EBook Gratis

APRENDIZAJE

Prolog Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#prolog

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: prolog-language

It is an unofficial and free Prolog Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Prolog Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Prolog Language

Observaciones

Implementaciones

1. SWI-Prolog (gratis) swi-prolog
    - Implementado en c
2. SICStus (comercial) sicstus-prolog
3. YAP (gratis) yap
4. GNU Prolog (gratis) gnu-prolog
5. XSB (gratis) xsb
6. B (comercial) b-prolog
7. IF (comercial)
8. Ciao (gratis)
9. Minerva (comercial)
10. ECLiPSe-CLP (gratis) eclipse-clp
11. Jekejeke Prolog (comercial)
12. Prolog IV
13. Prólogo de rendimiento (gratis)
    - Implementado en c # , javascript y phyton.
14. Prólogo visual (comercial) visual-prólogo

Examples

Instalación o configuración

SWI-Prolog

Windows y Mac:

- Descarga SWI-Prolog en la web oficial
- Simplemente instale siguiendo las instrucciones del instalador.

Linux (PPA):

- Agregue el PPA `ppa:swi-prolog/stable` a las fuentes de software de su sistema (los desarrolladores pueden elegir para `ppa:swi-prolog/devel`):
    - Abra una terminal (Ctrl + Alt + T) y escriba: `sudo add-apt-repository ppa:swi-prolog/stable`
    - Después, actualice la información del paquete: `sudo apt-get update`
- Ahora instale SWI-Prolog a través del administrador de paquetes: `sudo apt-get install swi-prolog`

- Ahora puede iniciar SWI-Prolog a través de la línea de comandos con el comando `swipl`

añadir / 3

```
append([], Bs, Bs).
append([A|As], Bs, [A|Cs]) :-
    append(As, Bs, Cs).
```

`append/3` es una de las relaciones Prolog más conocidas. Define una relación entre tres argumentos y es verdadera *si* el tercer argumento es una lista que denota la concatenación de las listas que se especifican en el primer y segundo argumento.

Notablemente, y como es típico de un buen código de Prolog, el `append/3` se puede usar en *varias direcciones* : Se puede usar para:

- *Adjuntar* dos listas total o parcialmente instanciadas:

  ```
  ?- A = [1, 2, 3], B=[4, 5, 6], append(A, B, Y)
  Output:
  A = [1, 2, 3],
  B = [4, 5, 6],
  Y = [1, 2, 3, 4, 5, 6].
  ```

- *verifique* si la relación es verdadera para tres listas totalmente instanciadas:

  ```
  ?- A = [1, 2, 3], B = [4, 5], C = [1, 2, 3, 4, 5, 6], append(A, B, C)
  Output:
  false
  ```

- *generar* todas las formas posibles de anexar dos listas a una lista dada:

  ```
  ?- append(A, B, [1, 2, 3, 4]).
  Output:
  A = [],
  B = [1, 2, 3, 4] ;
  A = [1],
  B = [2, 3, 4] ;
  A = [1, 2],
  B = [3, 4] ;
  A = [1, 2, 3],
  B = [4] ;
  A = [1, 2, 3, 4],
  B = [] ;
  false.
  ```

Restricciones de CLP (FD)

Todas las implementaciones serias de Prolog proporcionan **restricciones CLP (FD)** . Nos permiten razonar sobre los **enteros** de una manera pura.

```
?- X #= 1 + 2.
X = 3.
```

```
?- 5 #= Y + 2.
Y = 3.
```

Programación de base de datos

Prolog categoriza todo en:

- **Átomos** : cualquier secuencia de caracteres que no comienza con un alfabeto en mayúsculas. Por ejemplo, `a` , `b` , `okay`
- **Números** : no hay una sintaxis especial para los números, no se requiere declaración. Ej. `1` , `22` , `35.8`
- **Variables** : una cadena que comienza con un carácter en mayúscula o un guión bajo (_). Ej. `X` , `Y` , `Abc` , `AA`
- **Términos complejos** : están hechos de un *funtor* y una secuencia de *argumentos* . El nombre de un término complejo siempre es un átomo, mientras que los argumentos pueden ser átomos o variables. Ej. `father(john,doe)` , `relative(a)` , `mother(X,Y)` .

Una base de datos lógica contiene un conjunto de *hechos* y *reglas* .

Un término complejo con solo átomos como argumentos se llama un hecho, mientras que un término complejo con variables como argumentos se llama una regla.

Ejemplo de hechos en Prolog:

```
father_child(fred, susan).
mother_child(hillary, joe).
```

Ejemplo de una regla en Prolog:

```
child_of(X,Y):-
    father_child(Y,X)
    ;
    mother_child(Y,X).
```

Tenga en cuenta que el `;` Aquí es como el operador `or` en otros idiomas.

Prolog es un lenguaje declarativo y puede leer esta base de datos de la siguiente manera:

> *fred es el padre de susan*
>
> *hillary es la madre de joe.*
>
> *Para todas las `X` e `Y` , `X` es un hijo de `Y` si `Y` es un padre de `X` o `Y` es una madre de `X`*

De hecho, un conjunto finito de hechos y / o reglas constituye un *programa* lógico.

El uso de tal programa se demuestra haciendo *consultas* . Las consultas le permiten recuperar información de un programa lógico.

Para cargar la base de datos en el intérprete (asumiendo que ha guardado la base de datos en el directorio en el que está ejecutando el intérprete) simplemente ingrese:

```
?- [nameofdatabase].
```

reemplazando el `nameofdatabase` la base de `nameofdatabase` con el nombre del archivo real (tenga en cuenta que aquí excluimos la extensión `.pl` del nombre de archivo).

Ejemplo de consultas en el intérprete para el programa anterior y los resultados:

```
?- child_of(susan,fred).
true

?- child_of(joe,hillary).
true

?- child_of(fred,susan).
false

?- child_of(susan,hillary).
false

?- child_of(susan,X).
X = fred

?- child_of(X,Y).
X = susan,
Y = fred ;
X = joe,
Y = hillary.
```

Las consultas anteriores y sus respuestas se pueden leer de la siguiente manera:

> *¿Es Susan un hijo de Fred? - cierto*
>
> *¿Es Joe un hijo de Hillary? - cierto*
>
> *¿Es Fred un hijo de Susan? falso*
>
> *¿Es Susan un hijo de Hillary? falso*
>
> *quien es susan un hijo de fred*

Así es como programamos la lógica en Prolog. Un programa lógico es más formal: un conjunto de axiomas, o reglas, que definen relaciones (también conocido como predicados) entre objetos. Una forma alternativa de interpretar la base de datos anterior de una manera lógica más formal es:

> *La relación `father_child` mantiene entre fred y susan.*
>
> *La relación `mother_child` mantiene entre hillary y joe.*
>
> *Para todas las `X` e `Y` la relación `child_of` mantiene entre `X` e `Y` si la relación `father_child` mantiene entre `Y` y `X`, o la relación `mother_child` mantiene entre `Y` y `X`*

Hola, Mundo en el intérprete interactivo.

Para imprimir "¡Hola mundo!" en el intérprete de Prolog (aquí estamos usando `swipl` , el shell para SWI Prolog):

```
$ swipl
<...banner...>
?- write('Hello, World!'), nl.
```

`?-` es el indicador del sistema: indica que el sistema está listo para que el usuario ingrese una secuencia de *objetivos* (es decir, una *consulta*) que debe terminarse con un `.` (parada completa).

Aquí la consulta de `write('Hello World!'), nl` tiene dos objetivos:

- `write('Hello World!')` : `'Hello World!'` tiene que ser mostrado **y** (`,`)
- una nueva línea (`nl`) debe seguir.

`write/1` (el `/1` se usa para indicar que el predicado toma un argumento) y `nl/0` son *predicados incorporados* (la definición se proporciona de antemano en el sistema Prolog). Los predicados incorporados proporcionan recursos que no se pueden obtener mediante la definición pura de Prolog o para evitar que el programador tenga que definirlos.

La salida:

> ¡Hola Mundo!
>
> sí

termina con `yes` lo que significa que la consulta ha tenido éxito. En algunos sistemas se imprime `true` lugar de `yes` .

