PROLOG

Paradigmas de la programacion

Gonzalez Ceseña Adan

372799

1.-Introduccion.

En **Programación Funcional**, tenemos que definir los procedimientos y establecer cómo funcionan los procedimientos. 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.

¿Qué es Prolog?

Prolog o **PRO**gramming 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.

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:

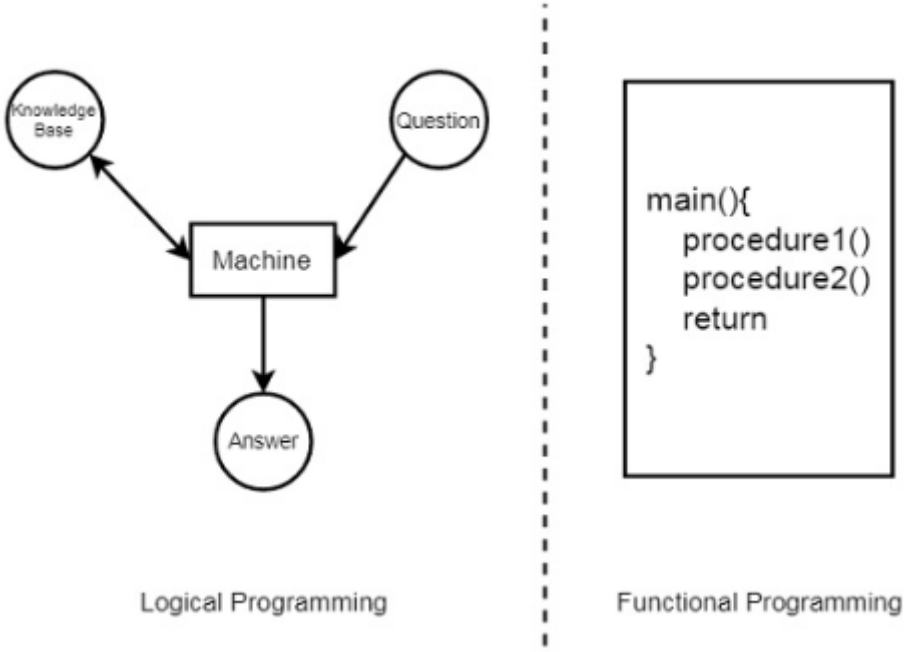
```
grandfather(X, Y) :- father(X, Z), parent(Z, Y)
% Esto significa que X es el abuelo de Y si X es el padre de Z y Z es el padre de
Y. %
```

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

Aplicaciones de Prolog:

- Intelligent Database Retrieval
- Natural Language Understanding
- Specification Language
- Machine Learning
- Robot Planning
- Automation System
- Problem Solving

La diferencia entre *programacion logica y programacion tradicional*, se pueden mostrar en la siguiente

