Prolog en 30 minutos

Adaptado de un tutorial de Eduardo C. Garrido Merchán

Indice

- Introducción y motivación.

Indice

- Introducción y motivación.
- Hechos, reglas y consultas.

Indice

- Introducción y motivación.
- Hechos, reglas y consultas.
- Condicionales en Prolog (AND y OR).

Indice

- Introducción y motivación.
- Hechos, reglas y consultas.
- Condicionales en Prolog (AND y OR).
- Recursión.

Indice

- Introducción y motivación.
- Hechos, reglas y consultas.
- Condicionales en Prolog (AND y OR).
- Recursión.
- Listas.

Indice

- Introducción y motivación.
- Hechos, reglas y consultas.
- Condicionales en Prolog (AND y OR).
- Recursión.
- Listas.
- Cortes.

Introducción

- La programación lógica trata de proveer una solución práctica para realizar inferencia de nuevo conocimiento a partir del conocimiento existente.

Introducción

- La programación lógica trata de proveer una solución práctica para realizar inferencia de nuevo conocimiento a partir del conocimiento existente.
- En el paradigma de programación estructurada, codificamos un algoritmo que resuelve un programa mediante instrucciones.

Introducción

- La programación lógica trata de proveer una solución práctica para realizar inferencia de nuevo conocimiento a partir del conocimiento existente.
- En el paradigma de programación estructurada, codificamos un algoritmo que resuelve un programa mediante instrucciones.
- En programación lógica, codificamos un problema mediante la declaración de hechos y reglas.

Introducción

- La programación lógica trata de proveer una solución práctica para realizar inferencia de nuevo conocimiento a partir del conocimiento existente.
- En el paradigma de programación estructurada, codificamos un algoritmo que resuelve un programa mediante instrucciones.
- En programación lógica, codificamos un problema mediante la declaración de hechos y reglas.
- Es decir, mediante la descripción del problema y sus propiedades. Con esta descripción y un motor de inferencia, podemos inferir nuevo conocimiento.

Introducción

- La programación lógica trata de proveer una solución práctica para realizar inferencia de nuevo conocimiento a partir del conocimiento existente.
- En el paradigma de programación estructurada, codificamos un algoritmo que resuelve un programa mediante instrucciones.
- En programación lógica, codificamos un problema mediante la declaración de hechos y reglas.
- Es decir, mediante la descripción del problema y sus propiedades. Con esta descripción y un motor de inferencia, podemos inferir nuevo conocimiento.
- La base de conocimiento agrupa la declaración de estos hechos y reglas.

Introducción

- La programación lógica trata de proveer una solución práctica para realizar inferencia de nuevo conocimiento a partir del conocimiento existente.
- En el paradigma de programación estructurada, codificamos un algoritmo que resuelve un programa mediante instrucciones.
- En programación lógica, codificamos un problema mediante la declaración de hechos y reglas.
- Es decir, mediante la descripción del problema y sus propiedades. Con esta descripción y un motor de inferencia, podemos inferir nuevo conocimiento.
- La base de conocimiento agrupa la declaración de estos hechos y reglas.
- El motor de inferencia toma una base de conocimiento y resuelve consultas mediante el algoritmo de unificación.

Prolog

- Prolog es el lenguaje de programación lógica por excelencia. Su nombre mismo lo indica: "Pro" de programming y "log" de logic.

Prolog

- Prolog es el lenguaje de programación lógica por excelencia. Su nombre mismo lo indica: "Pro" de programming y "log" de logic.
- Usa el algoritmo de unificación para hacer inferencia de nuevo conocimiento sobre una base de conocimiento existente.

Prolog

- Prolog es el lenguaje de programación lógica por excelencia. Su nombre mismo lo indica: "Pro" de programming y "log" de logic.
- Usa el algoritmo de unificación para hacer inferencia de nuevo conocimiento sobre una base de conocimiento existente.
- La sintaxis de Prolog es tremendamente sencilla. Podemos codificar una base de conocimiento mediante hechos y reglas.

Hechos y consultas

- Veamos un ejemplo de un hecho adentrándonos en el mundo de Juego de Tronos, que nos acompañará a lo largo del tutorial.

