Paradigma Funcional:
Paradigma **declarativo** que se basa en un modelo matemático de composición de funciones. En este modelo, el resultado de un cálculo es la entrada del siguiente, y así sucesivamente hasta que una composición produce el valor deseado, tal como sucede en la composición de funciones matemáticas.

Expresan mejor **QUE** hay que calcular. **No especifican** tanto **COMO** realizar el mencionado cálculo a diferencia de lo que ocurre en los programas imperativos.
LOS CÁLCULOS SE VEN COMO UNA FUNCIÓN MATEMÁTICA QUE HACEN CORRESPONDER ENTRADAS Y SALIDAS.

Todas las funciones tienen transparencia referencial. Esto es el resultado devuelto por una función sólo depende de los argumentos que se le pasen en esa llamada.
La repetición se modela utilizando la recursividad ya que no existe el valor de una variable.

Existen valores intermedios que son el resultado de cómputos intermedios, los cuales resultan en valores útiles para los cálculos subsiguientes.

Características:
Transparencia referencial: Permite que el valor que devuelve una función está únicamente determinado por el valor de sus argumentos consiguiendo que una misma expresión tenga siempre el mismo valor. De esta manera, los valores resultantes son inmutables. No existe el concepto de cambio de estado.

Utilización de tipos de datos genéricos: Permite aumentar su flexibilidad y realizar unidades de software genéricas, es una forma de implementar el polimorfismo en el paradigma funcional.

Recursividad: Basada en el principio matemático de inducción, que se ve expresada en el uso de tipos de datos recursivos, como las listas y funciones recursivas que las operan.

Tratar funciones como datos: Mediante la definición de funciones de orden superior, que permite un gran nivel de abstracción y generalidad en las soluciones.

Los lenguajes funcionales relevan al programa de la compleja tarea de gestión de memoria.

La manera de construir abstracciones es a través de funciones.

Ventajas:
1) Fácil de formular matemáticamente.
2) Administración automática de la memoria.
3) Simplicidad en el código.
4) Rapidez en la codificación de los programas.

Desventajas:
1) No es fácilmente escalable.
2) Difícil de integrar con otras aplicaciones.

3) No es recomendable para modelar lógica de negocios o para realizar tareas transaccionales.

Áreas de aplicación:
1) Demostraciones de teoremas: Por su naturaleza “funcional” este paradigma es útil en la demostración automática de teoremas, ya que permite especificar de manera adecuada y precisa problemas matemáticos.
2) Creación de compiladores, analizadores sintácticos: Su naturaleza inherentemente recursiva le permite modelar adecuadamente estos problemas.
3) Resolver problemas que requieran demostraciones por inducciones.
4) En la industria se puede usar para resolver problemas matemáticos complejos.
5) Se utiliza en centros de investigaciones y en universidades.

Racket no es un superconjunto ni un subconjunto de Scheme

Función: Regla de asociación que relaciona dos o más conjuntos entre sí.

Una función f entre dos conjuntos A y B, es una correspondencia que a cada elemento de un subconjunto de A, llamado “Dominio de f”, le hace corresponder uno y sólo uno de un subconjunto B llamado “Imagen de f”.

f es una función de A en B, o f es una función que toma elementos del dominio A y los aplica sobre otro llamado imagen B.

- La definición de funciones es una estructura jerárquica, en la cual las funciones más simples aparecen en la definición de las funciones más complejas.
- Esta sucesión se clausura con funciones primitivas
- Básicamente se aplican los conceptos del diseño modular, en el cual un problema complejo se descompone en problemas más simples.
- Cada uno de ellos es un problema primitivo, o bien, deberá
- volver a descomponerse.