Hola mundo de un archivo

Abra un nuevo archivo llamado `hello_world.pl` e inserte el siguiente texto:

```
:- initialization hello_world, halt.

hello_world :-
    write('Hello, World!'), nl.
```

La directiva de `initialization` especifica que el objetivo `hello_world, halt` debe llamarse cuando se carga el archivo. `halt` salidas del programa.

Este archivo puede ser ejecutado por su ejecutable Prolog. Las banderas exactas dependen del sistema Prolog. Si está utilizando SWI Prolog:

```
$ swipl -q -l hello_world.pl
```

Esto producirá salida `Hello, World!` . El indicador `-q` suprime el banner que normalmente se muestra cuando se llama a ejecutar `swipl` . El `-l` especifica un archivo para cargar.

Lea Empezando con Prolog Language en línea:
https://riptutorial.com/es/prolog/topic/1038/empezando-con-prolog-language

Capítulo 2: Actuación

Examples

Maquina abstracta

Por eficiencia, el código de Prolog se compila generalmente para **abstraer el código de la máquina** antes de ejecutarse.

Se han propuesto muchas arquitecturas y variantes de máquinas abstractas diferentes para la ejecución eficiente de los programas Prolog. Éstos incluyen:

- **WAM** , la *máquina abstracta de Warren*
- **TOAM** , una máquina abstracta utilizada en B-Prolog.
- **ZIP** , utilizado por ejemplo como base para la máquina virtual de SWI-Prolog
- **VAM** , una arquitectura de investigación desarrollada en Viena.

Indexación

Todos los intérpretes de Prolog ampliamente utilizados utilizan la **indexación de argumentos** para seleccionar eficazmente las cláusulas adecuadas.

Normalmente, los usuarios pueden confiar en al menos la *indexación del primer argumento* , lo que significa que las cláusulas se pueden distinguir de manera eficiente por el funtor y la aridad del término más externo del *primer* argumento. En las llamadas en las que ese argumento tiene una instancia suficiente, las cláusulas coincidentes pueden seleccionarse esencialmente en tiempo *constante* mediante el hash en ese argumento.

Más recientemente, la **indexación JIT** se ha implementado en más sistemas, lo que permite una indexación dinámica en cualquier argumento que tenga una instancia suficiente cuando se llama el predicado.

Optimización de llamadas de cola

Prácticamente todos los sistemas Prolog implementan **optimización de llamadas de cola** (TCO). Esto significa que las llamadas de predicado que están en una *posición de cola* pueden ejecutarse en *un* espacio de pila *constante* si el predicado es determinista.

La optimización de **recursión de** cola (TRO) es un caso especial de optimización de llamada de cola.

Lea Actuación en línea: https://riptutorial.com/es/prolog/topic/4205/actuacion

Capítulo 3: Árboles de derivación

Examples

Arbol de prueba

El árbol de prueba (también árbol de búsqueda o árbol de derivación) es un árbol que muestra la ejecución de un programa Prolog. Este árbol ayuda a visualizar el proceso de seguimiento cronológico presente en Prolog. La raíz del árbol representa la consulta inicial y las ramas se crean cuando se producen puntos de elección. Cada nodo en el árbol representa un objetivo. Las ramas solo se convierten en hojas cuando se probó verdadero / falso para el (los) objetivo (s) requerido (s) y la búsqueda en Prolog se realiza de una manera de **profundidad de izquierda a derecha** .

Considere el siguiente ejemplo:

```
% Facts
father_child(paul,chris).        % Paul is the father of Chris and Ellen
father_child(paul,ellen).
mother_child(ellen,angie).       % Ellen is the mother of Angie and Peter
mother_child(ellen,peter).


% Rules
grandfather_grandchild(X,Y) :-
    father_child(X,Z),
    father_child(Z,Y).

grandfather_grandchild(X,Y) :-
    father_child(X,Z),
    mother_child(Z,Y).
```

Cuando ahora consultamos:

```
?- grandfather_grandchild(paul,peter).
```

El siguiente árbol de pruebas visualiza el proceso de búsqueda primero en profundidad:

```
                                  ?- grandfather_grandchild(paul,peter).
                                      /                                \
                                     /                                  \
  ?- father_child(paul,Z1),father_child(Z1,peter).            ?-
father_child(paul,Z2),mother_child(Z2,peter).
             /                    \                                    /
\
      {Z1=chris}              {Z1=ellen}                        {Z2=chris}
{Z2=ellen}
           /                       \                                /
\
?- father_child(chris,peter).  ?- father_child(ellen,peter).  ?- mother_child(chris,peter). ?-
mother_child(ellen,peter).
```

```
          |                          |                                |
/               \
        fail                       fail                             fail
fail(*)           success
```

(*) falla para `mother_child(ellen,angie)` donde 'angie' no coincide con 'peter'

Lea Árboles de derivación en línea: https://riptutorial.com/es/prolog/topic/3097/arboles-de-derivacion

Capítulo 4: Estructuras de Control

Examples

Disyunción (OR lógica), implícita vs. explicta.

Prólogo intenta cláusulas alternativas para un predicado en el orden de aparición:

```
likes(alice, music).
likes(bob, hiking).

// Either alice likes music, or bob likes hiking will succeed.
```

El operador de disyunción (OR) `;` se puede utilizar para expresar esto en una regla:

```
likes(P,Q) :-
    ( P = alice , Q = music ) ; ( P = bob , Q = hiking ).
```

Los paréntesis son importantes aquí para mayor claridad. Ver esta cuestión en precedencia relativa para la combinación `,` y la disyunción `;` .

Conjunción (lógica AND)

La conjunción (AND lógico) está representada por la coma `,` operador (entre otros roles).

Conjunción entre cláusulas puede aparecer en una consulta:

```
?- X = 1, Y = 2.
```

La conjunción también puede aparecer entre las cláusulas de subgoal en el cuerpo de una regla:

```
triangleSides(X,Y,Z) :-
    X + Y > Z, X + Z > Y, Y + Z > X.
```

Cortar (eliminar puntos de elección)

A veces es conveniente evitar que Prolog retroceda a soluciones alternativas. La herramienta básica disponible para el programador para evitar que el prólogo continúe en su retroceso es el operador de corte. considera lo siguiente.

```
% (percent signs mean comments)
% a is the parent of b, c, and d.
parent(a,b).
parent(a,c).
parent(a,d).
```

Aquí el predicado `parent/2` tiene éxito más de una vez cuando

```
?- parent(a,X).
```

se llama. Para evitar que el prólogo busque más soluciones después de que se encuentre la primera, usaría el operador de corte, así.

```
?- parent(a,X), !.
```

Esto tendrá X igual a b (ya que es la primera solución posible) y no busque más soluciones.

Lea Estructuras de Control en línea: https://riptutorial.com/es/prolog/topic/4479/estructuras-de-control

Capítulo 5: Estructuras de datos

Examples

Liza

Las listas son un tipo especial de *término compuesto* . Las listas se definen inductivamente:

- el átomo `[]` es una lista, que denota la *lista vacía* .
- *si* `Ls` es una lista, entonces el término `'.'(L, Ls)` *también* es una lista.

Hay una sintaxis especial para denotar listas convenientemente en Prolog:

1. La lista `'.'(a, '.'(b, '.'(c, [])))` también se puede escribir como `[a,b,c]` .
2. El término `'.'(L, Ls)` también se puede escribir como `[L|Ls]` .

Estas notaciones se pueden combinar de cualquier manera. Por ejemplo, el término `[a,b|Ls]` es una lista *si* `Ls` es una lista.

Creando listas

Una lista que consta de literales unificados con la lista de variables:

```
?- List = [1,2,3,4].
List = [1, 2, 3, 4].
```

Construyendo una lista por medio de:

```
?- Tail = [2, 3, 4], List = [1|Tail].
Tail = [2, 3, 4],
List = [1, 2, 3, 4].
```

Construyendo una lista de valores desconocidos usando la `length/2` incorporada `length/2` :

```
?- length(List,5).
List = [_G496, _G499, _G502, _G505, _G508].
```

Como en Prolog todo es en esencia un término, las listas se comportan de forma heterogénea:

```
?- List = [1, 2>1, this, term(X), 7.3, a-A].
List = [1, 2>1, this, term(X), 7.3, a-A].
```

Esto significa que una lista también puede contener otras listas, también llamadas listas internas:

```
List = [[1,2],[3,[4]]].
```

Pares

Por convención, el functor `(-)/2` se usa a menudo para denotar **pares** de elementos en Prolog. Por ejemplo, el término `-(A, B)` denota el par de elementos `A` y `B` En Prolog, `(-)/2` se define como un *operador de infijo* . Por lo tanto, el término se puede escribir de manera equivalente como `AB` .

