Programación Lógica - Parte 1

Paradigmas de Lenguajes de Programación

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

14 de junio de 2024

Sobre Prolog

- Lenguaje de programación lógica.

Sobre Prolog

- Lenguaje de programación lógica.
- Los programas se escriben en un subconjunto de la lógica de primer orden.

Sobre Prolog

- Lenguaje de programación lógica.
- Los programas se escriben en un subconjunto de la lógica de primer orden.
- Es declarativo: se especifican hechos, reglas de inferencia y objetivos, sin indicar cómo se obtiene este último a partir de los primeros.

Sobre Prolog

- Lenguaje de programación lógica.
- Los programas se escriben en un subconjunto de la lógica de primer orden.
- Es declarativo: se especifican hechos, reglas de inferencia y objetivos, sin indicar cómo se obtiene este último a partir de los primeros.
- Cómputo basado en cláusulas de Horn y resolución SLD.

Sobre Prolog

- Lenguaje de programación lógica.
- Los programas se escriben en un subconjunto de la lógica de primer orden.
- Es declarativo: se especifican hechos, reglas de inferencia y objetivos, sin indicar cómo se obtiene este último a partir de los primeros.
- Cómputo basado en cláusulas de Horn y resolución SLD.
- Mundo cerrado: sólo se puede suponer lo que se declaró explícitamente, todo lo que no pueda deducirse a partir del programa se supone falso.

Sobre Prolog

- Lenguaje de programación lógica.
- Los programas se escriben en un subconjunto de la lógica de primer orden.
- Es declarativo: se especifican hechos, reglas de inferencia y objetivos, sin indicar cómo se obtiene este último a partir de los primeros.
- Cómputo basado en cláusulas de Horn y resolución SLD.
- Mundo cerrado: sólo se puede suponer lo que se declaró explícitamente, todo lo que no pueda deducirse a partir del programa se supone falso.
- Tiene un solo tipo: los términos

Sobre Prolog

- Lenguaje de programación lógica.
- Los programas se escriben en un subconjunto de la lógica de primer orden.
- Es declarativo: se especifican hechos, reglas de inferencia y objetivos, sin indicar cómo se obtiene este último a partir de los primeros.
- Cómputo basado en cláusulas de Horn y resolución SLD.
- Mundo cerrado: sólo se puede suponer lo que se declaró explícitamente, todo lo que no pueda deducirse a partir del programa se supone falso.
- Tiene un solo tipo: los términos
- Recomendamos utilizar SWI-Prolog

Bases de conocimiento

Podemos pensar los programas en Prolog como bases de conocimiento que describen el dominio del problema. Están formados por hechos y reglas de inferencia. Se utilizan realizando consultas sobre dicha base.

Bases de conocimiento

Podemos pensar los programas en Prolog como bases de conocimiento que describen el dominio del problema. Están formados por hechos y reglas de inferencia. Se utilizan realizando consultas sobre dicha base.

Base de conocimiento

Lisa quiere a Nelson.
Milhouse quiere a Lisa.
Juanis quiere a Milhouse.
Utter quiere a Milhouse.
Si alguien (X) quiere a (Y) y además Y quiere a otro (Z), entonces X quiere a Z.

Bases de conocimiento

Podemos pensar los programas en Prolog como bases de conocimiento que describen el dominio del problema. Están formados por hechos y reglas de inferencia. Se utilizan realizando consultas sobre dicha base.

Base de conocimiento

Lisa quiere a Nelson.
Milhouse quiere a Lisa.
Juanis quiere a Milhouse.
Utter quiere a Milhouse.
Si alguien (X) quiere a (Y) y además Y quiere a otro (Z), entonces X quiere a Z.

¿Qué consultas podríamos hacer sobre esta base?

Bases de conocimiento

Podemos pensar los programas en Prolog como bases de conocimiento que describen el dominio del problema. Están formados por hechos y reglas de inferencia. Se utilizan realizando consultas sobre dicha base.

Base de conocimiento

Lisa quiere a Nelson.
Milhouse quiere a Lisa.
Juanis quiere a Milhouse.
Utter quiere a Milhouse.
Si alguien (X) quiere a (Y) y además Y quiere a otro (Z), entonces X quiere a Z.

¿Qué consultas podríamos hacer sobre esta base?