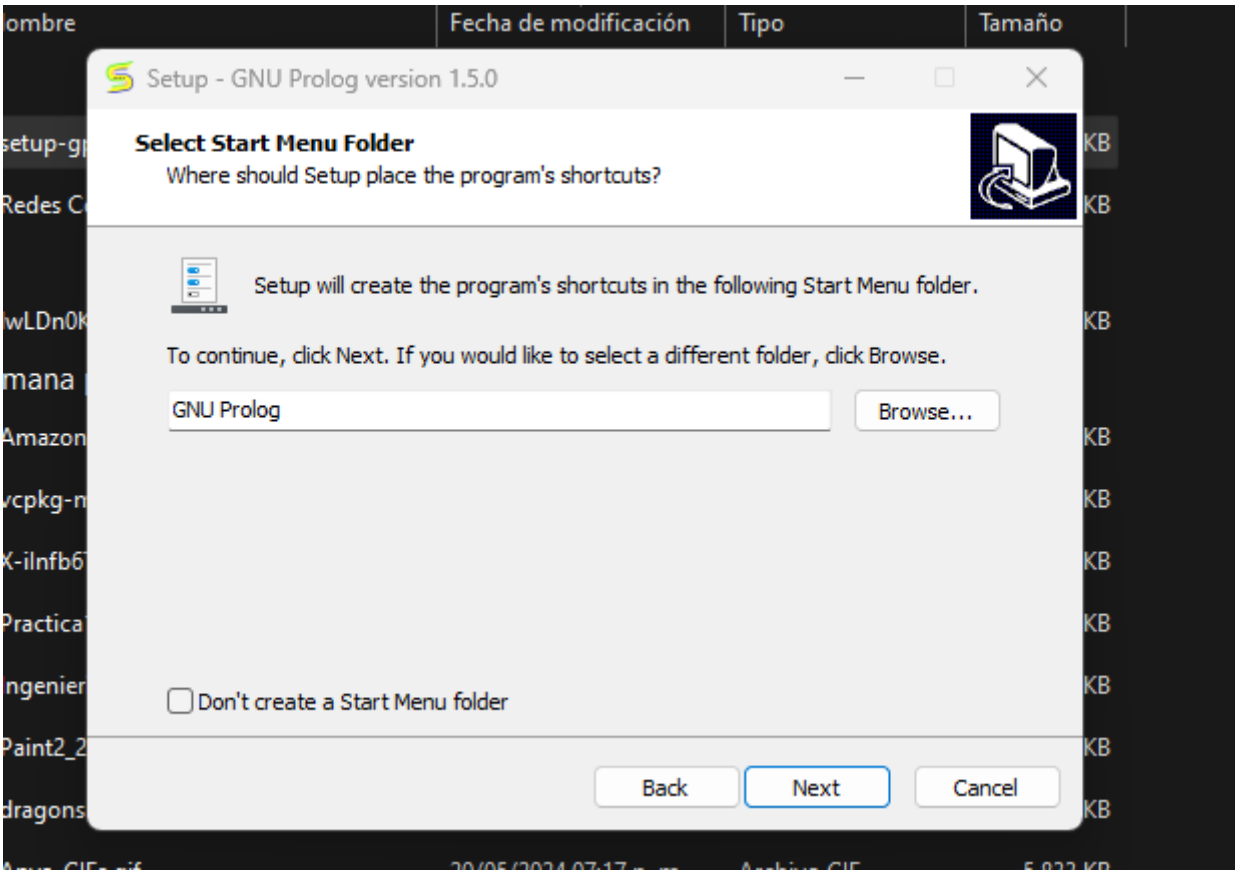


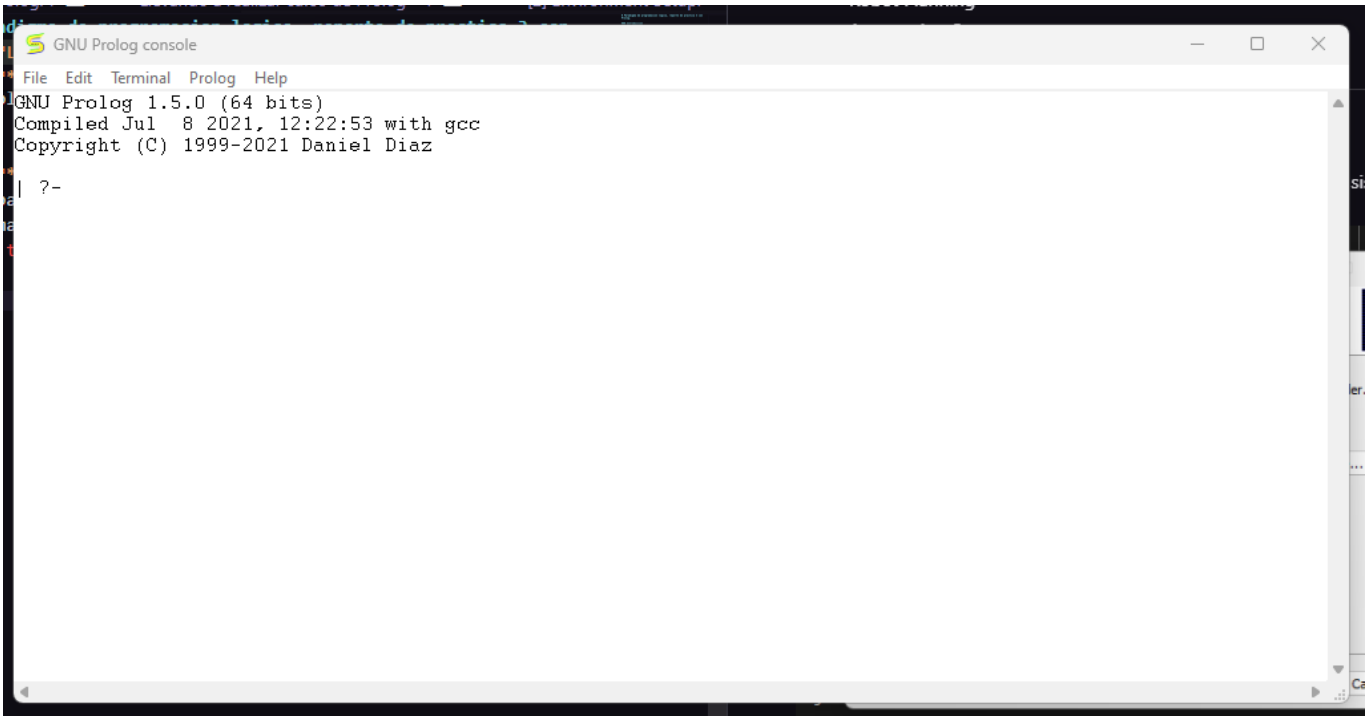
ilustracion:

---

[2] Environment Setup.

Pasos para instalar Prolog.