Las funciones se denominan expresiones y se pueden representar con la siguiente sintaxis:

```
<expresión> ::=
                <variable> |
                <constante> |
                (<variable>, ... ,<variable>)
                <expresión> | <expresión>
                (<expresión>, ... ,<expresión>)
```

Todas las expresiones denotan un valor, pueden ser expresiones que no admiten ser aplicadas a ningún argumento, ya que están totalmente evaluadas o expresiones que pueden ser aplicadas a un cierto número de argumentos para ser totalmente evaluadas.

Las formas permitidas por la gramática son:

A. **expresiones atómicas**, que pueden ser <variable> y <constante>
B. **expresiones compuestas**
   abstracciones funcionales: (<variable>, . . .,<variable>) <expresión>
   aplicaciones funcionales: <expresión> (<expresión>, ... ,<expresión>)

Funciones de orden superior:

Es una función tal que alguno de sus argumentos es una función o que devuelve una función como resultado.
● Son útiles porque permiten capturar esquemas de cómputo generales (abstracción).
● Son más útiles que las funciones normales (parte del comportamiento se especifica al usarlas).
Podemos destacar que los argumentos y resultados de la aplicación de funciones pueden ser a su vez funciones, las cuales pueden ser también aplicadas a otros argumentos.
Este tipo de función es denominado FUNCIÓN DE ORDEN SUPERIOR denotando el hecho que sus argumentos y resultados pueden ser funciones.

Lambda-Calculo: Es un sistema formal diseñado para investigar la definición de función, la noción de aplicación de funciones y la recursión. Es el más pequeño lenguaje universal de programación. Consiste de una regla de transformación simple (substitución de variables) y un esquema simple para definir funciones.
El cálculo lambda es universal porque cualquier función computable puede ser expresada y evaluada a través de él.

Evaluación Perezosa (Lazy):
Los lenguajes tradicionales, evalúan todos los argumentos de una función antes de conocer si éstos serán utilizados. Dicha técnica de evaluación se conoce como evaluación ansiosa (eager evaluation) porque evalúa todos los argumentos de una función antes de conocer si son necesarios.
Por otra parte, en ciertos lenguajes funcionales se utiliza evaluación perezosa (lazy evaluation) que consiste en no evaluar un argumento hasta que no se necesit**a.**
(Scheme es estricto)

Racket - Datos atómicos: tipos de datos elementales que generalmente se consideran entidades indivisibles

Listas:
car: retorna el primer elemento de la lista
cdr (could-er): retorna el resto de la lista
cons: construye listas. Recibe dos argumentos. Usualmente el segundo es una lista y en ese caso retorna una lista
construye pares. Cuando el segundo parámetro es una lista devuelve una lista Propia.

Lista Propia: Una lista vacía es una lista propia, y toda lista cuyo cdr sea una lista propia es una lista propia.
Listas impropias: compuestas por pares donde se marca las separación de elementos por puntos

Asignaciones
Let*: permite realizar asignaciones secuenciales, donde la definición de las variables internas pueden ver a las variables externas.
(let* ((x (* 5.0 5.0))
(y (- x (* 4.0 4.0))))
(sqrt y)) ⇒ 3.0

Recursividad
la recursividad se produce cuando un procedimiento se llama a si mismo

¿es posible hacer un let-bound recursivo?

```
(let ((sum (lambda (ls)
        (if (null? ls)
        0
        (+ (car ls) (sum (cdr ls)))))))
(sum '(1 2 3 4 5)))
```

NO!! Sum solo existe en el cuerpo del LET y no en la definición de las variables

letrec: al igual que let permite definir un conjunto de pares variable-valor y un conjunto de sentencias que las referencian.
A diferencia de let, las variables son vistas en la cabecera también.

```
(letrec ((sum (lambda (ls)
        (if (null? ls)
        0
        (+ (car ls) (sum (cdr ls)))))))
(sum '(1 2 3 4 5)))
```

Vectores: Secuencia de objetos separados por un blanco y precedidos por un # o con la siguiente sintaxis:

- #(a b c) → vector de elementos a, b y c
- (vector) ⇒ #()
- (vector 'a 'b 'c) ⇒ #(a b c)

(make-vector n obj**):** Retorna un vector de n posiciones. Si se provee “obj” se llenaran las posiciones on obj, en caso contrario permanecerán como indefinido.