Consultas

¿Nadie quiere a Milhouse?
¿Utter quiere a Nelson?
¿Quiénes quieren a Lisa?

Cláusulas y Consultas

```
zombie(juan).
zombie(valeria).

tomaron_mate(juan,carlos).
tomaron_mate(clara,juan).

infectadx(ernesto).
infectadx(X) :- zombie(X).
infectadx(X) :- zombie(Y), tomaron_mate(Y,X).
```

Cláusulas y Consultas

```
zombie(juan).
zombie(valeria).

tomaron_mate(juan,carlos).
tomaron_mate(clara,juan).

infectadx(ernesto).
infectadx(X) :- zombie(X).
infectadx(X) :- zombie(Y), tomaron_mate(Y,X).
```

```
?- zombie(juan).
```

Cláusulas y Consultas

```
zombie(juan).
zombie(valeria).

tomaron_mate(juan,carlos).
tomaron_mate(clara,juan).

infectadx(ernesto).
infectadx(X) :- zombie(X).
infectadx(X) :- zombie(Y), tomaron_mate(Y,X).
```

```
?- zombie(juan).
true.
```

Cláusulas y Consultas

```
zombie(juan).
zombie(valeria).

tomaron_mate(juan,carlos).
tomaron_mate(clara,juan).

infectadx(ernesto).
infectadx(X) :- zombie(X).
infectadx(X) :- zombie(Y), tomaron_mate(Y,X).
```

```
?- zombie(juan).
true.

?- tomaron_mate(juan,X).
```

Cláusulas y Consultas

```
zombie(juan).
zombie(valeria).

tomaron_mate(juan,carlos).
tomaron_mate(clara,juan).

infectadx(ernesto).
infectadx(X) :- zombie(X).
infectadx(X) :- zombie(Y), tomaron_mate(Y,X).
```

```
?- zombie(juan).
true.

?- tomaron_mate(juan,X).
X = carlos.
```

Cláusulas y Consultas

```
zombie(juan).
zombie(valeria).

tomaron_mate(juan,carlos).
tomaron_mate(clara,juan).

infectadx(ernesto).
infectadx(X) :- zombie(X).
infectadx(X) :- zombie(Y), tomaron_mate(Y,X).
```

```
?- zombie(juan).
true.

?- tomaron_mate(juan,X).
X = carlos.

?- infectadx(I).
```

Cláusulas y Consultas

```
zombie(juan).
zombie(valeria).

tomaron_mate(juan,carlos).
tomaron_mate(clara,juan).

infectadx(ernesto).
infectadx(X) :- zombie(X).
infectadx(X) :- zombie(Y), tomaron_mate(Y,X).
```

```
?- zombie(juan).
true.

?- tomaron_mate(juan,X).
X = carlos.

?- infectadx(I).
I = ernesto;
```

Cláusulas y Consultas

```
zombie(juan).
zombie(valeria).

tomaron_mate(juan,carlos).
tomaron_mate(clara,juan).

infectadx(ernesto).
infectadx(X) :- zombie(X).
infectadx(X) :- zombie(Y), tomaron_mate(Y,X).
```

```
?- zombie(juan).
true.

?- tomaron_mate(juan,X).
X = carlos.

?- infectadx(I).
I = ernesto;
I = juan;
```

Cláusulas y Consultas

```
zombie(juan).
zombie(valeria).

tomaron_mate(juan,carlos).
tomaron_mate(clara,juan).

infectadx(ernesto).
infectadx(X) :- zombie(X).
infectadx(X) :- zombie(Y), tomaron_mate(Y,X).
```

```
?- zombie(juan).
true.

?- tomaron_mate(juan,X).
X = carlos.

?- infectadx(I).
I = ernesto;
I = juan;
I = valeria;
```

Cláusulas y Consultas

```
zombie(juan).
zombie(valeria).

tomaron_mate(juan,carlos).
tomaron_mate(clara,juan).

infectadx(ernesto).
infectadx(X) :- zombie(X).
infectadx(X) :- zombie(Y), tomaron_mate(Y,X).
```

```
?- zombie(juan).
true.

?- tomaron_mate(juan,X).
X = carlos.

?- infectadx(I).
I = ernesto;
I = juan;
I = valeria;
I = carlos;
```