---

[3] Hello World.

Para escribir "Hello World" es necessario tener abierto el programa de Prolog, y escribir el siguiente codigo:

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

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, tenemos que escribir el nombre del archivo de la siguiente manera:

```
[hello_world]
```

---

[4] Basics.

FACTS:

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.

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, p. color(penink, 'rojo').
- foneno(agnibha, 1122334455). También se le llama predicado o cláusula.

Sintaxis

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

Ejemplo

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

RULES:

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 se indican 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.

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).
```

QUERIES:

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?
- ¿Ryan irá a jugar?

EJEMPLO

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.

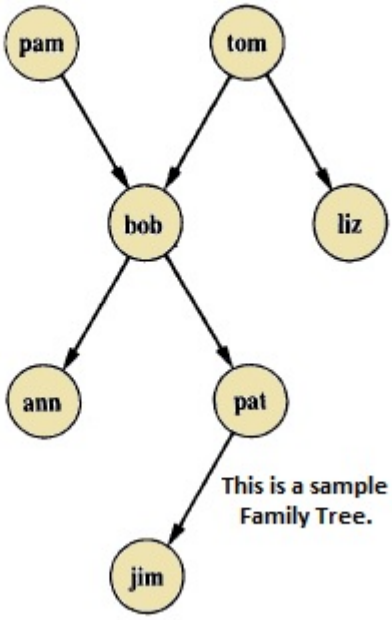
```
girl(priya).
girl(tiyasha).
girl(jaya).
can_cook(priya).
```

```
%Output%
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
| ?-
```

---

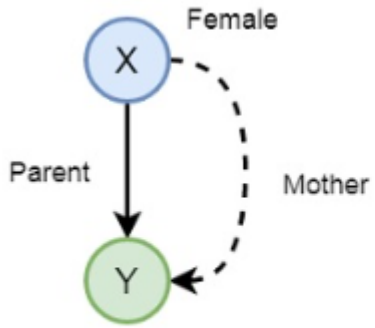
[5] Relations.

Suppose the family tree is as follows –

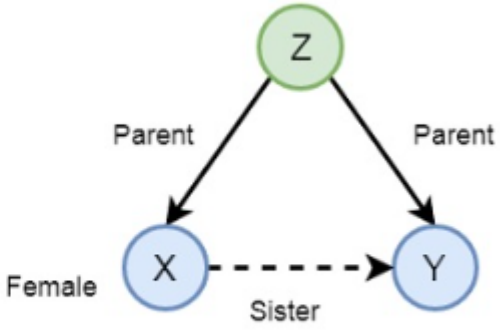


Aquí desde este árbol podemos entender que existen pocas relaciones. Aquí Bob es hijo de Pam y Tom, y Bob también tiene dos hijos: Ann y Pat. Bob tiene un hermano, Liz, cuyo padre también es Tom. Entonces queremos hacer predicados de la siguiente manera.

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



Mother Relationship



Sister Relationship

```
mother(X,Y) :- parent(X,Y), female(X).
sister(X,Y) :- parent(Z,X), parent(Z,Y), female(X), X \== Y.
```

Ejemplo