(vector-length vector): Retorna la cantidad de elementos de un vector.

(vector-ref vector n): Retorna la enésima posición de un vector

(vector-set! vector n obj): Establece el valor de la enésima posición del vector a obj.

```
let ((v (vector 'a 'b 'c 'd 'e)))
        (vector-set! v 2 'x)
        v)
        ⇒ #(a b x d e)
```

(vector-fill! vector obj): reemplaza cada elemento del vector obj

(vector->list vector): devuelve una lista a partir de un vector

(list->vector list): convierte una lista en vector

MAP: Aplica el procedimiento a cada elemento de la lista y devuelve una lista con los resultados

```
map (lambda (x) (+ x 2)) '(1 2 3))
→ (3 4 5)
```

También es posible tener múltiples argumentos

```
map cons '(1 2 3) '(10 20 30))
→ ((1 . 10) (2 . 20) (3 . 30))
```

Paradigma lógico:

Programación Imperativa vs Declarativa

Programación Imperativa:

- Programar es codificar el **cómo**
- Términos "de máquina"
- Traducir conocimientos a la máquina
- Puede dificultarse la descripción de algunos problemas

Programación Declarativa:

- Programar es describir el **qué**
- Términos "humanos"
- Se expresa y organiza el conocimiento
- Se ejecuta lo expresado

Programación lógica: Enfocado en el razonamiento lógico y la inferencia automática.

Campos de aplicación:
Sistemas Expertos: emulan el comportamiento de un experto humano en un dominio específico de conocimiento.
Ejemplo: diagnóstico de enfermedades basándose en reglas lógicas y conocimiento experto

Demostración Automática de Teoremas: verificación formal de programas informáticos para demostrar propiedades de seguridad, corrección y rendimiento.
Ejemplo: verificación de software (verificar matemáticamente su comportamiento en diferentes condiciones)

Inteligencia Artificial: puede considerarse parte de la inteligencia artificial debido a su capacidad para representar conocimiento, razonar sobre este conocimiento y llegar a conclusiones lógicas.
Ejemplo: aplicación en Procesamiento del Lenguaje Natural, con un sistema que comprenda y responda preguntas en lenguaje natural, facilitando la interacción con los usuarios

Programa:
- Se compone de conocimientos expresados como axiomas lógicos.
- Para ejecutar un programa, se provee un nuevo axioma que el sistema intentará probar usando los axiomas existentes

Ejecución; aplicando reglas y axiomas previamente definidos

Formalización en Lógica de Predicados: ejemplo

- Toda madre ama a sus hijos
- María es madre y Juan es hijo de María
- María ama a Juan

$$\forall x(\text{Madre}(x) \rightarrow \text{Ama}(x, \text{Hijo}(x)))$$

$$\text{Madre}(\text{María}) \land \text{Hijo}(\text{Juan}, \text{María})$$

$$\text{Ama}(\text{María}, \text{Juan})$$

Fórmulas atómicas: Representan afirmaciones simples o relaciones entre entidades del dominio del problema.
Tienen la forma p(t1…tn) donde p es un predicado y ti son términos.
Son la base de las expresiones logicas en prolog.

Orden de Precedencia de los Conectores Lógicos

Cómo se evalúan las fórmulas complejas en lógica

$$\neg,\ \forall,\ \exists,\ \vee,\ \wedge,\ \rightarrow,\ \leftrightarrow$$

Ejemplo:

$$\neg(\forall x(P(x) \rightarrow Q(x)))$$

se evalúa como

$$\neg(\forall x(P(x) \rightarrow Q(x)))$$

Y no como:

$$(\forall x(\neg(P(x) \rightarrow Q(x))))$$

Cláusulas definidas: Definen las relaciones entre los objetos del dominio del problema.
Permiten expresar reglas lógicas y relaciones entre entidades del mundo del problema.
La **cabecera** identifica la relación que se establece, mientras que el **cuerpo** especifica las condiciones bajo las cuales esa relación es válida.