Cláusulas y Consultas

```
zombie(juan).
zombie(valeria).

tomaron_mate(juan,carlos).
tomaron_mate(clara,juan).

infectadx(ernesto).
infectadx(X) :- zombie(X).
infectadx(X) :- zombie(Y), tomaron_mate(Y,X).
```

```
?- zombie(juan).
true.

?- tomaron_mate(juan,X).
X = carlos.

?- infectadx(I).
I = ernesto;
I = juan;
I = valeria;
I = carlos;
false.
```

Sintaxis de Prolog

- **Variables:** `X, Persona, _var`
  Valores que todavía no fueron ligados. Después de ligarse ya no pueden ser modificadas. Empiezan con mayúscula o _.

Sintaxis de Prolog

- **Variables:** `X, Persona, _var`
  Valores que todavía no fueron ligados. Después de ligarse ya no pueden ser modificadas. Empiezan con mayúscula o _.

- **Números:** `10, 15.6`

Sintaxis de Prolog

- **Variables:** `X, Persona, _var`
  Valores que todavía no fueron ligados. Después de ligarse ya no pueden ser modificadas. Empiezan con mayúscula o _.

- **Números:** `10, 15.6`

- **Átomos:** `zombie, 'hola mundo'`
  Constantes, texto, nombres de términos compuestos. Empiezan con minúscula o están entre comillas simples.

Sintaxis de Prolog

- **Variables:** `X, Persona, _var`
  Valores que todavía no fueron ligados. Después de ligarse ya no pueden ser modificadas. Empiezan con mayúscula o _.

- **Números:** `10, 15.6`

- **Átomos:** `zombie, 'hola mundo'`
  Constantes, texto, nombres de términos compuestos. Empiezan con minúscula o están entre comillas simples.

- **Términos compuestos:** `tomaron_mate(clara,juan)`
  También llamado estructura. Consiste en un nombre (átomo) seguido de *n* argumentos, cada uno de los cuales es un término. Decimos que *n* es la aridad del término compuesto.

Sintaxis de Prolog

- **Término:** variable, número, átomo o término compuesto.

Sintaxis de Prolog

- **Término:** variable, número, átomo o término compuesto.
- **Cláusula:** es una línea del programa. Termina con punto. Puede ser:

Sintaxis de Prolog

- **Término:** variable, número, átomo o término compuesto.
- **Cláusula:** es una línea del programa. Termina con punto. Puede ser:
  - **Hecho:** `zombie(juan).`

Sintaxis de Prolog

- **Término:** variable, número, átomo o término compuesto.
- **Cláusula:** es una línea del programa. Termina con punto. Puede ser:
  - **Hecho:** `zombie(juan).`
  - **Regla:** `infectadx(X) :- zombie(Y), tomaron_mate(Y,X).`
    El símbolo `:-` se puede pensar como un $\Leftarrow$, y las comas como $\wedge$.

Sintaxis de Prolog

- **Término:** variable, número, átomo o término compuesto.
- **Cláusula:** es una línea del programa. Termina con punto. Puede ser:
  - **Hecho:** `zombie(juan).`
  - **Regla:** `infectadx(X) :- zombie(Y), tomaron_mate(Y,X).`
    El símbolo `:-` se puede pensar como un $\Leftarrow$, y las comas como $\wedge$.
- **Predicado:** colección de cláusulas.

Sintaxis de Prolog

- **Término:** variable, número, átomo o término compuesto.
- **Cláusula:** es una línea del programa. Termina con punto. Puede ser:
  - **Hecho:** `zombie(juan).`
  - **Regla:** `infectadx(X) :- zombie(Y), tomaron_mate(Y,X).`
    El símbolo `:-` se puede pensar como un $\Leftarrow$, y las comas como $\wedge$.
- **Predicado:** colección de cláusulas.
- **Objetivo (goal):** consulta hecha al motor de Prolog. Conjunción de términos compuestos. Por ejemplo:
  `tomaron_mate(juan,X), zombie(X).`

Ejercicios

Vamos a representar a los naturales (incluyendo el 0) como `cero` y `suc(X)`. Consideremos el siguiente predicado:

```
natural(cero).
natural(suc(X)) :- natural(X).
```

Ejercicios

Vamos a representar a los naturales (incluyendo el 0) como `cero` y `suc(X)`. Consideremos el siguiente predicado:

```
natural(cero).
natural(suc(X)) :- natural(X).
```

- Escribir el predicado `menor(X,Y)` que es verdadero cuando X es menor que Y.