Muchos predicados comúnmente disponibles también usan esta sintaxis para denotar pares. Ejemplos de esto son `keysort/2` y `pairs_keys_values/3` .

Listas de asociaciones

En todos los sistemas Prolog serios, **las listas de asociación** están disponibles para permitir un acceso más rápido que lineal a una colección de elementos. Estas listas de asociaciones suelen basarse en *árboles equilibrados* como los **árboles AVL** . Existe una biblioteca de dominio público llamada `library(assoc)` que se envía con muchos sistemas Prolog y proporciona operaciones *O (log (N))* para insertar, recuperar y cambiar elementos a una colección.

Condiciones

En un nivel muy alto, Prolog solo tiene un tipo de datos único, denominado **término** . En Prolog, todos los datos están representados por términos Prolog. Los términos se definen inductivamente:

- un **átomo** es un término. Ejemplos de átomos son: `x` , `test` y `'quotes and space'` .
- una **variable** es un termino Las variables comienzan con una letra mayúscula o un guión bajo _ .
- Los enteros y los números de punto flotante son términos. Ejemplos: `42` y `42.42` .
- un **término compuesto** es un término, definido inductivamente de la siguiente manera: *Si* `T1` , `T2` , ..., `T_n` son términos, *entonces F* (`T1` , `T2` , ..., `T_n`) también es un término, donde *F* se denomina **functor** de El término compuesto.

Términos con campos nombrados usando biblioteca (registro)

La biblioteca `[record][1]` proporciona la capacidad de crear términos compuestos con campos nombrados. La directiva `:- record/1 <spec>` compila en una colección de predicados que inicializan, establecen y obtienen campos en el término definido por `<spec>` .

Por ejemplo, podemos definir un `point` estructura de datos con campos denominados `x` e `y` :

```
:- use_module(library(record)).

:- record point(x:integer=0,
                y:integer=0).

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

?- default_point(Point), point_x(Point, X), set_x_of_point(10, Point, Point1).
Point = point(0, 0),
X = 0,
Point1 = point(10, 0).

?- make_point([y(20)], Point).
Point = point(0, 20).
```

```
?-  is_point(X).
false.

?- is_point(point(_, _)).
false.

?- is_point(point(1, a)).
false.

?- is_point(point(1, 1)).
true.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
```

Lea Estructuras de datos en línea: https://riptutorial.com/es/prolog/topic/2417/estructuras-de-datos

Capítulo 6: Gramática de la cláusula definida (DCGs)

Examples

Cualquier cosa en absoluto: `... // 0`

Uno de los no terminales más elementales de DCG es `... //0` , que puede leerse como "cualquier cosa":

```
... --> [] | [_], ... .
```

Se puede usar para describir una lista `Ls` que contiene el elemento `E` través de:

```
phrase(( ..., [E], ... ), Ls)
```

Analizando con DCGs

Los DCG se pueden utilizar para analizar. Lo mejor de todo es que *el mismo* DCG se puede usar para analizar y *generar* listas que se describen. Por ejemplo:

```
sentence --> article, subject, verb, object.

article --> [the].

subject --> [woman] | [man].

verb --> [likes] | [enjoys].

object --> [apples] | [oranges].
```

Consultas de ejemplo:

```
?- phrase(sentence, Ls).
Ls = [the, woman, likes, apples] ;
Ls = [the, woman, likes, oranges] ;
Ls = [the, woman, enjoys, apples] .

?- phrase(sentence, [the,man,likes,apples]).
true .
```

Goles extra

Los objetivos adicionales permiten agregar procesamiento a las cláusulas DCG, por ejemplo, las condiciones que deben cumplir los elementos de la lista.

Los objetivos adicionales se observan entre llaves al final de una cláusula DCG.

```
% DCG clause requiring an integer
int --> [X], {integer(X)}.
```

Uso:

```
?- phrase(int, [3]).
true.

?- phrase(int, [a]).
false.
```

Argumentos extra

Los argumentos adicionales agregan resultados a los predicados de una cláusula DCG, decorando el árbol de derivación. Por ejemplo, es posible crear una gramática algebraica que calcula el valor al final.

Dada una gramática que soporta la adición de la operación:

```
% Extra arguments are passed between parenthesis after the name of the DCG clauses.
exp(C) --> int(A), [+], exp(B), {plus(A, B, C)}.
exp(X) --> int(X).
int(X) --> [X], {integer(X)}.
```

El resultado de esta gramática puede ser validado y consultado:

```
?- phrase(exp(X), [1,+,2,+,3]).
X = 6 ;
```

Lea Gramática de la cláusula definida (DCGs) en línea:
https://riptutorial.com/es/prolog/topic/2426/gramatica-de-la-clausula-definida--dcgs-

Capítulo 7: Listas de diferencia

Introducción

Las listas de diferencias en Prolog denotan el concepto de conocer la estructura de una lista *hasta cierto punto* . El resto de la lista puede dejarse sin consolidar hasta la evaluación completa de un predicado. Una lista donde se desconoce su final se conoce como una *lista abierta* , terminada por un *agujero* . Esta técnica es especialmente útil para validar sintaxis complejas o gramáticas.

Las conocidas gramáticas de cláusula definida (DCG) utilizan listas de diferencias para operar bajo el capó.

Examples

Uso básico

Consideremos el predicado `sumDif/2` , verificado si la estructura de una lista coincide con varias restricciones. El primer término representa la lista para analizar y el segundo término otra lista que contiene la parte de la primera lista que desconocemos para nuestras restricciones.

Para la demostración, `sumDif/2` reconoce una expresión aritmética para sumar *n* enteros.

```
sumDif([X, +|OpenList], Hole) :-
    integer(X),
    sumDif(OpenList, Hole).
```

Sabemos que el primer elemento de la lista para validar es un entero, aquí ilustrado con `X` , seguido del símbolo de la suma (`+`). El resto de la lista que aún debe procesarse más adelante (`OpenList`) se deja sin validar en ese nivel. `Hole` representa la parte de la lista que *no necesitamos* validar.

Demos otra definición del predicado `sumDif/2` para completar la validación de la expresión aritmética:

```
sumDif([X|Hole], Hole) :-
    integer(X).
```

Esperamos un entero llamado `X` directamente al inicio de la lista abierta. Curiosamente, el resto del `Hole` de la lista se desconoce y ese es el propósito de las listas de diferencias: la estructura de la lista se conoce hasta cierto punto.

Finalmente, la pieza faltante viene cuando se evalúa una lista:

```
?- sumDif([1,+,2,+,3], []).
true
```

Esto es cuando se usa el predicado que menciona el final de la lista, aquí `[]` , indica que la lista no contiene elementos adicionales.

Evaluar una expresión aritmética.

Definamos una gramática que nos permita realizar adiciones, multiplicaciones con el uso de paréntesis. Para agregar más valor a este ejemplo, vamos a calcular el resultado de la expresión aritmética. Resumen de la gramática:

> expresión → veces
> expresión → veces '+' expresión
> tiempos → elemento
> tiempos → elemento '*' veces
> elemento → "entero"
> elemento → '(' expresión ')'

Todos los predicados tienen una aridad de 3, porque necesitan abrir la lista, el agujero y el valor de la expresión aritmética.

```
expression(Value, OpenList, FinalHole) :-
    times(Value, OpenList, FinalHole).

expression(SumValue, OpenList, FinalHole) :-
    times(Value1, OpenList, ['+'|Hole1]),
    expression(Value2, Hole1, FinalHole),
    plus(Value1, Value2, SumValue).

times(Value, OpenList, FinalHole) :-
    element(Value, OpenList, FinalHole).

times(TimesValue, OpenList, FinalHole) :-
    element(Value1, OpenList, ['*'|Hole1]),
    times(Value2, Hole1, FinalHole),
    TimesValue is Value1 * Value2.

element(Value, [Value|FinalHole], FinalHole) :-
    integer(Value).

element(Value, ['('|OpenList], FinalHole) :-
    expression(Value, OpenList, [')'|FinalHole]).
```

Para explicar adecuadamente el principio de los *agujeros* y cómo se calcula el valor, tomemos la segunda `expression` cláusula:

```
expression(SumValue, OpenList, FinalHole) :-
    times(Value1, OpenList, ['+'|Hole1]),
    expression(Value2, Hole1, FinalHole),
    plus(Value1, Value2, SumValue).
```

La lista abierta es denotada por el predicado `OpenList` . El primer elemento a validar es *lo que viene antes del símbolo de adición (`+`)* . Cuando se valida el primer elemento, es seguido directamente por el símbolo de adición y por la continuación de la lista, llamada `Hole1` . Sabemos

que `Hole1` es el siguiente elemento para validar y puede ser otra `expression` , por `Hole1` tanto, `Hole1` es el término dado a la `expression` predicado.