```
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.
```

```
% Output %
| ?- [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
| ?-
```

Seguimiento de la salida

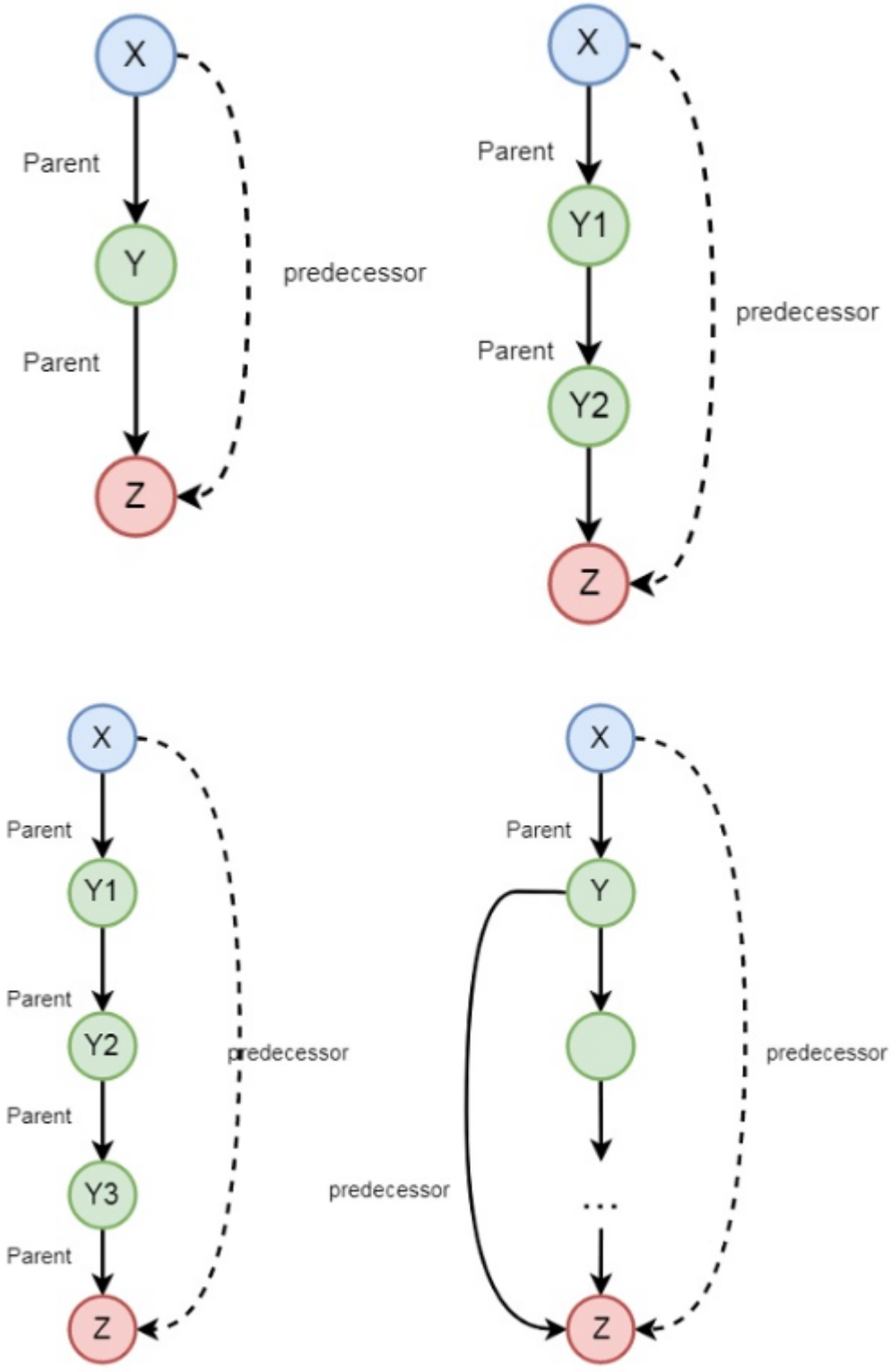
En Prolog podemos rastrear la ejecución. Para rastrear la salida, debe ingresar al modo de rastreo escribiendo "trace". Luego, en el resultado podemos ver que solo estamos rastreando "¿pam es madre de quién?".

```
| ?- [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, 10 ms

(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

Estas relaciones son de naturaleza estática. También podemos crear algunas relaciones recursivas que se pueden expresar en la siguiente ilustración.



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

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

male(jim).
```

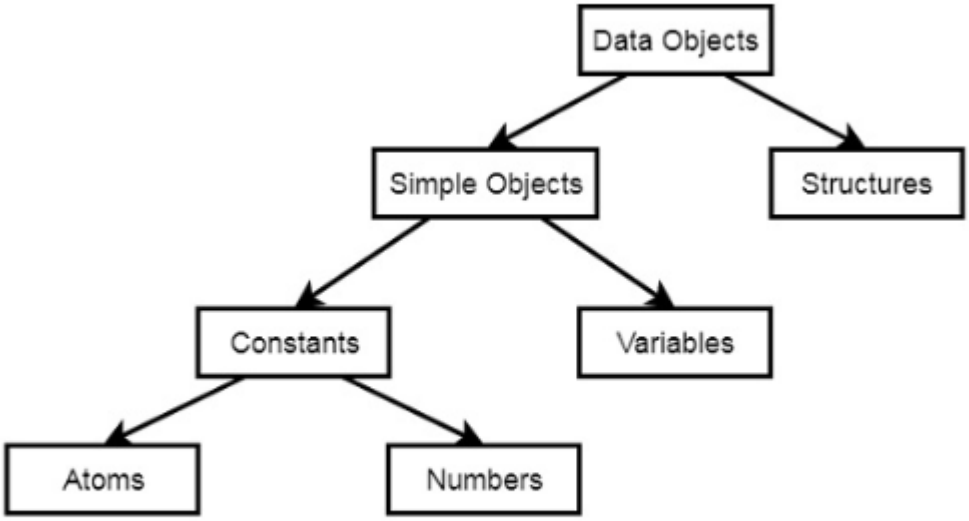
```
male(bob).
male(tom).
male(peter).

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).
```

```
%Output%
| ?- [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, 14 ms

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}
| ?-
```

---

[6] Data Objects.



Data objects in Prolog

Atoms

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

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

Cadenas de caracteres especiales Ejemplos:

- <--->
- ======>
- ...
- .:.
- ::=

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'

Numeros

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.

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.

- X
- Sum
- Memer_name
- Student_list
- Shoppinglist
- _a50
- _15

Variables anonimas en PROLOG

Las variables anónimas no tienen nombre. Las variables anónimas en el prólogo se escriben con un solo 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.

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

```
%Output%
| ?- [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, 16 ms

yes
| ?- hates(X,tom).