Ejercicios

Vamos a representar a los naturales (incluyendo el 0) como `cero` y `suc(X)`. Consideremos el siguiente predicado:

```
natural(cero).
natural(suc(X)) :- natural(X).
```

- Escribir el predicado `menor(X,Y)` que es verdadero cuando X es menor que Y.
- Indicar qué ocurre si efectuamos las siguientes consultas:
  - `menor(cero,uno).`
  - `menor(cero,X).`
  - `menor(suc(cero),cero).`
  - `menor(X,Y).`

Sustitución y Unificación

Sea *Term* el conjunto formado por todos los posibles términos. Una **sustitución** es una función $\sigma : \mathit{Variables} \rightarrow \mathit{Term}$. Podemos extender σ a una función $\mathit{Term} \rightarrow \mathit{Term}$ de la siguiente manera:

Sustitución y Unificación

Sea *Term* el conjunto formado por todos los posibles términos. Una **sustitución** es una función $\sigma : \mathit{Variables} \to \mathit{Term}$. Podemos extender σ a una función $\mathit{Term} \to \mathit{Term}$ de la siguiente manera:

$$\sigma(c) = c$$

Sustitución y Unificación

Sea *Term* el conjunto formado por todos los posibles términos. Una **sustitución** es una función $\sigma : \mathit{Variables} \rightarrow \mathit{Term}$. Podemos extender σ a una función $\mathit{Term} \rightarrow \mathit{Term}$ de la siguiente manera:

$$\sigma(c) = c$$

$$\sigma(f(t_1, ..., t_n)) = f(\sigma(t_1), ..., \sigma(t_n))$$

Sustitución y Unificación

Sea *Term* el conjunto formado por todos los posibles términos. Una **sustitución** es una función $\sigma : \mathit{Variables} \rightarrow \mathit{Term}$. Podemos extender σ a una función $\mathit{Term} \rightarrow \mathit{Term}$ de la siguiente manera:

$$\sigma(c) = c$$

$$\sigma(f(t_1, ..., t_n)) = f(\sigma(t_1), ..., \sigma(t_n))$$

Por ejemplo, si $\sigma = \{X \leftarrow a, Y \leftarrow Z\}$, entonces:

$$\sigma(b(X, Y, c)) = b(\sigma(X), \sigma(Y), \sigma(c)) = b(a, Z, c)$$

Sustitución y Unificación

Dado un conjunto de ecuaciones de unificación $S = \{s_1 \doteq t_1, ..., s_n \doteq t_n\}$ (con s_i, t_i términos), queremos saber si existe una sustitución σ tal que $\sigma(s_i) = \sigma(t_i)\ \forall i \in 1..n$. En caso de existir, se dice que σ es un **unificador** de S.

Sustitución y Unificación

Dado un conjunto de ecuaciones de unificación $S = \{s_1 \doteq t_1, ..., s_n \doteq t_n\}$ (con s_i, t_i términos), queremos saber si existe una sustitución σ tal que $\sigma(s_i) = \sigma(t_i)\ \forall i \in 1..n$. En caso de existir, se dice que σ es un **unificador** de S.

Por ejemplo, `f(X)` $\doteq$ `f(Y)` se puede resolver con la sustitución $\sigma = \{X \leftarrow 0, Y \leftarrow 0\}$, ya que quedaría `f(0) = f(0)`

Sustitución y Unificación

Dado un conjunto de ecuaciones de unificación $S = \{s_1 \doteq t_1, ..., s_n \doteq t_n\}$ (con s_i, t_i términos), queremos saber si existe una sustitución σ tal que $\sigma(s_i) = \sigma(t_i)\ \forall i \in 1..n$. En caso de existir, se dice que σ es un **unificador** de S.

Por ejemplo, `f(X)` $\doteq$ `f(Y)` se puede resolver con la sustitución $\sigma = \{X \leftarrow 0, Y \leftarrow 0\}$, ya que quedaría `f(0) = f(0)`, pero claramente parecen más interesantes las sustituciones de la forma $\sigma' = \{X \leftarrow Y\}$.