El valor siempre está representado en el primer término. En esta cláusula, se define por la suma del `Value1` (todo antes del símbolo de adición) y el `Value2` (todo después del símbolo de suma).

Finalmente, la expresión puede ser evaluada.

```
?- expression(V, [1,+,3,*,'(',5,+,5,')'], []).
V = 31
```

Lea Listas de diferencia en línea: https://riptutorial.com/es/prolog/topic/9414/listas-de-diferencia

Capítulo 8: Los operadores

Examples

Operadores predefinidos

Operadores predefinidos según ISO / IEC 13211-1 y 13211-2:

Prioridad	Tipo	Operador (es)	Utilizar
1200	xfx	`:- -->`	
1200	fx	`:- ?-`	Directiva, consulta
1100	xfy	`;`	
1050	xfy	`->`	
1000	xfy	`','`	
900	fy	`\+`	
700	xfx	`= \\=`	Unificación del término
700	xfx	`== \\== @< @=< @> @>=`	Comparación de términos
700	xfx	`=..`	
700	xfx	`is =:= =\= < > =< >=`	Evaluación aritmética y comparación.
600	xfy	`:`	Cualificación del módulo
500	yfx	`+ - /\ \/`	
400	yfx	`* / div mod // rem << >>`	
200	xfx	`**`	Poder de flotación
200	xfy	`^`	Cuantificación variable, potencia entera
200	fy	`+ - \`	Identidad aritmética, negación; complemento bit a bit

Muchos sistemas proporcionan operadores adicionales como una extensión específica de implementación:

Prioridad	Tipo	Operador (es)	Utilizar
1150	fx	`dynamic multifile discontiguous initialization`	Directivas estandar
1150	fx	`mode public block volatile meta_predicate`	
900	fy	`spy nospy`	

Declaración del operador

En Prolog, los operadores personalizados se pueden definir usando `op/3` :

```
op(+Precedence, +Type, :Operator)
```

- Declara que Operador es un operador de un Tipo con una Precedencia. El operador también puede ser una lista de nombres, en cuyo caso todos los elementos de la lista se declaran como operadores idénticos.
- La precedencia es un número entero entre 0 y 1200, donde 0 elimina la declaración.
- El tipo es uno de los siguientes: `xf` , `yf` , `xfx` , `xfy` , `yfx` , `fy` o `fx` donde `f` indica la posición del functor y `x` e `y` indican las posiciones de los argumentos. `y` denota un término con una precedencia inferior o igual a la precedencia del functor, mientras que `x` denota una precedencia estrictamente inferior.
    - Prefijo: `fx` , `fy`
    - Infijo: `xfx` (no asociativo), `xfy` (asociativo derecho), `yfx` (asociativo izquierdo)
    - Postfijo: `xf` , `yf`

Ejemplo de uso:

```
:- op(900, xf, is_true).

X_0 is_true :-
  X_0.
```

Consulta de ejemplo:

```
?- dif(X, a) is_true.
dif(X, a).
```

Orden de plazo

Se pueden comparar dos términos a través del pedido estándar:

variables @ <números @ <átomos @ <cadenas @ <estructuras @ <listas

Notas:

- Las estructuras se comparan alfabéticamente por functor primero, luego por aridad y por

último por la comparación de cada argumento.

- Las listas comparan por longitud primero, luego por cada elemento.

Operador de pedidos	Tiene éxito si
X @ <Y	X es menor que Y en el orden estándar
X @> Y	X es mayor que Y en el orden estándar
X @ = <Y	X es menor o igual que Y en el orden estándar
X @> = Y	X es mayor o igual que Y en el orden estándar

Consultas de ejemplo:

```
?- alpha @< beta.
true.

?- alpha(1) @< beta.
false.

?- alpha(X) @< alpha(1).
true.

?- alpha(X) @=< alpha(Y).
true.

?- alpha(X) @> alpha(Y).
false.

?- compound(z) @< compound(inner(a)).
true.
```

Igualdad de términos

Operador de igualdad	Tiene éxito si
X = Y	X puede ser unificado con Y
X \ = Y	X no puede ser unificado con Y
X == Y	X e Y son idénticos (es decir, se unifican *sin* que se produzcan enlaces de variables)
X \ == Y	X e Y no son idénticos
X =: = Y	X e Y son aritméticamente iguales
X = \ = Y	X e Y no son aritméticamente iguales

Lea Los operadores en línea: https://riptutorial.com/es/prolog/topic/2479/los-operadores

Capítulo 9: Manejo de errores y excepciones.

Examples

Errores de instanciación

Se **genera un** error de **creación de instancias** si un argumento no está suficientemente *instanciado* .

Críticamente, un error de instanciación *no puede* ser reemplazado por *una falla silenciosa* : fallar en tales casos significaría que no **hay solución** , mientras que un error de instanciación significa que una *instancia* del argumento puede participar en una solución.

Esto contrasta, por ejemplo **, con un error de dominio** , que puede ser reemplazado por una falla silenciosa sin cambiar el significado declarativo de un programa.

Puntos generales sobre el manejo de errores.

Prolog presenta **excepciones** , que son parte del estándar ISO Prolog.

Se puede lanzar una excepción con el `throw/1` , y capturarse con `catch/3` .

La norma ISO define muchos casos en los que se deben o se pueden lanzar errores. Las excepciones estandarizadas son todas de `error(E,_)` formulario `error(E,_)` , donde `E` indica el error. Los ejemplos son `instantiation_error` , `domain_error` y `type_error` , que veremos.

Un predicado importante en relación con las excepciones es `setup_call_cleanup/3` , que vea.

Limpieza después de las excepciones.

El predicado `setup_call_cleanup/3` , que actualmente está siendo considerado para su inclusión en el estándar ISO Prolog y proporcionado por un número cada vez mayor de implementaciones, nos permite asegurarnos de que los recursos se liberen correctamente después de lanzar una excepción.

Una invocación típica es:

```
setup_call_cleanup(open(File, Mode, Stream), process_file(File), close(Stream))
```

Tenga en cuenta que una excepción o interrupción puede ocurrir incluso inmediatamente después `open/3` llamar a `open/3` en este caso. Por esta razón, la fase de `Setup` se realiza *atómicamente* . En los sistemas Prolog que solo proporcionan `call_cleanup/2` , esto es mucho más difícil de expresar.

Errores de tipo y dominio

Se produce un **error de tipo** si un argumento no es del *tipo* esperado. Ejemplos de tipos son:

- `integer`
- `atom`
- `list`

Si el predicado es del tipo esperado, pero fuera del *dominio* esperado, se genera un **error de dominio** .

Por ejemplo, un error de dominio es admisible si se espera un número entero entre 0 y 15, pero el argumento es el número entero 20.

Declarativamente, un error de tipo o dominio es equivalente a *una falla silenciosa* , ya que ninguna instanciación puede hacer que un predicado cuyo argumento sea del tipo incorrecto o de dominio tenga éxito.

Lea Manejo de errores y excepciones. en línea:
https://riptutorial.com/es/prolog/topic/7114/manejo-de-errores-y-excepciones-

Capítulo 10: Monotonicidad

Examples

Razonamiento sobre predicados monotónicos.

Los predicados **monótonos** se pueden depurar aplicando un razonamiento *declarativo* .

