoría llamada recopilación de todas las soluciones, que se incluye en los predicados integrados en Prolog.

Hemos visto que para generar todas las soluciones dadas de un objetivo determinado se utiliza el punto y coma en el mensaje. Así que aquí tienes un ejemplo de ello.

Ejemplo


```
| ?- member(X, [1,2,3,4]).
```

```
X = 1 ? ;
```

```
X = 2 ? ;
```

```
X = 3 ? ;
```

```
X = 4
```

```
yes
```

A veces, necesitamos generar todas las soluciones para algún objetivo dentro de un programa en algunas aplicaciones relacionadas con la IA. Entonces, hay tres predicados integrados que nos ayudarán a obtener los resultados. Estos predicados son los siguientes:

- encontrar todo/3
- compensación/3
- bolsa de/3

Estos tres predicados toman tres argumentos, por lo que hemos escrito '/3' después del nombre de los predicados.

Estos también se conocen como **metapredicados**. Estos se utilizan para manipular la estrategia de prueba de Prolog.

Sintaxis

```
findall(X,P,L).
```

```
setof(X,P,L)
```

```
bagof(X,P,L)
```

Estos tres predicados predicen una lista de todos los objetos X, de modo que se satisface el objetivo P (ejemplo: edad (X, Edad)). Todos llaman repetidamente al objetivo P, instanciando la variable X dentro de P y agregándola a la lista L. Esto se detiene cuando no hay más solución.

Findall/3, Setof/3 y Bolsaof/3

Aquí veremos los tres predicados integrados diferentes `findall/3`, `setof/3` y `bagof/3`, que caen en la categoría, **recopilando todas las soluciones**.

El predicado `findall/3`

Este predicado se usa para hacer una lista de todas las soluciones X , a partir del predicado P . La lista devuelta será L . Entonces leeremos esto como "encontrar todas las X , de modo que X sea una solución del predicado P y poner la lista de resultados en L ". Aquí, este predicado almacena los resultados en el mismo orden en que Prolog los encuentra. Y si hay soluciones duplicadas, todas entrarán en la lista resultante, y si hay soluciones infinitas, el proceso nunca terminará.

Ahora también podemos hacer algunos avances en ellos. El segundo argumento, que es el objetivo, podría ser un objetivo compuesto. Entonces la sintaxis será **`findall(X, (Predicado en X, otro objetivo), L)`**

Y además el primer argumento puede ser un término de cualquier complejidad. Entonces, veamos los ejemplos de estas pocas reglas y verifiquemos el resultado.

Ejemplo

```
| ?- findall(X, member(X, [1,2,3,4]), Results).
```

```
Results = [1,2,3,4]
```

yes

```
| ?- findall(X, (member(X, [1,2,3,4]), X > 2), Results).
```

```
Results = [3,4]
```

yes

```
| ?- findall(X/Y, (member(X,[1,2,3,4]), Y is X * X), Results).
```

```
Results = [1/1,2/4,3/9,4/16]
```

yes

```
| ?-
```

El predicado `setof/3`

El `setof/3` también es como `findall/3`, pero aquí elimina todas las salidas duplicadas y las respuestas se ordenarán.

Si se utiliza alguna variable en el objetivo, entonces no aparecerá en el primer argumento, setof/3 devolverá un resultado separado para cada posible instanciación de esa variable.

Veamos un ejemplo para entender este conjunto de/3. Supongamos que tenemos una base de conocimientos como se muestra a continuación:

```
age(peter, 7).  
age(ann, 5).  
age(pat, 8).  
age(tom, 5).  
age(ann, 5).
```

Aquí podemos ver que age(ann, 5) tiene dos entradas en la base de conocimientos. Y las edades no están ordenadas y los nombres no están ordenados lexicográficamente en este caso. Ahora veamos un ejemplo del uso de setof/3.

Ejemplo

```
| ?- setof(Child, age(Child,Age),Results).  
  
Age = 5  
Results = [ann,tom] ? ;  
  
Age = 7  
Results = [peter] ? ;  
  
Age = 8  
Results = [pat]  
  
(16 ms) yes  
| ?-
```

Aquí podemos ver las edades y los nombres, ambos vienen ordenados. Para los 5 años, hay dos entradas, por lo que el predicado ha creado una lista correspondiente al valor de edad, con dos elementos. Y la entrada duplicada está presente sólo una vez.