Sustitución y Unificación

Dado un conjunto de ecuaciones de unificación $S = \{s_1 \doteq t_1, ..., s_n \doteq t_n\}$ (con s_i, t_i términos), queremos saber si existe una sustitución σ tal que $\sigma(s_i) = \sigma(t_i)\ \forall i \in 1..n$. En caso de existir, se dice que σ es un **unificador** de S.

Por ejemplo, `f(X)` $\doteq$ `f(Y)` se puede resolver con la sustitución $\sigma = \{X \leftarrow 0, Y \leftarrow 0\}$, ya que quedaría `f(0) = f(0)`, pero claramente parecen más interesantes las sustituciones de la forma $\sigma' = \{X \leftarrow Y\}$.

Nosotrxs estamos interesadxs en el **unificador más general (MGU)** de S.

Sustitución y Unificación

Dado un programa lógico P y un goal $G_1, \ldots, G_n$, se quiere saber si el goal es consecuencia lógica de P.

Sustitución y Unificación

Dado un programa lógico P y un goal $G_1, \ldots, G_n$, se quiere saber si el goal es consecuencia lógica de P.

La regla de resolución que se utiliza es:

$$\frac{G_1, \ldots, G_i, \ldots, G_n \qquad H \ :- \ A_1, \ldots, A_k \qquad \sigma \text{ es el MGU de } G_i \text{ y } H}{\sigma(G1, \ldots, G_{i-1}, A_1, \ldots, A_k, G_{i+1}, \ldots, G_n)}$$

Sustitución y Unificación

Dado un programa lógico P y un goal $G_1, \ldots, G_n$, se quiere saber si el goal es consecuencia lógica de P.

La regla de resolución que se utiliza es:

$$\frac{G_1, \ldots, G_i, \ldots, G_n \qquad H \ :- \ A_1, \ldots, A_k \qquad \sigma \text{ es el MGU de } G_i \text{ y } H}{\sigma(G1, \ldots, G_{i-1}, A_1, \ldots, A_k, G_{i+1}, \ldots, G_n)}$$

La conclusión de la regla de resolución es el nuevo goal a resolver.

Sustitución y Unificación

Dado un programa lógico P y un goal $G_1, \ldots, G_n$, se quiere saber si el goal es consecuencia lógica de P.

La regla de resolución que se utiliza es:

$$\frac{G_1, \ldots, G_i, \ldots, G_n \qquad H \ :- \ A_1, \ldots, A_k \qquad \sigma \text{ es el MGU de } G_i \text{ y } H}{\sigma(G1, \ldots, G_{i-1}, A_1, \ldots, A_k, G_{i+1}, \ldots, G_n)}$$

La conclusión de la regla de resolución es el nuevo goal a resolver.

Prolog resuelve el goal empezando desde G_1, de izquierda a derecha y haciendo DFS. Para cada G_i, recorre el programa de arriba hacia abajo buscando unificar G_i con la cabeza de una cláusula.

Sustitución y Unificación

Dado un programa lógico P y un goal $G_1, \ldots, G_n$, se quiere saber si el goal es consecuencia lógica de P.

La regla de resolución que se utiliza es:

$$\frac{G_1, \ldots, G_i, \ldots, G_n \qquad H \ :- \ A_1, \ldots, A_k \qquad \sigma \text{ es el MGU de } G_i \text{ y } H}{\sigma(G1, \ldots, G_{i-1}, A_1, \ldots, A_k, G_{i+1}, \ldots, G_n)}$$

La conclusión de la regla de resolución es el nuevo goal a resolver.

Prolog resuelve el goal empezando desde G_1, de izquierda a derecha y haciendo DFS. Para cada G_i, recorre el programa de arriba hacia abajo buscando unificar G_i con la cabeza de una cláusula.

Tener en cuenta que el orden de las cláusulas y sus literales en el programa influyen en el resultado.

Sustitución y Unificación

Dado un programa lógico P y un goal $G_1, \ldots, G_n$, se quiere saber si el goal es consecuencia lógica de P.