En Prólogo puro, un error de programación puede llevar a uno o todos los siguientes fenómenos:

1. el predicado *tiene éxito* incorrectamente en un caso donde debe *fallar*
2. el predicado *falla* incorrectamente en un caso en el que debería *tener éxito*
3. el predicado *bucles de* forma inesperada en el que sólo debe producir un conjunto finito de respuestas.

Como ejemplo, considere cómo podemos depurar el caso (2) mediante el razonamiento declarativo: podemos *eliminar* sistemáticamente los objetivos de las cláusulas del predicado y ver si la consulta *sigue* fallando. En el código monotónico, la eliminación de objetivos puede, como máximo, hacer que el programa resultante sea *más general* . Por lo tanto, podemos identificar los errores al ver cuál de los objetivos conduce al fracaso inesperado.

Ejemplos de predicados monotónicos.

Ejemplos de predicados **monotónicos** son:

- **unificación** con `(=)/2` o `unify_with_occurs_check/2`
- `dif/2` , expresando desigualdad de términos
- **Las restricciones de CLP (FD)** como `(#=)/2` y `(#>)/2` , utilizan un modo de ejecución monotónico.

Los predicados de Prolog que solo usan objetivos monotónicos son a su vez monotónicos.

Los predicados monótonos permiten el razonamiento declarativo:

1. Agregar una restricción (es decir, un objetivo) a una consulta puede, como máximo, *reducir* , nunca extender, el conjunto de soluciones.
2. Eliminar un objetivo de tales predicados puede, como mucho, *extender* , nunca reducir, el conjunto de soluciones.

Predicados no monotónicos

Aquí hay ejemplos de predicados que **no** son monotónicos:

- predicados meta-lógicos como `var/1` , `integer/1` etc.
- predicados de comparación de términos como `(@<)/2` y `(@>=)/2`
- predicados que usan `!/0` , `(\+)/1` y otras construcciones que rompen la monotonicidad
- *Todas las soluciones predican* como `findall/3` y `setof/3` .

Si se usan estos predicados, entonces *agregar* metas puede llevar a más soluciones, lo que va en contra de la importante propiedad declarativa conocida por la lógica de que agregar restricciones puede, como máximo, *reducir* , nunca extender, el conjunto de soluciones.

Como consecuencia, otras propiedades en las que confiamos para la depuración declarativa y otros razonamientos también se rompen. Por ejemplo, los predicados no monotónicos rompen la noción fundamental de **conmutatividad** de conjunción conocida desde la lógica de primer orden. El siguiente ejemplo lo ilustra:

```
?- var(X), X = a.
X = a.

?- X = a, var(X).
false.
```

Los predicados de todas las soluciones, como `findall/3` también rompen la monotonicidad: *agregar* cláusulas puede llevar al *fracaso* de los objetivos que antes *se mantenían* . Esto también va en contra de la montonicidad conocida por la lógica de primer orden, donde *agregar* hechos puede *aumentar* , como máximo, nunca *reducir* el conjunto de consecuencias.

Alternativas monotónicas para construcciones no monotónicas.

Aquí hay ejemplos de cómo usar predicados **monótonos en** *lugar* de construcciones impuras y no monótonas en sus programas:

- `dif/2` está destinado a ser utilizado en *lugar* de construcciones no monotónicas como `(\=)/2`
- las **restricciones** aritméticas (CLP (FD), CLP (Q) y otras) están destinadas a ser utilizadas en *lugar* de predicados aritméticos modificados
- `!/0` casi siempre conduce a programas no monotónicos y debe **evitarse por** completo.
- **Los errores de creación de instancias** se pueden generar en situaciones en las que no puede tomar una decisión acertada en este momento.

Combinando la monotonicidad con la eficiencia.

A veces se argumenta que, en aras de la eficiencia, debemos aceptar el uso de construcciones no monotónicas en los programas Prolog del mundo real.

No hay evidencia de esto. Investigaciones recientes indican que el subconjunto monotónico puro de Prolog no solo puede ser suficiente para expresar la mayoría de los programas del mundo real, sino que también es aceptable en la práctica. Un constructo que recientemente se ha descubierto y alienta esta vista es `if_/3` : combina la monotonicidad con una reducción de los puntos de elección. Ver **Indización dif / 2** .

Por ejemplo, código del formulario:

```
pred(L, Ls) :-
    condition(L),
    then(Ls).
pred(L, Ls) :-
```

```
    \+ condition(L),
    else(Ls).
```

Se puede escribir con `if_/3` como:

```
pred(L, Ls) :-
    if_(condition(L),
        then(Ls),
        else(Ls)).
```

y *combina la* monotonicidad con el determinismo.

Lea Monotonicidad en línea: https://riptutorial.com/es/prolog/topic/3989/monotonicidad

Capítulo 11: Pautas de codificación

Examples

Nombrar

Al programar en Prolog, debemos elegir dos tipos de nombres:

- nombres de **predicados**
- Nombres de **variables** .

Un buen nombre de *predicado* deja claro lo que significa cada argumento. Por convención, los **guiones bajos** se usan en los nombres para separar la descripción de diferentes argumentos. Esto es porque `underscores_keep_even_longer_names_readable` , mientras que `mixingTheCasesDoesNotDoThisToTheSameExtent` .

Ejemplos de buenos nombres de predicados son:

- `parent_child/2`
- `person_likes/2`
- `route_to/2`

Tenga en cuenta que se utilizan nombres *descriptivos* . Se evitan los imperativos. El uso de nombres descriptivos es aconsejable porque los predicados de Prolog se pueden usar típicamente en *múltiples* direcciones, y el nombre debe ser aplicable también a todos o ninguno de los argumentos está instanciado.

La capitalización mixta es más común cuando se seleccionan nombres de *variables* . Por ejemplo: `BestSolutions` , `MinElement` , `GreatestDivisor` . Una convención común para nombrar variables que denotan *estados* sucesivos es usar `S0` , `S1` , `S2` , ..., `S` , donde `S` representa el estado final.

Sangría

Hay solo unas pocas construcciones de lenguaje en Prolog, y varias formas de sangrarlas son comunes.

Independientemente del estilo que se elija, un principio que siempre se debe respetar es **nunca** colocar `(;)/2` al *final* de una línea. Esto se debe `;` y `,` ven muy similares, y `,` frecuentemente ocurre al final de una línea. Por lo tanto, las cláusulas que utilizan una disyunción deben escribirse, por ejemplo, como:

```
(  Goal1
;  Goal2
)
```

Orden de los argumentos

Idealmente, los predicados Prolog pueden usarse en todas las direcciones. Para muchos predicados puros, este es también el caso. Sin embargo, algunos predicados solo funcionan en *modos* particulares, lo que significa patrones de ejemplificación de sus argumentos.

Por convención, el orden de argumentos más común para tales predicados es:

- **los** argumentos de **entrada** se colocan primero. Estos argumentos deben ser instanciados *antes de que* se llame el predicado.
- *los pares* de argumentos que pertenecen juntos se colocan adyacentes, como `p(..., State0, State, ...)`
- **los** argumentos de **salida** previstos se colocan en último lugar Estos predicados son instanciados por el predicado.

Lea Pautas de codificación en línea: https://riptutorial.com/es/prolog/topic/4612/pautas-de-codificacion

Capítulo 12: Predicados extra-lógicos

Examples

Predicados con efectos secundarios.

Los predicados que producen **efectos secundarios** dejan el ámbito de la lógica pura. Estos son por ejemplo:

- `writeq/1`
- `read/1`
- `format/2`

Los efectos secundarios son fenómenos que no se pueden razonar dentro del programa. Por ejemplo, la eliminación de un archivo o salida en el terminal del sistema.

Predicados meta-lógicos

Los predicados que razonan sobre las *instancias* se llaman **meta-lógicos** . Algunos ejemplos son:

- `var/1`
- `ground/1`
- `integer/1`

Estos predicados están fuera del ámbito de los programas de lógica monotónica pura, porque rompen propiedades como la *conmutación* de la conjunción.

Otros predicados que son meta-lógicos incluyen:

- `arg/3`
- `functor/3`
- `(=..)/2`

Estos predicados podrían, *en principio,* ser modelados dentro de la lógica de primer orden, pero requieren un número infinito de cláusulas.

