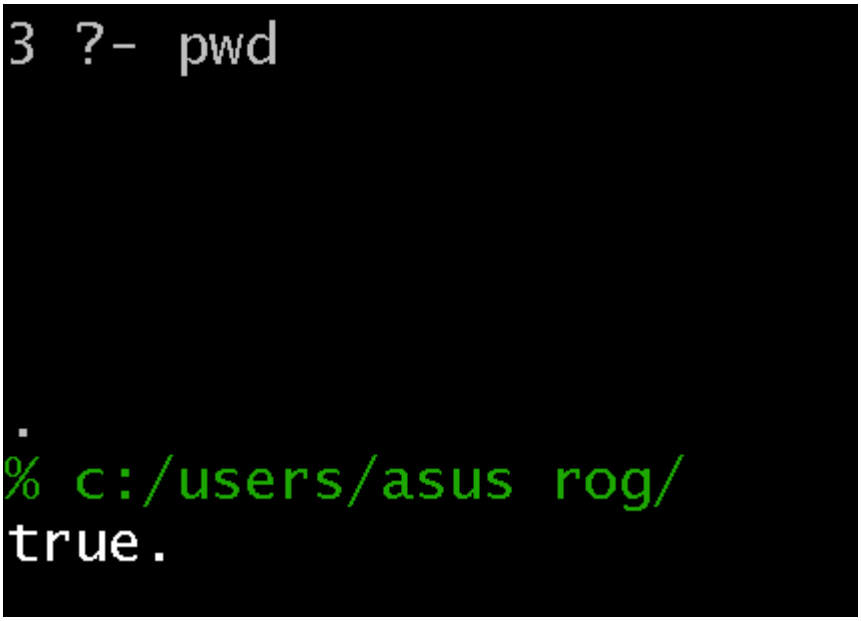
Implementación del Paradigma Lógico

Primera sesión

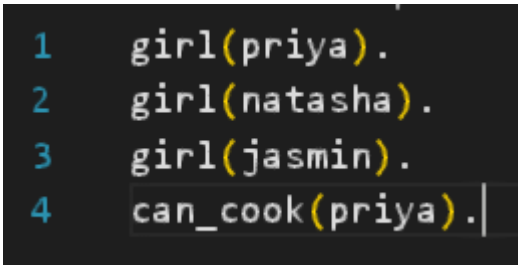
Durante la primera sesión de este Paradigma iniciamos instalando Prolog en nuestras computadoras, en donde configuramos el programa para que fuera posible utilizarlo sin problemas.

```
Asus ROG@DESKTOP-7AQ214R MINGW64 ~ (master)
$ swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 9.2.9)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

1 ?- pwd.
% c:/users/asus rog/
true.
```

Después probamos un poco el programa, en donde aprendimos que todos los comandos deben llevar punto al final, además de usar ; para conseguir otra respuesta posible.



Después de esto creamos un archivo para generar nuestras reglas y hechos. Los archivos de prolog terminan en .pl.



En este archivo generamos 3 personas (Priya, Natasha y Jasmin)m además de indicar que Priya sabe cocinar (can_cook).

```
1 ?- can_cook(priya).
true.

2 ?- can_cook(jasmin).
false.
```

Después creamos otro archivo, en el cual también definimos algunas reglas y hechos.

```
1   sing_a_song(ana).
2   listens_to_music(rodrigo).
3
4   listens_to_music(ana) :- sing_a_song(ana).
5
6   happy(ana) :- sing_a_song(ana).
7
8   happy(rodrigo) :- listens_to_music(rodrigo).
9   plays_guitar(rodrigo) :- listens_to_music(rodrigo).
```

```
1 ?- happy(rodrigo).
true.

2 ?- sing_a_song(ana).
true.

3 ?- sing_a_song(rodrigo).
false.
```

** Segunda sesión**

Durante la segunda sesión utilizamos otro archivo, en donde en esta ocasión imprimía dos mensajes en la consola utilizando write.

```
1   main :-
2   write('This is sample Prolog program'),
3   write(' This program is written into Imprimir.pl file').
```

```
1 ?- main.
This is sample Prolog program This program is written into Imprimir.pl file
true.
```

Después generamos un nuevo código, esta vez generamos family.pl.

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

male(jim).
male(bob).
male(tom).
male(pete).

parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).
parent(pete, 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.
grandparent(X, Z):- parent(X, Y), parent(Y, Z).
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).
```

```
1 ?- grandfather(X, ann).
X = tom ;
false.
```

Tercera sesión

Durante la tercera sesión revisamos algunas de las aplicaciones con las que cuenta Prolog.

El problema de las Torres de Hanoi

El Problema de las Torres de Hanói es un rompecabezas clásico que implica mover N discos de una torre de origen a una de destino, utilizando una torre intermedia como auxiliar. Para su resolución, se deben cumplir dos condiciones fundamentales:

- No se puede colocar un disco más grande sobre uno más pequeño.
- Solo se puede mover un disco a la vez.

El predicado `move(N,X,Y,Z)` modela el movimiento de N discos desde la torre de origen X a la torre de destino Y, con Z siendo la torre auxiliar. La primera regla, `move(1,X,Y,_)`, constituye el caso base: si solo hay un disco (N=1), este se mueve directamente del origen X al destino Y, y se imprime un mensaje indicando la acción. La segunda regla, `move(N,X,Y,Z)`, maneja el caso recursivo para más de un disco. Primero, se mueven N−1 discos de X a la torre auxiliar Z (usando Y como auxiliar). Luego, el disco más grande (el N-ésimo) se mueve de X a Y. Finalmente, los N−1 discos restantes se trasladan de la torre auxiliar Z al destino Y.

Listas enlazadas

En Prolog, las listas enlazadas pueden representarse de manera flexible. La lista más elemental es nil (nula), que denota una lista vacía. Cualquier otra lista contendrá nil como el "siguiente" nodo al final. En la terminología de listas, el primer elemento se conoce como la cabeza de la lista, y el resto se denomina la cola. Por ejemplo, en la lista `node(2, node(5, node(6, nil)))`, la cabeza es 2 y la cola es `node(5, node(6, nil))`.

Los predicados presentados demuestran la manipulación básica de listas enlazadas en Prolog. Una lista enlazada se representa como node(Head, Tail), donde Head es el elemento actual y Tail es el resto de la lista.

- `add_front(L,E,NList)`: Este predicado, que no utiliza recursión, tiene como objetivo añadir un elemento E al inicio de una lista enlazada existente L. El resultado es una NList (nueva lista) donde E se convierte en la nueva cabeza y L en la cola. Es decir, NList se unifica con node(E,L).
- `add_back(nil, E, NList)`: Esta es la primera regla para añadir un elemento E al final de una lista. Constituye el caso base: si la lista de entrada es nil (vacía), el nuevo elemento E se convierte en el único nodo de la lista, formando node(E,nil).
- `add_back(node(Head,Tail), E, NList)`: Esta es la regla recursiva. Si la lista no está vacía, se invoca add_back recursivamente con la cola de la lista actual (Tail) para localizar el final. Una vez que add_back(Tail, E, NewTail) unifica NewTail con la cola actualizada (con E añadido al final), el predicado construye NList como node(Head,NewTail), manteniendo la cabeza original y adjuntando la cola ya modificada. add_front(L,E,NList) :- NList = node(E,L).