Ejemplo:

padre(tomás, juan) ← verdadero
Establece que Tomás es padre de Juan.

nieto(X, Z) ← hijo(X, Y) ∧ padre(Y, Z)
Define que X es nieto de Z si X es hijo de Y y Y es padre de Z.

PROLOG: Un programa en Prolog está compuesto por un conjunto finito de cláusulas definidas, que pueden ser hechos o reglas.
Describe el modelo previsto para resolver un problema específico, utilizando hechos para representar información concreta y reglas para establecer relaciones entre los elementos del dominio del problema

```
% Hechos: Representan información concreta
padre(juan, pedro).
madre(maría, ana).
abuelo(pedro, juan).

% Reglas: Establecen relaciones entre los hechos
hermano(X, Y) :- padre(Z, X), padre(Z, Y), X \= Y.
```

Objetivos definidos: Consultas que el usuario formula al programa para obtener información específica sobre el dominio del problema.
Se escriben en forma de fórmulas lógicas y pueden contener variables que deben ser instanciadas por el programa.

Para el programa:

```
% Hechos: Representan información concreta
padre(juan, pedro).
madre(maría, ana).
abuelo(pedro, juan).

% Reglas: Establecen relaciones entre los hechos
hermano(X, Y) :- padre(Z, X), padre(Z, Y), X \= Y.
```

hermano(X, juan) madre(X, ana) abuelo(X,juan)

Elementos:
Base de conocimiento: Representado por un conjunto de afirmaciones (hechos y reglas) representando los conocimientos que poseemos en un determinado dominio de campo.

Motor de inferencia Es el que se encarga de la "ejecución". Que en esencia es un comprobador de teoremas, el cual utiliza la regla de inferencia (Proceso de deducir conclusiones lógicas a partir de premisas o proposiciones dadas)

Resolución (intérprete de comandos) cuyo objetivo es permitir la posibilidad de responder consultas.

El hecho de programar en **Prolog consiste en brindar a la computadora un universo finito en forma de hechos y reglas**, proporcionando los medios para realizar inferencias de un hecho a otro. A continuación si se hacen las preguntas adecuadas, Prolog buscará la respuesta en dicho universo y las presentará en pantalla.

Es un programa **conformado por un conjunto de hechos y reglas** que representan el problema que se pretende resolver. Ante una determinada pregunta sobre el problema, el Prolog utilizará estos hechos y reglas para intentar **demostrar la veracidad o falsedad de la pregunt**a que se le ha planteado

LÓGICA + CONTROL = PROGRAM

Los pasos a seguir para escribir un programa en Prolog:
a) Declarar **HECHOS** sobre los objetos y relaciones. Un hecho expresa una relación entre objetos
b) Definir **REGLAS** sobre los objetos y relaciones. Las reglas se utilizan para significar que un hecho depende de uno o más hechos
c) Hacer **PREGUNTAS** sobre los objetos y relaciones. Las preguntas son las herramientas que tenemos para recuperar información, al hacer una pregunta a un programa lógico queremos determinar si es consecuencia lógica del programa.

Cuando se hace una pregunta a Prolog, éste efectuará una búsqueda por toda la Base de Conocimiento intentando encontrar hechos que coincidan con la pregunta.