Todos los predicados de soluciones

Los predicados que razonan sobre *todas las soluciones* son extra-lógicos. Estos son por ejemplo:

- `setof/3`
- `findall/3`
- `bagof/3`

! / 0 y predicados relacionados

Los predicados que impiden o prohíben una lectura **declarativa** de los programas Prolog son extra-lógicos. Ejemplos de tales predicados son:

- `!/0`
- `(->)/2` y if-then-else
- `(\+)/1`

Estos predicados solo se pueden entender de manera procesal, teniendo en cuenta el flujo de control real del intérprete y, como tal, están más allá del ámbito de la lógica pura.

Lea Predicados extra-lógicos en línea: https://riptutorial.com/es/prolog/topic/2282/predicados-extra-logicos

Capítulo 13: Programación de la lógica de restricción

Examples

CLP (FD)

Las restricciones de CLP (FD) (*dominios finitos*) implementan aritmética sobre **enteros** . Están disponibles en todas las implementaciones serias de Prolog.

Hay dos casos de uso principales de restricciones CLP (FD):

- Aritmética entera declarativa
- Resolución de problemas combinatorios como tareas de planificación, programación y asignación.

Ejemplos:

```
?- X #= 1+2.
X = 3.

?- 3 #= Y+2.
Y = 1.
```

Tenga en cuenta que si `is/2` utilizara `is/2` en la segunda consulta, se produciría un error de creación de instancias:

```
?- 3 is Y+2.
ERROR: is/2: Arguments are not sufficiently instantiated
```

CLP (Q)

CLP (Q) implementa razonamiento sobre números *racionales* .

Ejemplo:

```
?- { 5/6 = X/2 + 1/3 }.
X = 1.
```

CLP (H)

El propio prólogo se puede considerar como **CLP (H)** : Programación de la lógica de restricción sobre los *términos de Herbrand* . Con esta perspectiva, un programa Prolog publica restricciones sobre los *términos* . Por ejemplo:

```
?- X = f(Y), Y = a.
```

```
X = f(a),
Y = a.
```

Lea Programación de la lógica de restricción en línea:
https://riptutorial.com/es/prolog/topic/2057/programacion-de-la-logica-de-restriccion

Capítulo 14: Programación de orden superior

Examples

predicados de llamada / N

La familia de predicados `call/N` puede llamar objetivos Prolog arbitrarios en tiempo de ejecución:

```
?- G=true, call(G).
true.

?- G=(true,false), call(G).
false.
```

maplista / [2,3]

`maplist/2` y `maplist/3` son predicados de orden superior, que permiten que la definición de un predicado se levante de un solo elemento a las *listas* de dichos elementos. Estos predicados se pueden definir utilizando `call/2` y `call/3` como bloques de construcción y se envían con muchos sistemas Prolog.

Por ejemplo:

```
?- maplist(dif(a), [X,Y,Z]).
dif(X, a),
dif(Y, a),
dif(Z, a).
```

Meta-llamada

En Prolog, la llamada **meta-llamada** es una función de lenguaje incorporada. Todo el código de Prolog está representado por los *términos de* Prolog, lo que permite que los objetivos se construyan dinámicamente y se usen como otros objetivos sin predicados adicionales:

```
?- Goal = dif(X, Y), Goal.
dif(X, Y).
```

Usando este mecanismo, otros predicados de orden superior pueden definirse en el propio Prolog.

foldl / 4

Un *pliegue* (desde la izquierda) es una relación de orden superior entre:

- un predicado con 3 argumentos
- una lista de elementos
- un estado inicial

- un estado final, que es el resultado de aplicar el predicado a elementos sucesivos mientras se transporta a través de estados intermedios.

Por ejemplo: use `foldl/4` para expresar la *suma* de todos los elementos en una lista, usando un predicado como bloque de construcción para definir la suma de *dos* elementos:

```
?- foldl(plus, [2,3,4], 0, S).
S = 9.
```

Llama a una lista de objetivos

Para llamar a una lista de objetivos como si fuera una conjunción de objetivos, combine los predicados de orden superior call / 1 y maplist / 2:

```
?- Gs = [X = a, Y = b], maplist(call, Gs).
Gs = [a=a, b=b],
X = a,
Y = b.
```

Lea Programación de orden superior en línea:
https://riptutorial.com/es/prolog/topic/2420/programacion-de-orden-superior

Capítulo 15: Pureza logica

Examples

dif / 2

El predicado `dif/2` es un predicado **puro:** Se puede utilizar en todas las direcciones y con todos los modelos de instancias, *siempre* que significa que sus dos argumentos son *diferentes.*

Restricciones de CLP (FD)

Las restricciones de CLP (FD) son relaciones completamente puras. Se pueden usar en todas las direcciones para la aritmética de enteros declarativa:

```
?- X #= 1+2.
X = 3.

?- 3 #= Y+2.
Y = 1.
```

Unificación

La unificación es una relación **pura** . No produce efectos secundarios y se puede usar en todas las direcciones, con uno o ambos argumentos total o parcialmente instanciados.

En Prolog, la unificación puede suceder.

- **explícitamente** , utilizando predicados incorporados como `(=)/2` o `unify_with_occurs_check/2`
- **implícitamente** , cuando se usa la unificación para seleccionar una cláusula adecuada.

Lea Pureza logica en línea: https://riptutorial.com/es/prolog/topic/2058/pureza-logica

Capítulo 16: Razonamiento sobre los datos.

Observaciones

Se dio vida a una nueva sección llamada **Estructuras de datos** donde se proporcionan explicaciones de ciertas estructuras + algunos ejemplos simples de creación. Para mantener su contenido conciso y ordenado, no debe contener ninguna documentación sobre la manipulación de datos.

Por lo tanto, esta sección cambió su nombre a "Razonamiento sobre datos" con el propósito de generalizar el razonamiento sobre datos en Prolog. Esto podría incluir temas que van desde la 'inferencia de arriba hacia abajo' hasta 'recorrido de las listas', así como muchos otros. Debido a su amplia generalización, se deben hacer subsecciones claras!

Examples

Recursion

Prolog no tiene iteración, pero toda la iteración se puede reescribir usando la recursión. La recursión aparece cuando un predicado contiene un objetivo que se refiere a sí mismo. Al escribir tales predicados en Prolog, un patrón recursivo estándar siempre tiene al menos dos partes:

- **Cláusula base (no recursiva)** : por lo general, la (s) regla (s) de caso base representará el ejemplo (s) más pequeño posible del problema que está tratando de resolver: una lista sin miembros, o solo un miembro, o si está trabajando con una estructura de árbol, podría tratar con un árbol vacío o un árbol con un solo nodo, etc. Describe de forma no recursiva la base del proceso recursivo.
- **Cláusula recursiva (continua)** : contiene cualquier lógica requerida, incluida una llamada a sí misma, continuando la recursión.

Como ejemplo definiremos el conocido predicado `append/3` . Visto de forma declarativa, el `append(L1,L2,L3)` mantiene cuando la lista `L3` es el resultado de las listas adjuntas `L1` y `L2` . Cuando intentamos averiguar el significado declarativo de un predicado, tratamos de describir las soluciones para las cuales se cumple el predicado. La dificultad radica aquí en tratar de evitar cualquier detalle recurrente paso a paso mientras se sigue teniendo en cuenta el comportamiento procesal que debe mostrar el predicado.

```
% Base case
append([],L,L).

% Recursive clause
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

El caso base declara declarativamente "cualquier L anexada a la lista vacía es L", tenga en cuenta que esto no dice nada acerca de que L esté vacío, o incluso que sea una lista (recuerde,

en Prolog todo se reduce a términos):

```
?- append(X,some_term(a,b),Z).
X = [],
Z = some_term(a, b).
```

Para describir la regla recursiva, aunque Prolog ejecuta las reglas de izquierda a derecha, omitimos la cabeza por un segundo y miramos el cuerpo primero, leyendo la regla de derecha a izquierda:

```
    append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Ahora decimos que si el cuerpo aguanta: "asumiendo que el `append(L1,L2,L3)` es válido"

```
    append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Entonces también lo hace la cabeza: "entonces también lo `append([X|L1],L2,[X|L3])` "

En un lenguaje sencillo, esto simplemente se traduce en:

> Suponiendo que L3 es la concatenación de L1 y L2, entonces [X seguido de L3] también es la concatenación de [X seguido de L1] y L2.

En un ejemplo práctico:

> "Suponiendo que [1,2,3] es la concatenación de [1] y [2,3], entonces [a, 1,2,3] es también la concatenación de [a, 1] y [2,3]. ”

Ahora veamos algunas consultas:

Siempre es una buena idea probar inicialmente su predicado con la **consulta más general en** lugar de proporcionarle un caso de prueba de escenario específico. Piénselo: debido a la unificación de Prolog, no estamos obligados a proporcionar datos de prueba, ¡solo le damos variables gratuitas!