La regla de resolución que se utiliza es:

$$\frac{G_1, \ldots, G_i, \ldots, G_n \qquad H \ :- \ A_1, \ldots, A_k \qquad \sigma \text{ es el MGU de } G_i \text{ y } H}{\sigma(G1, \ldots, G_{i-1}, A_1, \ldots, A_k, G_{i+1}, \ldots, G_n)}$$

La conclusión de la regla de resolución es el nuevo goal a resolver.

Prolog resuelve el goal empezando desde G_1, de izquierda a derecha y haciendo DFS. Para cada G_i, recorre el programa de arriba hacia abajo buscando unificar G_i con la cabeza de una cláusula.

Tener en cuenta que el orden de las cláusulas y sus literales en el programa influyen en el resultado.

En este link hay ejemplos sobre el proceso de reducción de Prolog.

Ejemplo de resolución

Veamos un ejemplo. Sea el siguiente programa:

```
gato(garfield).
tieneMascota(john,odie).
tieneMascota(john,garfield).
amaALosGatos(X) :- tieneMascota(X,Y), gato(Y).
```

Mostrar el seguimiento (árbol de ejecución) de la consulta: `amaALosGatos(Z).`

Ejemplo de resolución

¿Qué pasa si cambiamos el orden de algunas cláusulas?

```
gato(garfield).
tieneMascota(john,garfield).
tieneMascota(john,odie).
amaALosGatos(X) :- tieneMascota(X,Y), gato(Y).
```

y el goal: `amaALosGatos(Z).`

Resolución

Al evaluar un goal, los resultados posibles son los siguientes:

Resolución

Al evaluar un goal, los resultados posibles son los siguientes:

- `true`: la resolución terminó en la cláusula vacía.

Resolución

Al evaluar un goal, los resultados posibles son los siguientes:

- `true`: la resolución terminó en la cláusula vacía.
- `false`: la resolución terminó en una cláusula que no unifica con ninguna regla del programa.

Resolución

Al evaluar un goal, los resultados posibles son los siguientes:

- `true`: la resolución terminó en la cláusula vacía.
- `false`: la resolución terminó en una cláusula que no unifica con ninguna regla del programa.
- El proceso de aplicación de la regla de resolución no termina.

Reversibilidad

Un predicado define una relación entre elementos. No hay parámetros de "entrada" ni de "salida".

Reversibilidad

Un predicado define una relación entre elementos. No hay parámetros de "entrada" ni de "salida".

Conceptualmente, cualquier argumento podría cumplir ambos roles dependiendo de cómo se consulte.

Reversibilidad

Un predicado define una relación entre elementos. No hay parámetros de "entrada" ni de "salida".

Conceptualmente, cualquier argumento podría cumplir ambos roles dependiendo de cómo se consulte.

Un predicado podría estar implementado asumiendo que ciertas variables ya están instanciadas, por diversas cuestiones prácticas.

Patrones de instanciación

El modo de instanciación esperado por un predicado se comunicará en los comentarios.

Patrones de instanciación

El modo de instanciación esperado por un predicado se comunicará en los comentarios.

```
% pow(+B,+E,-P)
pow(...) :- ...
```

Patrones de instanciación

El modo de instanciación esperado por un predicado se comunicará en los comentarios.

```
% pow(+B,+E,-P)
pow(...) :- ...
```

+X	−X	?X
debe estar instanciado	**no** debe estar instanciado	puede o no estar instanciado

Patrones de instanciación

El modo de instanciación esperado por un predicado se comunicará en los comentarios.

```
% pow(+B,+E,-P)
pow(...) :- ...
```

+X	−X	?X
debe estar instanciado	**no** debe estar instanciado	puede o no estar instanciado

Se debe tener en cuenta que el usuario no puede suponer más cosas de las que se especificaron. En caso de llamar a un predicado con argumentos instanciados de otra manera, el resultado puede no ser el esperado.

Aritmética

El motor de operaciones aritméticas de Prolog es independiente del motor lógico (es extra-lógico).

Aritmética

El motor de operaciones aritméticas de Prolog es independiente del motor lógico (es extra-lógico).

Expresión aritmética:

- Un número.

Aritmética

El motor de operaciones aritméticas de Prolog es independiente del motor lógico (es extra-lógico).

Expresión aritmética:

- Un número.
- Una variable ya instanciada en una expresión aritmética.

Aritmética

El motor de operaciones aritméticas de Prolog es independiente del motor lógico (es extra-lógico).