X = jim ? ;

X = peter
```

```
yes
| ?- hates(_,tom).

true ? ;

(16 ms) yes
| ?- hates(_,pat).

no
| ?- hates(_,fox).

true ? ;

no
| ?-
```

---

[7] Operators.

Comparación de operadores en Prolog

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

Operator	Meaning
X > Y	X is greater than Y
X < Y	X is less than Y
X >= Y	X is greater than or equal to Y
X =< Y	X is less than or equal to Y
X =:= Y	the X and Y values are equal
X =\= Y	the X and Y values are not equal

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
```

Operadores aritméticos en Prolog Los operadores aritméticos se utilizan para realizar operaciones aritméticas. Hay algunos tipos diferentes de operadores aritméticos de la siguiente manera

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power
//	Integer Division
mod	Modulus

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.
```

Output

```
| ?- 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 ms

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
| ?-
```

---

[8] Loop & Decision Making.

Loops

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

Programa

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

```
%Output%
| ?- [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
| ?-
```

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
```

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').
```

```
| ?- [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
| ?-
```

---

[9] Conjunciones y disyunciones.

Conjunciones

La conjunción (lógica Y) se puede implementar utilizando el operador coma (,). Así, dos predicados separados por una coma se combinan con una declaración AND. Por ejemplo, si tenemos el predicado parent(jhon, bob), que significa "Jhon es padre de Bob", y el predicado male(jhon), que significa "Jhon es masculino", podemos crear otro predicado, padre(jhon, bob), que significa "Jhon es padre de Bob y es varón". De esta manera, definimos el predicado padre cuando se cumplen ambas condiciones: ser padre Y ser varón.

Disyunciones

La disyunción (lógica OR) se puede implementar utilizando el operador punto y coma (😉. Así, dos predicados separados por un punto y coma se combinan con una declaración OR. Por ejemplo, si tenemos el predicado padre(jhon, bob), que significa "Jhon es padre de Bob", y el predicado madre(lili, bob), que significa "Lili es madre de Bob", podemos crear otro predicado como niño(), que será verdadero cuando sea verdadero padre(jhon, bob) O madre(lili, bob).

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).
```

```
%Output
| ?- [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 ms

yes
| ?- father(jhon,bob).

yes
| ?- child_of(jhon,bob).

true ?

yes
| ?- child_of(lili,bob).

yes
| ?-
```

---

[10] Lists.

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 indica a continuación.

- El primer elemento, denominado cabeza de lista;
- La parte restante de la lista se llama cola.
- [a, b, c] = [x | [b, c]]
- [a, b, c] = [a, b | [c]]
- ☐]

Operations	Definition
Membership Checking	During this operation, we can verify whether a given element is member of specified list or not?
Length Calculation	With this operation, we can find the length of a list.
Concatenation	Concatenation is an operation which is used to join/add two lists.
Delete Items	This operation removes the specified element from a list.
Append Items	Append operation adds one list into another (as an item).
Insert Items	This operation inserts a given item into a list.