Podemos usar la llamada anidada de setof/3 para recopilar los resultados individuales. Veremos otro ejemplo, donde el primer argumento será Edad/Niños. Como segundo argumento, se necesitará otro conjunto de como antes. Entonces

esto devolverá una lista del par (edad/Niños). Veamos esto en la ejecución del prólogo:

Ejemplo

```
| ?- setof(Age/Children, setof(Child,age(Child,Age), Children), AllResults).  
  
AllResults = [5/[ann,tom],7/[peter],8/[pat]]  
  
yes  
| ?-
```

Ahora, si no nos importa una variable que no aparece en el primer argumento, podemos usar el siguiente ejemplo:

Ejemplo

```
| ?- setof(Child, Age^age(Child,Age), Results).  
  
Results = [ann,pat,peter,tom]  
  
yes  
| ?-
```

Aquí estamos usando el símbolo de intercalación superior (^), esto indica que la Edad no está en el primer argumento. Entonces leeremos esto como: "Encuentre el conjunto de todos los niños, de modo que el niño tenga una Edad (cualquiera que sea) y ponga el resultado en Resultados".

El predicado bagof/3

bagof/3 es como setof/3, pero aquí no elimina las salidas duplicadas y es posible que las respuestas no se ordenen.

Veamos un ejemplo para entender este bagof/3. Supongamos que tenemos una base de conocimientos de la siguiente manera:

Base de conocimientos

```
age(peter, 7).  
age(ann, 5).  
age(pat, 8).  
age(tom, 5).  
age(ann, 5).
```

Ejemplo

```
| ?- bagof(Child, age(Child,Age),Results).
```

Age = 5

Results = [ann,tom,ann] ? ;

Age = 7

Results = [peter] ? ;

Age = 8

Results = [pat]

(15 ms) yes

```
| ?-
```

Aquí, para el valor de Edad 5, los resultados son [ann, tom, ann]. Entonces las respuestas no se ordenan y las entradas duplicadas no se eliminan, por lo que tenemos dos valores 'ann'.

bagof/3 es diferente de findall/3, ya que genera resultados separados para todas las variables en el objetivo que no aparecen en el primer argumento. Veremos esto usando un ejemplo a continuación:

Ejemplo

```
| ?- findall(Child, age(Child,Age),Results).
```

Results = [peter,ann,pat,tom,ann]

yes

```
| ?-
```

Predicados matemáticos

Los siguientes son los predicados matemáticos:

Predicados	Descripción
aleatorio(L,H,X).	Obtener valor aleatorio entre L y H
entre(L,H,X).	Obtener todos los valores entre L y H
éxito(X,Y).	Suma 1 y asígnalo a X
abs(X).	Obtener el valor absoluto de X
máx(X,Y).	Obtener el valor más grande entre X e Y
mín(X,Y).	Obtener el valor más pequeño entre X e Y
redondo(X).	Redondear un valor cercano a X
truncar(X).	Convierta flotante a entero, elimine la parte fraccionaria
piso(X).	Redondear a la baja
techo(X).	Redondeo
raíz cuadrada (X).	Raíz cuadrada

Además de estos, existen otros predicados como sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, tanh, asinh, acosh, atanh, log, log10, exp, pi, etc.

Ahora veamos estas funciones en acción usando un programa Prolog.

Ejemplo

```
| ?- random(0,10,X).
```

```
X = 0
```

```
yes
```

```
| ?- random(0,10,X).
```

```
X = 5
```

```
yes  
| ?- random(0,10,X).
```

```
X = 1
```

```
yes  
| ?- between(0,10,X).
```

```
X = 0 ? a
```

```
X = 1
```

```
X = 2
```

```
X = 3
```

```
X = 4
```

```
X = 5
```

```
X = 6
```

```
X = 7
```

```
X = 8
```

```
X = 9
```

```
X = 10
```

```
(31 ms) yes  
| ?- succ(2,X).
```

```
X = 3
```

```
yes  
| ?- X is abs(-8).
```

```
X = 8
```

```
yes
```



```
| ?- X is max(10,5).
```

```
X = 10
```

```
yes
```

```
| ?- X is min(10,5).
```

```
X = 5
```

```
yes
```

```
| ?- X is round(10.56).
```

```
X = 11
```

```
yes
```

```
| ?- X is truncate(10.56).
```

```
X = 10
```

```
yes
```

```
| ?- X is floor(10.56).
```

```
X = 10
```

```
yes
```

```
| ?- X is ceiling(10.56).
```

```
X = 11
```

```
yes
```

```
| ?- X is sqrt(144).
```

```
X = 12.0
```