Expresión aritmética:

- Un número.
- Una variable ya instanciada en una expresión aritmética.
- `E1+E2`, `E1-E2`, `E1*E2`, `E1/E2`, etc, siendo `E1` y `E2` expresiones aritméticas.

Aritmética

Algunos operadores aritméticos:

- `E1 < E2`, `E1 =< E2`, `E1 =:= E2`, `E1 =\= E2`: evalúa ambas expresiones aritméticas y realiza la comparación indicada.

Aritmética

Algunos operadores aritméticos:

- `E1 < E2`, `E1 =< E2`, `E1 =:= E2`, `E1 =\= E2`: evalúa ambas expresiones aritméticas y realiza la comparación indicada.

- `X is E`: tiene éxito sí y sólo si `X` unifica con el resultado de evaluar la expresión aritmética `E`.

Aritmética

Algunos operadores aritméticos:

- `E1 < E2`, `E1 =< E2`, `E1 =:= E2`, `E1 =\= E2`: evalúa ambas expresiones aritméticas y realiza la comparación indicada.

- `X is E`: tiene éxito sí y sólo si `X` unifica con el resultado de evaluar la expresión aritmética `E`.

Algunos operadores no aritméticos:

- `X = Y`: tiene éxito si y sólo si `X` unifica con `Y`.

Aritmética

Algunos operadores aritméticos:

- `E1 < E2`, `E1 =< E2`, `E1 =:= E2`, `E1 =\= E2`: evalúa ambas expresiones aritméticas y realiza la comparación indicada.
- `X is E`: tiene éxito sí y sólo si `X` unifica con el resultado de evaluar la expresión aritmética `E`.

Algunos operadores no aritméticos:

- `X = Y`: tiene éxito si y sólo si `X` unifica con `Y`.
- `X \= Y`: `X` no unifica con `Y`. Ambos términos deben estar instanciados.

Aritmética

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

Aritmética

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

Aritmética

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

?- 1+1 = 2.
```

Aritmética

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

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

Aritmética

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

?- 1+1 = 2.
false.

?- 1+1 = 1+1.
```

Aritmética

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

?- 1+1 = 2.
false.

?- 1+1 = 1+1.
true.
```

Aritmética

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

?- 1+1 = 2.
false.

?- 1+1 = 1+1.
true.

?- X is 1+1.
```

Aritmética

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

?- 1+1 = 2.
false.

?- 1+1 = 1+1.
true.

?- X is 1+1.
X = 2.
```

Aritmética

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

?- 1+1 = 2.
false.

?- 1+1 = 1+1.
true.

?- X is 1+1.
X = 2.

?- 2 is 1+1.
```

Aritmética

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

?- 1+1 = 2.
false.

?- 1+1 = 1+1.
true.

?- X is 1+1.
X = 2.

?- 2 is 1+1.
true.
```

Aritmética

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

?- 1+1 = 2.
false.

?- 1+1 = 1+1.
true.

?- X is 1+1.
X = 2.

?- 2 is 1+1.
true.

?- 1+1 is 2.
```

Aritmética

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

?- 1+1 = 2.
false.

?- 1+1 = 1+1.
true.

?- X is 1+1.
X = 2.

?- 2 is 1+1.
true.

?- 1+1 is 2.
false.
```

Aritmética

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

?- 1+1 = 2.
false.

?- 1+1 = 1+1.
true.

?- X is 1+1.
X = 2.

?- 2 is 1+1.
true.

?- 1+1 is 2.
false.

?- 1+1 is 1+1.
```

Aritmética

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

?- 1+1 = 2.
false.

?- 1+1 = 1+1.
true.

?- X is 1+1.
X = 2.

?- 2 is 1+1.
true.

?- 1+1 is 2.
false.