```
?- append(L1,L2,L3).
L1 = [],
L2 = L3 ;                                   % Answer #1
L1 = [_G1162],
L3 = [_G1162|L2] ;                          % Answer #2
L1 = [_G1162, _G1168],
L3 = [_G1162, _G1168|L2] ;                  % Answer #3
L1 = [_G1162, _G1168, _G1174],
L3 = [_G1162, _G1168, _G1174|L2] ;          % Answer #4
...
```

Reemplazemos la variable libre `_G1162` -como la notación con letras alfabéticas para obtener una mejor visión general:

```
?- append(L1,L2,L3).
L1 = [],
```

```
L2 = L3 ;                                   % Answer #1
L1 = [_A],
L3 = [_A|L2] ;                              % Answer #2
L1 = [_A, _B],
L3 = [_A, _B|L2] ;                          % Answer #3
L1 = [_A, _B, _C],
L3 = [_A, _B, _C|L2] ;                      % Answer #4
...
```

En la primera respuesta, el caso base se ajustó al patrón y Prolog hizo una instancia de `L1` a la lista vacía y unificó `L2` y `L3` lo que demuestra que `L3` es la concatenación de la lista vacía y L2.

En la respuesta # 2, a través del retroceso cronológico, la cláusula recursiva entra en juego y Prolog intenta probar que algún elemento en la cabeza de `L1` concatenado con `L2` es `L3` con ese mismo elemento en su lista. Para hacerlo, una nueva variable libre `_A` está unificada con la cabeza de L1 y se ha demostrado que L3 ahora es `[_A|L2]` .

Se realiza una nueva llamada recursiva, ahora con `L1 = [_A]` . Una vez más, Prolog intenta probar que algún elemento colocado en la cabeza de `L1` , concatenado con `L2` es `L3` con ese mismo elemento en su cabeza. Tenga en cuenta que `_A` ya es el jefe de `L1` , que coincide perfectamente con la regla, por lo que ahora, mediante la recursión, Prolog coloca a `_A` frente a una nueva variable libre y obtenemos `L1 = [_A,_B]` y `L3 = [_A,_B|L2]`

Vemos claramente que el patrón recursivo se repite y podemos ver fácilmente que, por ejemplo, el resultado del paso 100 en recursión se vería así:

```
L1 = [X1,X2,..,X99],
L3 = [X1,X2,..,X99|L2]
```

Nota: como es típico de un buen código de Prolog, la definición recursiva de `append/3` no solo nos brinda la posibilidad de *verificar* si una lista es la concatenación de otras dos listas, sino que también *genera* todas las respuestas posibles que satisfacen las relaciones lógicas con cualquiera de ellas. o listas parcialmente instanciadas.

Listas de acceso

Miembro

`member/2` tiene un `member(?Elem, ?List)` firma `member(?Elem, ?List)` y denota `true` si `Elem` es un miembro de `List` . Este predicado se puede usar para acceder a las variables en una lista, donde se recuperan diferentes soluciones a través del seguimiento.

Consultas de ejemplo:

```
?- member(X, [1,2,3]).
X = 1 ;
X = 2 ;
X = 3.

?- member(X,[Y]).
X = Y.
```

```
?- member(X,Y).
Y = [X|_G969] ;
Y = [_G968, X|_G972] ;
Y = [_G968, _G971, X|_G975] ;
Y = [_G968, _G971, _G974, X|_G978]
...
```

La coincidencia de patrones

Cuando los índices a los que necesita acceder son pequeños, la coincidencia de patrones puede ser una buena solución, por ejemplo:

```
third([_,_,X|_], X).
fourth([_,_,_,X|_], X).
```

Lea Razonamiento sobre los datos. en línea:
https://riptutorial.com/es/prolog/topic/2005/razonamiento-sobre-los-datos-

Capítulo 17: Usando Prolog Moderno

Examples

Introducción

Muchos sistemas modernos de Prolog están en continuo desarrollo y han agregado nuevas características para abordar las deficiencias clásicas del lenguaje. Desafortunadamente, muchos libros de texto de Prolog e incluso cursos de enseñanza aún presentan solo el prólogo obsoleto. El objetivo de este tema es ilustrar cómo el Prolog moderno ha superado algunos de los problemas y la sintaxis bastante compleja que aparece en el Prolog antiguo y que aún se puede introducir.

CLP (FD) para aritmética de enteros

Tradicionalmente, Prolog realizaba aritmética usando los operadores `is` y `=:=` . Sin embargo, varios Prólogos actuales ofrecen CLP (FD) (Programación de lógica de restricción sobre dominios finitos) como una alternativa más limpia para la aritmética de enteros. CLP (FD) se basa en almacenar las restricciones que se aplican a un valor entero y combinarlas en la memoria.

CLP (FD) es una extensión en la mayoría de los Prologs que la admiten, por lo que debe cargarse explícitamente. Una vez que se carga, la sintaxis `#=` puede tomar el lugar de ambos `is` y `=:=` . Por ejemplo, en SWI-Prolog:

```
?- X is 2+2.
X = 4.

?- use_module(library(clpfd)).
?- X #= 2+2.
X = 4.
```

A diferencia de `is` , `#=` es capaz de resolver ecuaciones simples y unificar en ambas direcciones:

```
?- 4 is 2+X.
ERROR: is/2: Arguments are not sufficiently instantiated

?- 4 #= 2+X.
X = 2.
```

CLP (FD) proporciona su propia sintaxis de generador.

```
?- between(1,100,X).
X = 1;
X = 2;
X = 3...

?- X in 1..100.
X in 1..100.
```

Tenga en cuenta que el generador no se ejecuta realmente: solo se almacena la restricción de rango, listo para que las restricciones posteriores se combinen con él. El generador puede ser forzado a correr (y restricciones de fuerza bruta) usando el predicado de la `label` :

```
?- X in 1..100, label([X]).
X = 1;
X = 2;
X = 3..
```

El uso de CLP puede permitir una reducción inteligente de los casos de fuerza bruta. Por ejemplo, usando aritmética de enteros de estilo antiguo:

```
?- trace.
?- between(1,10,X), Y is X+5, Y>10.
...
Exit: (8) 6 is 1+5 ? creep
Call: (8) 6 > 10 ? creep
...
X = 6, Y = 11; ...
```

El prólogo aún recorre los valores del 1 al 5, aunque a partir de las condiciones dadas se puede demostrar matemáticamente que estos valores no pueden ser útiles. Utilizando CLP (FD):

```
?- X in 1..10, Y #= X+5, Y #> 10.
X is 6..10,
X+5 #= Y,
Y is 11..15.
```

CLP (FD) hace las cuentas inmediatamente y calcula los rangos disponibles. Agregar una `label([Y])` hará que X pase por los valores útiles 6..10. En este ejemplo de juguete, esto no aumenta el rendimiento porque con un rango tan pequeño como 1-10, el procesamiento de álgebra toma tanto tiempo como el bucle; pero cuando se procesa un mayor rango de números, esto puede reducir de manera valiosa el tiempo de cálculo.

El soporte para CLP (FD) es variable entre Prologs. El mejor desarrollo reconocido de CLP (FD) se encuentra en SICStus Prolog, que es comercial y costoso. SWI-Prolog y otros Prologs abiertos a menudo tienen alguna implementación. Visual Prolog no incluye CLP (FD) en su biblioteca estándar, aunque hay bibliotecas de extensión disponibles.

Forall en lugar de bucles impulsados por fallas

Algunos libros de texto de Prolog "clásicos" todavía utilizan la sintaxis de bucle impulsada por fallas, confusa y propensa a errores, donde se usa una construcción `fail` para forzar el retroceso para aplicar un objetivo a cada valor de un generador. Por ejemplo, para imprimir todos los números hasta un límite dado:

```
fdl(X) :- between(1,X,Y), print(Y), fail.
fdl(_).
```

La gran mayoría de los Prólogos modernos ya no requieren esta sintaxis, en lugar de

proporcionar un predicado de orden superior para abordar esto.