Operaciones en listas

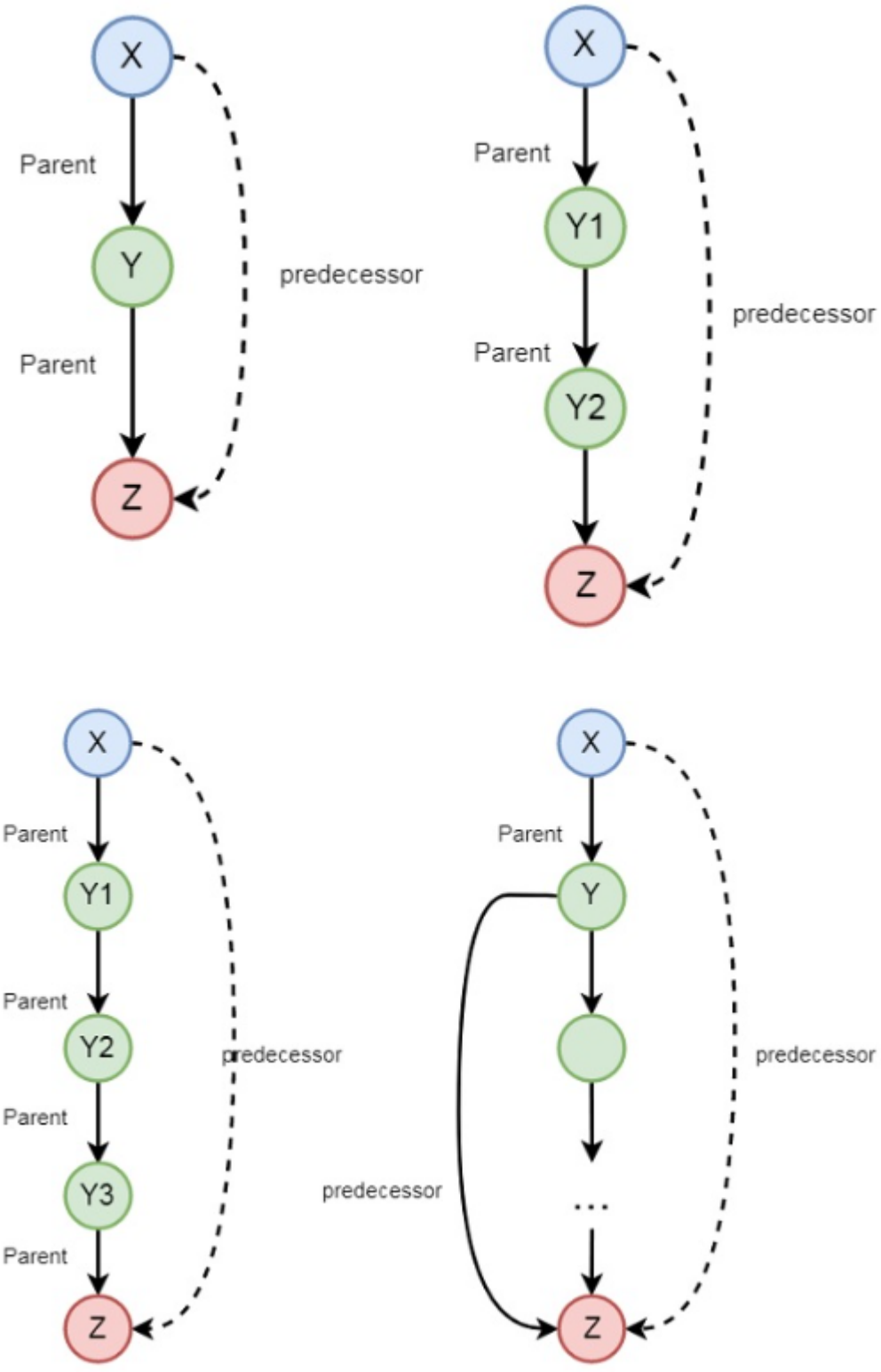
---

[11] Recursion and Structures.

Recursion

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.

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



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).

> [!NOTE]Nota: la estructura, a su vez, puede tener otra estructura como componente. Entonces podemos ver las vistas como estructura de árbol y funtores Prolog.

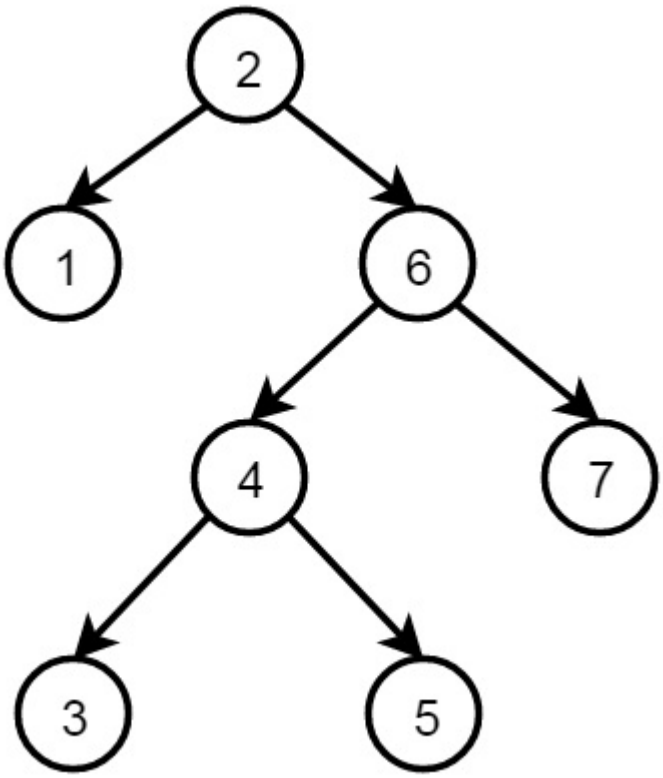
Matching in Prolog

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 fecha(D,M,2020) = fecha(D1,abr, Y1), esto indica que D = D1, M = feb y Y1 = 2020.