?- 1+1 is 1+1.
false.
```

Ejercicio

Definir el predicado `entre(+X,+Y,-Z)` que sea verdadero cuando el número entero `Z` esté comprendido entre los números enteros `X` e `Y` (inclusive).

Ejercicio

Definir el predicado `entre(+X,+Y,-Z)` que sea verdadero cuando el número entero `Z` esté comprendido entre los números enteros `X` e `Y` (inclusive).

Notar que lo que se nos pide es un predicado capaz de instanciar sucesivamente `Z` en cada número entero entre `X` e `Y`:

```
?- entre(1,3,Z).
  Z = 1;
  Z = 2;
  Z = 3;
  false.
```

Listas

Sintaxis:

- `[]`
- `[X,Y,...,Z | L]`

Listas

Sintaxis:

- `[]`
- `[X,Y,...,Z | L]`

Ejemplos:

- `[1,2], [1,2 | []], [1 | [2]], [1 | [2 | []]]`
- `[1,cero,'hola mundo',[3,5]]`

Ejercicios sobre listas

- Definir el predicado `long(+XS,-L)` que relaciona una lista con su longitud.

Ejercicios sobre listas

- Definir el predicado `long(+XS,-L)` que relaciona una lista con su longitud.

- Definir el predicado `sinConsecRep(+XS,-YS)` que relaciona una lista con otra que contiene los mismos elementos sin las repeticiones consecutivas.
  Por ejemplo:
  ```
  ?- sinConsecRep([1,2,2,3,2],L).
  L = [1,2,3,2].
  ```

Ejercicios sobre listas

- Definir el predicado `long(+XS,-L)` que relaciona una lista con su longitud.

- Definir el predicado `sinConsecRep(+XS,-YS)` que relaciona una lista con otra que contiene los mismos elementos sin las repeticiones consecutivas.
  Por ejemplo:
  ```
  ?- sinConsecRep([1,2,2,3,2],L).
  L = [1,2,3,2].
  ```

- Definir el predicado `partes(+XS,-YS)` que relaciona una lista con los elementos de su conjunto de partes.

Ejercicios sobre listas

Utilizando el siguiente predicado:

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

Implementar los siguientes:

Ejercicios sobre listas

Utilizando el siguiente predicado:

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

Implementar los siguientes:

- `prefijo(+L,?P)`: que tiene éxito si `P` es un prefijo de la lista `L`.

Ejercicios sobre listas

Utilizando el siguiente predicado:

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

Implementar los siguientes:

- `prefijo(+L,?P)`: que tiene éxito si `P` es un prefijo de la lista `L`.
- `sufijo(+L,?S)`: que tiene éxito si `S` es un sufijo de la lista `L`.

Ejercicios sobre listas

Utilizando el siguiente predicado:

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

Implementar los siguientes:

- `prefijo(+L,?P)`: que tiene éxito si `P` es un prefijo de la lista `L`.
- `sufijo(+L,?S)`: que tiene éxito si `S` es un sufijo de la lista `L`.
- `sublista(+L,?SL)`: que tiene éxito si `SL` es una sublista de `L`.

Ejercicios sobre listas

Utilizando el siguiente predicado:

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

Implementar los siguientes:

- `prefijo(+L,?P)`: que tiene éxito si `P` es un prefijo de la lista `L`.
- `sufijo(+L,?S)`: que tiene éxito si `S` es un sufijo de la lista `L`.
- `sublista(+L,?SL)`: que tiene éxito si `SL` es una sublista de `L`.
- `insertar(?X,+L,?LX)`: que tiene éxito si `LX` puede obtenerse insertando a `X` en alguna posición de `L`.

Ejercicios sobre listas

Utilizando el siguiente predicado:

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

Implementar los siguientes:

- `prefijo(+L,?P)`: que tiene éxito si `P` es un prefijo de la lista `L`.
- `sufijo(+L,?S)`: que tiene éxito si `S` es un sufijo de la lista `L`.
- `sublista(+L,?SL)`: que tiene éxito si `SL` es una sublista de `L`.
- `insertar(?X,+L,?LX)`: que tiene éxito si `LX` puede obtenerse insertando a `X` en alguna posición de `L`.
- `permutacion(+L,?P)`: que tiene éxito si `P` es una permutación de la lista `L`.

Ejercicios sobre listas

Considerando el siguiente predicado:

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```

Ejercicios sobre listas

Considerando el siguiente predicado:

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```

Realizar un seguimiento de las siguientes consultas:

- `?- member(2,[1,2]).`
- `?- member(X,[1,2]).`
- `?- member(5,[X,3,X]).`
- `?- length(L,2), member(5,L), member(2,L).`

Fin

¿¿¿¿¿Preguntas?????