Principio de resolución / Regla de inferencia: Algoritmo , que a partir de la negación del a pregunta y los hechos y reglas del programa, intenta llegar al absurdo para demostrar que la pregunta es cierta.
La implementación de la Regla de Inferencia en Prolog se basa en los conceptos de Unificación y Backtracking

Unificación:

Mecanismo mediante el cual las variables lógicas toman valor en Prolog.
Cuando una variable no tiene valor se dice que está libre. Pero una vez que se le asigna valor, este ya no cambia, por eso se dice que la variable está ligada.
Se dice que **dos términos unifican cuando existe una posible ligadura**(asignación de valor) de las variables,tal que ambos términos son idénticos sustituyendo las variables por dichos valores
Ej: a(X,3) y a(4,Z) unifican dando valores a las variables: X vale 4, Z vale 3.
La unificación no debe confundirse con la asignación de los lenguajes imperativos puesto que representa la igualdad lógica.

Para entender la unificación en Prolog, es importante conocer algunos conceptos básicos:
Functor: Es el nombre de una estructura compuesta. Ej: en “padre(juan,pedro)” padre es el functor.
Aridad: Es el número de argumentos que una estructura compuesta tiene, en el ejemplo anterior la aridad es 2.

Para saber si dos términos unifican podemos aplicar las siguientes normas:

1 - **Una variable siempre unifica con un término, quedando ésta ligada a dicho término.**

```
?- X = 5.
 X = 5.
```

2 - **Dos variables siempre unifican entre sí, y cuando una de ellas se liga a un término, todas las que unifican se ligan a dicho término.**

```
?- X = Y, Y = 5.
X = 5,
Y = 5.
```

3- **Para que dos términos unifiquen, deben tener el mismo functor y la misma aridad. Luego, se comprueba que los argumentos unifican uno a uno, manteniendo las ligaduras que se produzcan en cada uno.**
Esto significa que:

- Los dos términos deben tener el mismo nombre de functor.
- Los dos términos deben tener el mismo número de argumentos.
- Los argumentos de ambos términos deben unificar recursivamente.

Ejemplos:

Mismo functor y aridad:

?- padre(juan, pedro) = padre(X, Y).

X = juan,

Y = pedro.

Aquí, `padre` es el functor, y ambos términos tienen aridad 2. Los argumentos `juan` y `pedro` unifican con `X` y `Y`, respectivamente.

Diferente functor: (No unifican)

?- padre(juan, pedro) = madre(X, Y).
false.

Diferente aridad:(No unifican)

?- padre(juan) = padre(X, Y).
false.

Unificación de argumentos:

?- padre(juan, pedro) = padre(juan, X).
X = pedro.

Búsqueda de soluciones

Una llamada concreta a un predicado, con unos argumentos concretos, se denomina objetivo (en inglés, goal). Todos los objetivos tienen un resultado de éxito o fallo tras su ejecución indicando si el predicado es cierto para los argumentos dados, o por el contrario, es falso.
Cuando un objetivo tiene éxito las variables libres que aparecen en los argumentos pueden quedar ligadas. Estos son los valores que hacen cierto el predicado. Si el predicado falla, no ocurren ligaduras en las variables.