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

Arboless Binarios

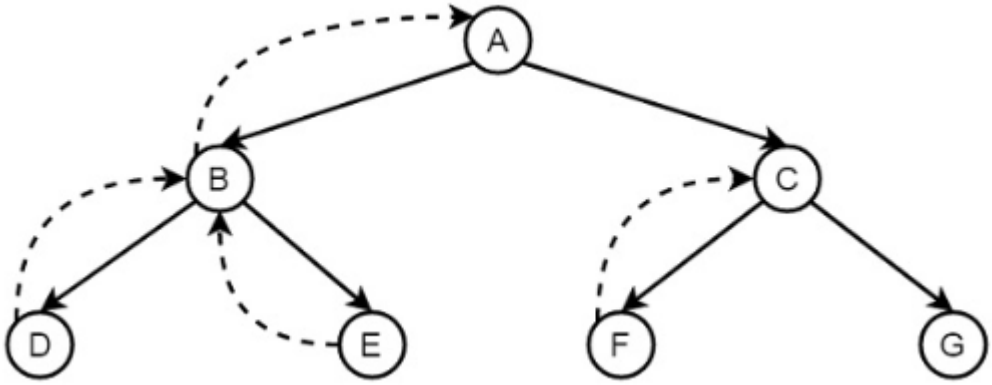



```
node(2, node(1,nil,nil), node(6, node(4,node(3,nil,nil), node(5,nil,nil)),
node(7,nil,nil))
```

Cada nodo tiene tres campos, datos y dos nodos. Un nodo sin estructura secundaria (nodo hoja) se escribe como nodo (valor, nulo, nulo), el nodo con un solo hijo izquierdo se escribe como nodo (valor, nodo_izquierdo, nulo), el nodo con solo un hijo derecho se escribe como nodo (valor, nulo; nodo_derecho), y el nodo con ambos hijos tiene un nodo (valor, nodo_izquierdo, nodo_derecho).

[12] Backtracking.

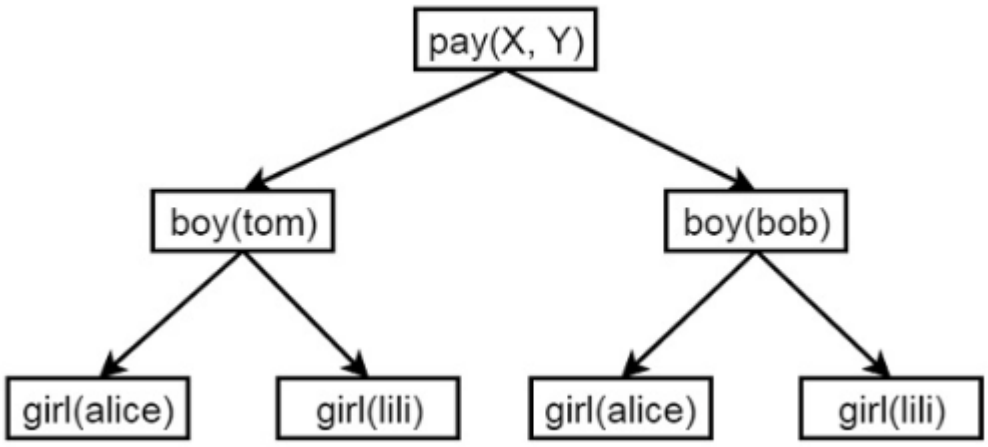
El término retroceso es bastante común en el diseño de algoritmos y en diferentes entornos de programación. En Prolog, hasta que llega al destino correcto, intenta retroceder. Cuando se encuentra el destino, se detiene.



¿Como funciona Backtracking en Prolog?

Programa

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

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




```
%Output
| ?- [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 ms

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}
| ?-
```

---

[13] Different and Not.

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.

Programa

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

```
% Output
| ?- [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/2
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
| ?-
```

---

[14] Inputs and Outputs.

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.

The write() Predicate Program

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

yes
| ?- write('hello').
hello

yes
| ?- write('hello'),nl,write('world').
hello
world

yes
| ?- write("ABCDE")
.
[65,66,67,68,69]

yes
```

The read() Predicate 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.
```

The tab() Predicate 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
| ?-
```

Reading/Writing Files

Prolog Commands

```
| ?- 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
| ?-
```

Output (myFile.txt)

```
Hello World Writing into a file     myFile.txt
[87,114,105,116,101,32,115,111,109,101,32,65,83,67,73,73,32,118,97,108,117,101,115
]
```

Manipulating characters

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.

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
| ?-
```

---

[15] Built-In Predicates.

- Identifying terms
- Decomposing structures
- Collecting all solutions

So this is the list of some predicates that are falls under the identifying terms group –

Predicate	Description
var(X)	succeeds if X is currently an un-instantiated variable.
novar(X)	succeeds if X is not a variable, or already instantiated
atom(X)	is true if X currently stands for an atom
number(X)	is true if X currently stands for a number
integer(X)	is true if X currently stands for an integer
float(X)	is true if X currently stands for a real number.
atomic(X)	is true if X currently stands for a number or an atom.
compound(X)	is true if X currently stands for a structure.
ground(X)	succeeds if X does not contain any un-instantiated variables.

Decomposing Structures

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 funtor no puede representarse mediante una variable.