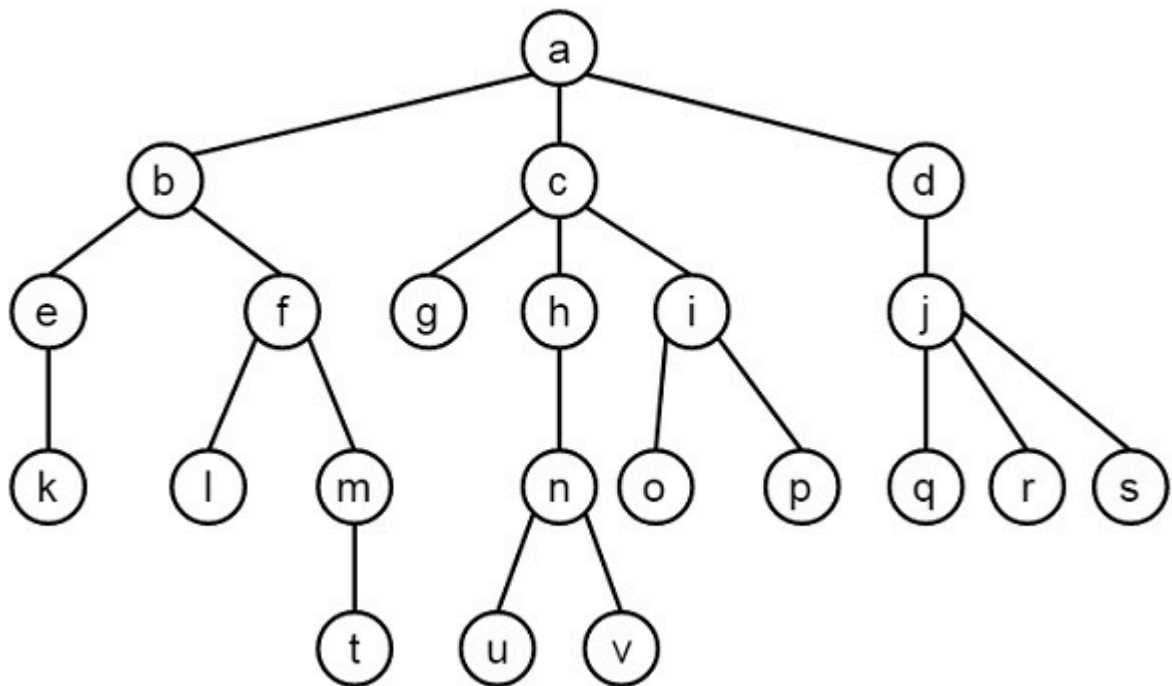
```
yes
```

```
| ?-
```


Estructura de datos de árbol (estudio de caso)

Hasta ahora hemos visto diferentes conceptos de programación lógica en Prolog. Ahora veremos un estudio de caso sobre Prolog. Veremos cómo implementar una estructura de datos de árbol usando Prolog y crearemos nuestros propios operadores. Así que comencemos la planificación.

Supongamos que tenemos un árbol como se muestra a continuación:



Tenemos que implementar este árbol usando prolog. Tenemos algunas operaciones de la siguiente manera:

- `op(500, xfx, 'is_parent').`
- `op(500, xfx, 'es_hermano_de').`
- `op(500, xfx, 'está_en_el_mismo_nivel').`
- Y otro predicado es `leaf_node(Node)`

En estos operadores, ha visto algunos parámetros como (500, xfx, <nombre_operador>). El primer argumento (aquí 500) es la prioridad de ese operador. 'xfx' indica que se trata de un operador binario y <operator_name> es el nombre del operador.