```
nicer(X) :- forall(between(1,X,Y), print(Y)).
```

Esto no solo es mucho más fácil de leer, sino que si se utilizara un objetivo que pudiera fallar en lugar de la *impresión* , su fallo se detectaría y pasaría correctamente, mientras que los fallos de los objetivos en un bucle impulsado por fallas se confunden con el fallo forzado que impulsa el bucle.

Visual Prolog tiene un azúcar sintáctico personalizado para estos bucles, combinado con predicados de función (ver más abajo):

```
vploop(X) :- foreach Y = std::fromTo(1,X) do
                 console::write(X)
             end foreach.
```

Aunque esto parece un imperativo *para el* bucle, todavía sigue las reglas de Prolog: en particular, cada iteración del *foreach* es su propio alcance.

Predicados de estilo de función

Tradicionalmente, en Prolog, las "funciones" (con una salida y entradas enlazadas) se escribían como predicados regulares:

```
mangle(X,Y) :- Y is (X*5)+2.
```

Esto puede crear la dificultad de que si un predicado de estilo de función se llama varias veces, es necesario "encadenar" variables temporales.

```
multimangle(X,Y) :- mangle(X,A), mangle(A,B), mangle(B,Y).
```

En la mayoría de los Prólogos, es posible evitar esto escribiendo un operador de infijo alternativo para usar en lugar de `is` que expande expresiones que incluyen la función alternativa.

```
% Define the new infix operator
:- op(900, xfy, <-).

% Define our function in terms of the infix operator - note the cut to avoid
% the choice falling through
R <- mangle(X) :- R is (X*5)+2, !.

% To make the new operator compatible with is..
R <- X :-
    compound(X),            % If the input is a compound/function
    X =.. [OP, X2, X3],     % Deconstruct it
    R2 <- X2,               % Recurse to evaluate the arguments
    R3 <- X3,
    Expr =.. [OP, R2, R3],  % Rebuild a compound with the evaluated arguments
    R is Expr,              % And send it to is
    !.
R <- X :- R is X, !.        % If it's not a compound, just use is directly
```

Ahora podemos escribir:

```
multimangle(X,Y) :- X <- mangle(mangle(mangle(Y))).
```

Sin embargo, algunos Prólogos modernos van más allá y ofrecen una sintaxis personalizada para este tipo de predicado. Por ejemplo, en Visual Prolog:

```
mangle(X) = Y :- Y = ((X*5)+2).
multimangle(X,Y) :- Y = mangle(mangle(mangle(X))).
```

Tenga en cuenta que el operador `<-` y el predicado de estilo funcional de arriba aún se comportan como *relaciones* : es legal para ellos tener puntos de elección y realizar una unificación múltiple. En el primer ejemplo, evitamos esto utilizando cortes. En Visual Prolog, es normal usar la sintaxis funcional para las relaciones y los puntos de elección se crean de la manera normal, por ejemplo, el objetivo `X = (std::fromTo(1,10))*10` correctamente con enlaces X = 10 , X = 20, X = 30, X = 40, etc.

Declaraciones de flujo / modo

Al programar en Prolog, no siempre es posible, o deseable, crear predicados que se unifiquen para cada combinación posible de parámetros. Por ejemplo, el predicado `between(X,Y,Z)` que expresa que Z es numéricamente entre X e Y. Se implementa fácilmente en los casos en que X, Y y Z están todos unidos (o Z está entre X e Y o no lo es), o donde X e Y están vinculados y Z está libre (Z se unifica con todos los números entre X e Y, o el predicado falla si Y <X); pero en otros casos, como donde X y Z están vinculados y Y es libre, potencialmente hay un número infinito de unificaciones. Aunque esto puede ser implementado, por lo general no lo sería.

La declaración de flujo o las *declaraciones de modo* permiten una descripción explícita de cómo se comportan los predicados cuando se llaman con diferentes combinaciones de parámetros enlazados. En el caso de `between` , la declaración sería:

```
%! between(+X,+Y,+Z) is semidet.
%! between(+X,+Y,-Z) is nondet.
```

Cada línea especifica un patrón de llamada potencial para el predicado. Cada argumento está decorado con `+` para indicar los casos en los que está vinculado, o `-` para indicar los casos donde no lo está (también hay otras decoraciones disponibles para tipos más complejos, como tuplas o listas que pueden estar parcialmente enlazadas). La palabra clave after *es* el comportamiento del predicado en ese caso, y puede ser uno de estos:

- `det` si el predicado siempre tiene éxito sin un punto de elección. Por ejemplo, `add(+X,+Y,-Z)` es `det` porque agregar dos números dados X e Y siempre tendrá exactamente una respuesta.
- `semidet` si el predicado tiene éxito o falla, sin un punto de elección. Como arriba, `between(+X,+Y,+Z)` es `semidet` porque Z es entre X e Y o no lo es.
- `multi` si el predicado siempre tiene éxito, pero puede tener puntos de elección (pero también puede que no). Por ejemplo, el `factor(+X,-Y)` sería `multi` porque un número siempre tiene al

menos un factor, sí, pero puede tener más.
- `nondet` si el predicado puede tener éxito con los puntos de elección, o fallar. Por ejemplo, `between(+X,+Y,-Z)` está `nondet` porque puede haber varias unificaciones posibles de Z a números entre X e Y, o si Y <X entonces no hay números entre ellos y el predicado falla.

Las declaraciones de flujo / modo también se pueden combinar con el etiquetado de argumentos para aclarar qué significan los términos, o con la escritura. Por ejemplo, `between(+From:Int, +To:Int, +Mid:Int) is semidet .`

En los Prólogos puros, las declaraciones de flujo y modo son opcionales y solo se usan para la generación de documentación, pero pueden ser extremadamente útiles para ayudar a los programadores a identificar la causa de los errores de creación de instancias.

En Mercury, las declaraciones de flujo y modo (y tipos) son obligatorias y son validadas por el compilador. La sintaxis utilizada es la anterior.

En Visual Prolog, las declaraciones y tipos de flujo y modo también son obligatorios y la sintaxis es diferente. La declaración anterior se escribiría como:

```
between : (int From, int To, int Mid) determ (i,i,i) nondeterm (i,i,o).
```

El significado es el mismo que el anterior, pero con las diferencias que:

- Las declaraciones de flujo / modo están separadas de las declaraciones de tipo (ya que se supone que el flujo / modo para un solo predicado no variará con la sobrecarga de tipo);
- `i` y `o` se utilizan para `+` y `-` y se comparan con los parámetros basados en el orden;
- Los términos utilizados son diferentes. `det` convierte `procedure` , `semidet` se convierte en `determ` y `nondet` convierte `nondeterm` (`multi` sigue siendo `multi`).

Lea Usando Prolog Moderno en línea: https://riptutorial.com/es/prolog/topic/5499/usando-prolog-moderno

Creditos

S. No	Capítulos	Contributors
1	Empezando con Prolog Language	coder, Community, Eyal, Limmen, manlio, mat, Nick the coder, Norman Creaney, Rajat Jain, Ruslan López Carro, SeekAndDestroy, Willem Van Onsem, Xevaquor
2	Actuación	mat
3	Árboles de derivación	SeekAndDestroy
4	Estructuras de Control	hardmath, Nick the coder
5	Estructuras de datos	Eyal, mat, SeekAndDestroy
6	Gramática de la cláusula definida (DCGs)	false, mat, Will Ness, ZenLulz
7	Listas de diferencia	ZenLulz
8	Los operadores	false, mat, SeekAndDestroy
9	Manejo de errores y excepciones.	mat
10	Monotonicidad	manlio, mat
11	Pautas de codificación	mat
12	Predicados extra-lógicos	false, mat
13	Programación de la lógica de restricción	mat, SeekAndDestroy
14	Programación de orden superior	4444, Eyal, mat
15	Pureza logica	mat
16	Razonamiento sobre los datos.	Daniel Lyons, manlio, mat, SeekAndDestroy

17	Usando Prolog Moderno	Mark Green