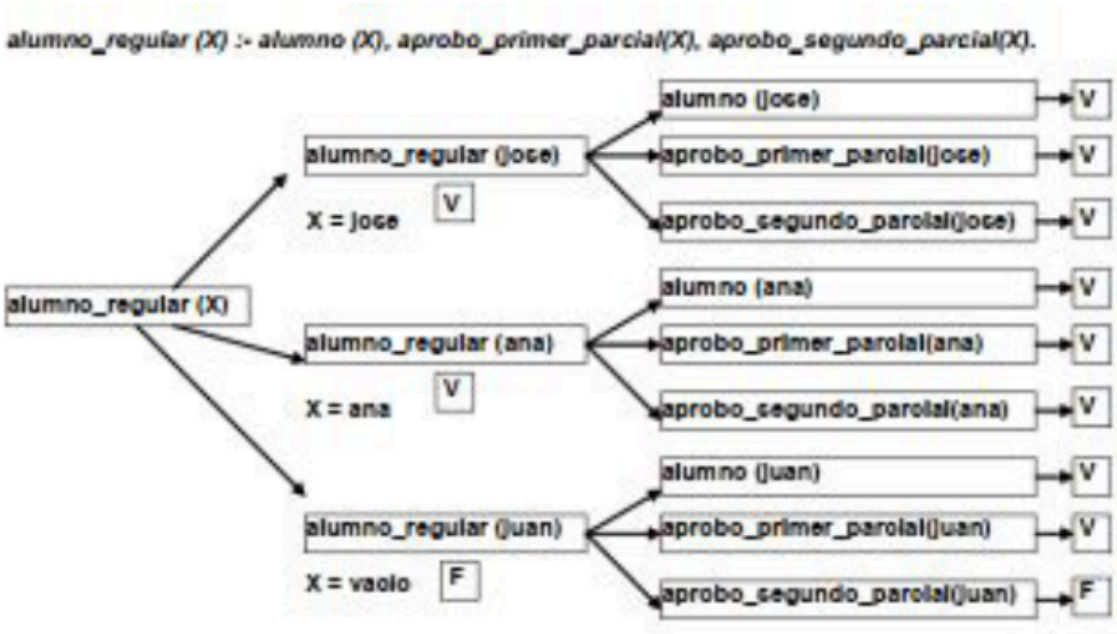
Las secuencias de objetivos o consultas tienen las siguientes características:

- Los objetivos se ejecutan secuencialmente por orden de escritura (es decir, de izquierda a derecha).
- Si un objetivo falla, los siguientes objetivos ya no se ejecutan.Además la conjunción,en total, falla.
- Si un objetivo tiene éxito,algunas o todas sus variables quedan ligadas,y por lo tanto,dejan de ser variables libres para el resto de objetivos en la secuencia.
- Si todos los objetivos tienen éxito, la conjunción tiene éxito y mantiene las ligaduras de los objetivos que la componen.
- Partiendo de un objetivo a probar se busca las aserciones que pueden probar el objetivo. Este proceso de búsqueda de soluciones,se basa en dos conceptos: la Unificación y el Backtracking.

- En el proceso de Unificación cada objetivo determina un subconjunto de cláusulas susceptibles de ser ejecutadas (puntos de elección). Prolog selecciona el primer punto de elección y sigue ejecutando el programa hasta determinar si el objetivo es verdadero o falso.

BACKTRAKING:

Cada solución es el resultado de una secuencia de decisiones.



Funcionamiento:

- Cuando se va ejecutar un objetivo, Prolog sabe de antemano cuántas soluciones alternativas puede tener.
- Cada una de las alternativas se denomina punto de elección.Dichos puntos de elección se anotan internamente y de forma ordenada. Para ser exactos, se introducen en una pila.
- Se escoge el primer punto de elección y ejecuta el objetivo eliminando punto de elección en el proceso.
- Si el objetivo tiene éxito se continúa con el siguiente objetivo aplicando estas mismas normas.
- Si el objetivo falla, Prolog da marcha atrás recorriendo los objetivos que anteriormente sí tuvieron éxito (en orden inverso) y deshaciendo las ligaduras de sus variables. Es decir, comienza el backtracking.
- Cuando uno de esos objetivos tiene un punto de elección anotado, se detiene el backtracking y se ejecuta de nuevo dicho objetivo usando la solución alternativa. Las variables se ligan a la nueva solución y la ejecución continúa de nuevo hacia adelante. El punto de elección se elimina en el proceso.
- El proceso se repite mientras haya objetivos y puntos de elección anotados. De hecho, se puede decir que un programa Prolog ha terminado su ejecución cuando no le quedan puntos de elección anotados ni objetivos por ejecutar en la secuencia

El Backtracking se puede controlar mediante el uso de dos predicados:
El corte (!) El fail

Predicado de corte:
Es un predicado predefinido que no recibe argumentos. Se representa mediante un signo de admiración (!). Es un predicado que siempre se cumple, que genera un resultado verdadero en la primera ejecución, y falla en el proceso de backtracking, impidiendo dicho retroceso.