Estos operadores se pueden utilizar para definir la base de datos del árbol. Podemos usar estos operadores de la siguiente manera:



- **a es_parent b, o is_parent(a, b).** Entonces esto indica que el nodo a es el padre del nodo b.
- **X es_hermano_de Y o es_hermano_de(X,Y).** Esto indica que X es hermano del nodo Y. Entonces, la regla es que si otro nodo Z es padre de X y Z también es padre de Y y X e Y son diferentes, entonces X e Y son hermanos.
- **leaf_node(Nodo).** Se dice que un nodo (Nodo) es un nodo hoja cuando un nodo no tiene hijos.
- **X está_en_el_mismo_nivel Y, o está_en_el_mismo_nivel(X,Y).** Esto comprobará si X e Y están al mismo nivel o no. Entonces, la condición es que cuando X e Y son iguales, entonces devuelve verdadero; de lo contrario, W es el padre de X, Z es el padre de Y y W y Z están en el mismo nivel.

Como se muestra arriba, otras reglas están definidas en el código. Así que veamos el programa para verlo mejor.

Programa

```
/* The tree database */

:- op(500,xfx,'is_parent').

a is_parent b. c is_parent g. f is_parent l. j is_parent q.
a is_parent c. c is_parent h. f is_parent m. j is_parent r.
a is_parent d. c is_parent i. h is_parent n. j is_parent s.
b is_parent e. d is_parent j. i is_parent o. m is_parent t.
b is_parent f. e is_parent k. i is_parent p. n is_parent u.
n
is_parent v.
/* X and Y are siblings i.e. child from the same parent */

:- op(500,xfx,'is_sibling_of').

X is_sibling_of Y :- Z is_parent X,
                    Z is_parent Y,
                    X \== Y.

leaf_node(Node) :- \+ is_parent(Node,Child). % Node grounded

/* X and Y are on the same level in the tree. */
```

```
:-op(500,xfx,'is_at_same_level').  
X is_at_same_level X .  
X is_at_same_level Y :- W is_parent X,  
                        Z is_parent Y,  
                        W is_at_same_level Z.
```

Producción

```
| ?- [case_tree].  
compiling D:/TP Prolog/Sample_Codes/case_tree.pl for byte code...  
D:/TP Prolog/Sample_Codes/case_tree.pl:20: warning: singleton variables [Child] for lea  
D:/TP Prolog/Sample_Codes/case_tree.pl compiled, 28 lines read - 3244 bytes written, 7
```

yes

```
| ?- i is_parent p.
```

yes

```
| ?- i is_parent s.
```

no

```
| ?- is_parent(i,p).
```

yes

```
| ?- e is_sibling_of f.
```

true ?

yes

```
| ?- is_sibling_of(e,g).
```

no

```
| ?- leaf_node(v).
```

yes

```
| ?- leaf_node(a).
```

no

```
| ?- is_at_same_level(l,s).
```

true ?



yes

| ?- ! is_at_same_level v.