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

Mathematical Predicates

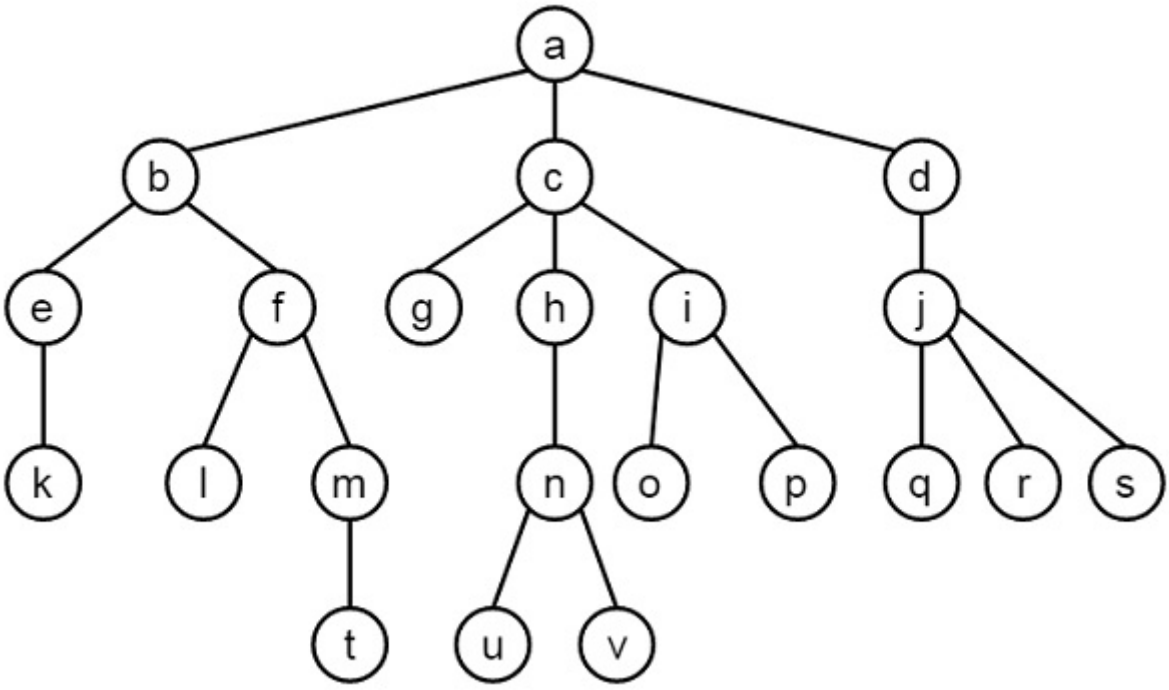
Following are the mathematical predicates –

Predicates	Description
random(L,H,X).	Get random value between L and H
between(L,H,X).	Get all values between L and H
succ(X,Y).	Add 1 and assign it to X
abs(X).	Get absolute value of X
max(X,Y).	Get largest value between X and Y
min(X,Y).	Get smallest value between X and Y
round(X).	Round a value near to X
truncate(X).	Convert float to integer, delete the fractional part
loor(X).	Round down
ceiling(X).	Round up
sqrt(X).	Square root

Besides these, there are some other predicates such as sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, tanh, asinh, acosh, atanh, log, log10, exp, pi, etc.

---

[16] Tree Data Structure (Case Study).



- op(500, xfx, ‘is_parent’).
- op(500, xfx, ‘is_sibling_of’).
- op(500, xfx, ‘is_at_same_level’).
- And another predicate namely leaf_node(Node)
- a is_parent b, or is_parent(a, b). So this indicates that node a is the parent of node b.
- X is_sibling_of Y or is_sibling_of(X,Y). This indicates that X is the sibling of node Y. So the rule is, if another node Z is parent of X and Z is also the parent of Y and X and Y are different, then X and Y are siblings.
- leaf_node(Node). A node (Node) is said to be a leaf node when a node has no children.
- X is_at_same_level Y, or is_at_same_level(X,Y). This will check whether X and Y are at the same level or not. So the condition is when X and Y are same, then it returns true, otherwise W is the parent of X, Z is the parent of Y and W and Z are at the same level.

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.
```

Output

```
| ?- [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 leaf_node/1
D:/TP Prolog/Sample_Codes/case_tree.pl compiled, 28 lines read - 3244 bytes
written, 7 ms

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
| ?- l is_at_same_level v.
```

```
no
| ?-
```

More on Tree Data Structure

Consideremos el mismo árbol aquí.

We will define other operations −

- path(Node)
- locate(Node)
- path(Node) − This will display the path from the root node to the given node. To solve this, suppose X is parent of Node, then find path(X), then write X. When root node 'a' is reached, it will stop.
- locate(Node) − This will locate a node (Node) from the root of the tree. In this case, we will call the path(Node) and write the Node.

Programa

```
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.
```