El corte tiene la propiedad de eliminar los puntos de elección del predicado que lo contiene. Es decir, cuando se ejecuta el corte, el resultado del objetivo (no sólo la cláusula en cuestión) queda comprometido al éxito o fallo de objetivos que aparecen a continuación. Es como si a Prolog "se le olvidase" que dicho objetivo puede tener varias soluciones.

Otra forma de ver el efecto del corte es pensar que solamente tiene la propiedad de detener el backtracking cuando éste se produce. Es decir, en la ejecución normal el corte no hace nada. Pero cuando el programa entra en backtracking y los objetivos se recorren marcha atrás,al llegar corte el backtracking se detiene repentinamente forzando el fallo del objetivo.

Se usa para:

- Para optimizar la ejecución
- Para facilitar la legibilidad y comprensión del algoritmo que está siendo programado
- Para implementar algoritmos diferentes según la combinación de argumentos de entrada Para conseguir que un predicado solamente tenga una solución.

Ejemplo 1: Controlando el backtracking

```
signo(X, positivo) :- X > 0, !.
signo(X, negativo) :- X < 0, !.
signo(0, cero).
```

Si `X` es mayor que `0`, Prolog unificará con `signo(X, positivo)` y encontrará el corte `!`.
Esto significa que no buscará ninguna otra solución para `X > 0`.
Si `X` es menor que `0`, Prolog unificará con `signo(X, negativo)` y encontrará el corte `!`.
Esto significa que no buscará ninguna otra solución para `X < 0`.
Solo si `X` es `0`, se unificará con `signo(0, cero)`

Ejemplo 2: Uso del Corte para Optimizar Búsqueda

```
triangulo(A, B, C, equilatero) :-
        A = B, B = C, !.
triangulo(A, B, C, isosceles) :-
        (A = B; B = C; A = C), !.
triangulo(_, _, _, escaleno).
```

Si los tres lados son iguales (`A = B, B = C`), Prolog determina que el triángulo es equilátero y no buscará más soluciones.
Si dos lados son iguales (`A = B; B = C; A = C`), Prolog determina que el triángulo es isósceles y no buscará más soluciones.
Solo si ninguna de las condiciones anteriores se cumple, Prolog determina que el triángulo es escaleno.

Predicado de fallo: Es un predicado predefinido, sin argumentos que siempre falla, por lo tanto, implica la realización del proceso de retroceso (backtracking) para que se generen nuevas soluciones.

Cuando la máquina Prolog encuentra una solución se detiene y devuelve el resultado de la ejecución. Con fail podemos forzar a que no se detenga y siga construyendo el árbol de búsqueda hasta que no queden más soluciones que mostrar.

Ejemplo 1: Búsqueda de Soluciones Alternativas

```
imprimir_elemento(X) :-
        write(X), nl, fail.
imprimir_elementos([]).
imprimir_elementos([X|Xs]) :-
        imprimir_elemento(X),
        imprimir_elementos(Xs).
```

Aquí, `imprimir_elemento(X)` escribe `X` y luego falla, forzando el backtracking para que `imprimir_elementos([X|Xs])` pueda continuar con el resto de la lista. El `fail` asegura que todos los elementos sean impresos.

Ejemplo 2: Encontrar Todas las Soluciones:

```
persona(juan).
persona(maria).
persona(carlos).

todas_las_personas :-
        persona(X),
        write(X), nl,
        fail.
todas_las_personas.
```

Aquí, `todas_las_personas` escribe todas las personas definidas en la base de datos. El `fail` después de `write(X)` fuerza el backtracking para que todas las soluciones sean consideradas

Recursividad: Definir relaciones en términos de ellas mismas. Clausula con llamado recursivo, es decir, “se llama a sí misma” para **verificar** que se cumple esa misma propiedad como parte de la condición que define a la regla,y sobre algún posible valor de sus variables.