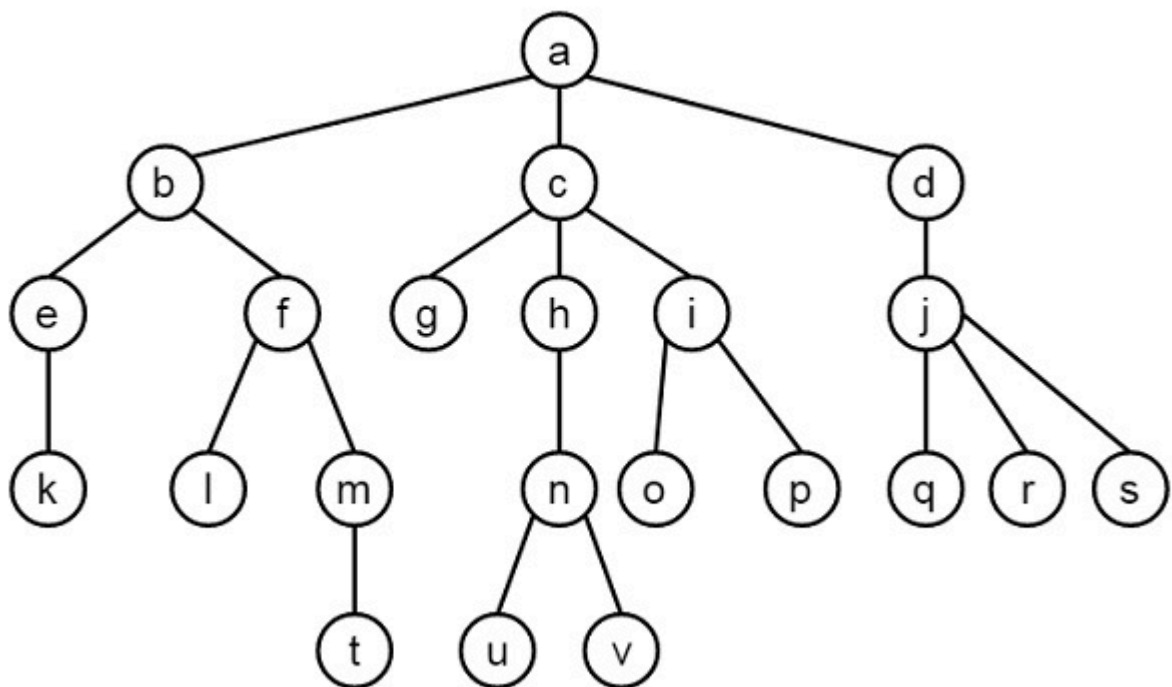
no

| ?-

Más sobre la estructura de datos del árbol

Aquí, veremos algunas operaciones más que se realizarán en la estructura de datos del árbol dada anteriormente.

Consideremos el mismo árbol aquí:



Definiremos otras operaciones:

- ruta (nodo)
- localizar (nodo)

Como hemos creado la última base de datos, crearemos un nuevo programa que contendrá estas operaciones, luego consultaremos el nuevo archivo para usar estas operaciones en nuestro programa preexistente.

Entonces, veamos cuál es el propósito de estos operadores:

- **ruta (Nodo)** : esto mostrará la ruta desde el nodo raíz hasta el nodo dado. Para resolver esto, supongamos que X es el padre del nodo, luego busque la

ruta (X) y luego escriba X. Cuando se alcance el nodo raíz 'a', se detendrá.

- **localizar (Nodo)** : esto ubicará un nodo (Nodo) desde la raíz del árbol. En este caso, llamaremos a la ruta (Nodo) y escribiremos el Nodo.

Programa

Veamos el programa en ejecución:

```
path(a).                                     /* Can start at a. */
path(Node) :- Mother is_parent Node, /* Choose parent, */
              path(Mother),           /* find path and then */
              write(Mother),
              write(' --> ').

/* Locate node by finding a path from root down to the node */
locate(Node) :- path(Node),
                write(Node),
                nl.
```

Producción

```
| ?- consult('case_tree_more.pl').
compiling D:/TP Prolog/Sample_Codes/case_tree_more.pl for byte code...
D:/TP Prolog/Sample_Codes/case_tree_more.pl compiled, 9 lines read - 866 bytes writte

yes
| ?- path(n).
a --> c --> h -->

true ?

yes
| ?- path(s).
a --> d --> j -->

true ?

yes
| ?- path(w).
```

```
no  
| ?- locate(n).  
a --> c --> h --> n
```

true ?

```
yes  
| ?- locate(s).  
a --> d --> j --> s
```

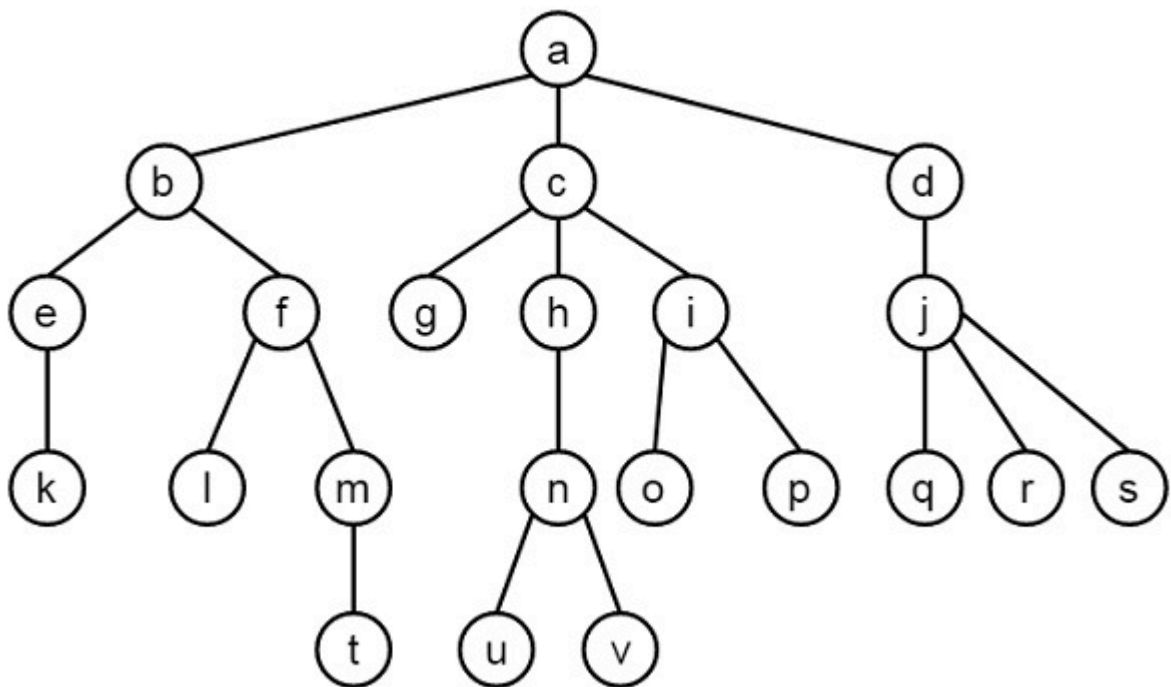
true ?

```
yes  
| ?- locate(w).
```

```
no  
| ?-
```

Avances en estructuras de datos de árbol

Ahora definamos algunas operaciones avanzadas en la misma estructura de datos de árbol.



Aquí veremos cómo encontrar la altura de un nodo, es decir, la longitud de la ruta más larga desde ese nodo, utilizando el predicado integrado `setof/3` de Prolog. Este

predicado toma (Plantilla, Objetivo, Conjunto). Esto vincula Set a la lista de todas las instancias de Plantilla que satisfacen el objetivo Objetivo.

Ya hemos definido el árbol antes, por lo que consultaremos el código actual para ejecutar este conjunto de operaciones sin redefinir la base de datos del árbol nuevamente.

Crearemos algunos predicados de la siguiente manera:

ht(Nodo,H). Esto encuentra la altura. También verifica si un nodo es hoja o no; de ser así, establece la altura H en 0; de lo contrario, encuentra de forma recursiva la altura de los hijos del nodo y les suma 1.

máx([X|R], M,A). Esto calcula el elemento máximo de la lista y un valor M. Entonces, si M es máximo, entonces devuelve M; de lo contrario, devuelve el elemento máximo de la lista que es mayor que M. Para resolver esto, si la lista dada está vacía, devuelva M como elemento máximo; de lo contrario, verifique si Head es mayor que M o no; de ser así, llame a max() usando la parte de la cola y el valor X; de lo contrario, llame a max() usando la cola y el valor M.

altura(N,H). Esto utiliza el predicado setof/3. Esto encontrará el conjunto de resultados utilizando el objetivo ht(N,Z) para la plantilla Z y lo almacenará en la variable de tipo lista llamada Conjunto. Ahora encuentre el máximo de Set y el valor 0, almacene el resultado en H.

Ahora veamos el programa en ejecución:

Programa

```
height(N,H) :- setof(Z,ht(N,Z),Set),
               max(Set,0,H).

ht(Node,0) :- leaf_node(Node),!.
ht(Node,H) :- Node is_parent Child,
               ht(Child,H1),
               H is H1 + 1.

max([],M,M).
max([X|R],M,A) :- (X > M -> max(R,X,A) ; max(R,M,A)).
```

Producción

```
| ?- consult('case_tree_adv.pl').
compiling D:/TP Prolog/Sample_Codes/case_tree_adv.pl for byte code...
```

D:/TP Prolog/Sample_Codes/case_tree_adv.pl compiled, 9 lines read - 2060 bytes written

yes

| ?- ht(c,H).

H = 1 ? a

H = 3

H = 3

H = 2

H = 2

yes

| ?- max([1,5,3,4,2],10,Max).

Max = 10

yes

| ?- max([1,5,3,40,2],10,Max).

Max = 40

yes

| ?- setof(H, ht(c,H),Set).

Set = [1,2,3]

yes

| ?- max([1,2,3],0,H).

H = 3

yes

| ?- height(c,H).

H = 3

yes

| ?- height(a,H).



H = 4

yes

| ?-