Hechos y consultas

- Veamos un ejemplo de un hecho adentrándonos en el mundo de Juego de Tronos, que nos acompañará a lo largo del tutorial.
- `stark("jon").`

Hechos y consultas

- Veamos un ejemplo de un hecho adentrándonos en el mundo de Juego de Tronos, que nos acompañará a lo largo del tutorial.
- `stark("jon").`
- Esto implica que jon es un stark. Es imprescindible el “.” al final de hechos y reglas. Si preguntamos por el objetivo:

Hechos y consultas

- Veamos un ejemplo de un hecho adentrándonos en el mundo de Juego de Tronos, que nos acompañará a lo largo del tutorial.
- `stark("jon").`
- Esto implica que jon es un stark. Es imprescindible el “.” al final de hechos y reglas. Si preguntamos por el objetivo:
- `:- stark(X).`

Hechos y consultas

- Veamos un ejemplo de un hecho adentrándonos en el mundo de Juego de Tronos, que nos acompañará a lo largo del tutorial.
- `stark("jon").`
- Esto implica que jon es un stark. Es imprescindible el “.” al final de hechos y reglas. Si preguntamos por el objetivo:
- `:- stark(X).`
- Donde `X` es una variable lógica, el intérprete de Prolog nos devolverá:

Hechos y consultas

- Veamos un ejemplo de un hecho adentrándonos en el mundo de Juego de Tronos, que nos acompañará a lo largo del tutorial.
- `stark("jon").`
- Esto implica que jon es un stark. Es imprescindible el "." al final de hechos y reglas. Si preguntamos por el objetivo:
- `:- stark(X).`
- Donde `X` es una variable lógica, el intérprete de Prolog nos devolverá:
- `X = "jon".`

Hechos y consultas

- Veamos un ejemplo de un hecho adentrándonos en el mundo de Juego de Tronos, que nos acompañará a lo largo del tutorial.
- `stark("jon").`
- Esto implica que jon es un stark. Es imprescindible el "." al final de hechos y reglas. Si preguntamos por el objetivo:
- `:- stark(X).`
- Donde `X` es una variable lógica, el intérprete de Prolog nos devolverá:
- `X = "jon".`
- Ha unificado la variable `X` con el término `jon`. Esta consulta equivale a preguntar por un valor de `X` tal que `X` sea un `stark`.

Reglas

- En Prolog, también podemos codificar reglas. Ojo, la sintaxis a priori es poco intuitiva, pero una vez que se entiende es trivial y muy fácil usarla. Veamos un ejemplo:

Reglas

- En Prolog, también podemos codificar reglas. Ojo, la sintaxis a priori es poco intuitiva, pero una vez que se entiende es trivial y muy fácil usarla. Veamos un ejemplo:
- `padre(X, Y) :- hijo(Y, X).`

Reglas

- En Prolog, también podemos codificar reglas. Ojo, la sintaxis a priori es poco intuitiva, pero una vez que se entiende es trivial y muy fácil usarla. Veamos un ejemplo:
- `padre(X, Y) :- hijo(Y, X).`
- Esta regla se divide en dos partes y se leería de la siguiente forma: "Si Y es hijo de X, entonces X es padre de Y".

Reglas

- En Prolog, también podemos codificar reglas. Ojo, la sintaxis a priori es poco intuitiva, pero una vez que se entiende es trivial y muy fácil usarla. Veamos un ejemplo:
- `padre(X, Y) :- hijo(Y, X).`
- Esta regla se divide en dos partes y se leería de la siguiente forma: "Si Y es hijo de X, entonces X es padre de Y".
- Haciendo el simil en programación estructurada: "if(hijo(Y,X)) then padre(X, Y)."

Reglas

- En Prolog, también podemos codificar reglas. Ojo, la sintaxis a priori es poco intuitiva, pero una vez que se entiende es trivial y muy fácil usarla. Veamos un ejemplo:
- `padre(X, Y) :- hijo(Y, X).`
- Esta regla se divide en dos partes y se leería de la siguiente forma: "Si Y es hijo de X, entonces X es padre de Y".
- Haciendo el simil en programación estructurada: "if(hijo(Y,X)) then padre(X, Y)."
- Podemos ver que la condición es lo que está a la derecha del operador ":-".

Reglas

- En Prolog, también podemos codificar reglas. Ojo, la sintaxis a priori es poco intuitiva, pero una vez que se entiende es trivial y muy fácil usarla. Veamos un ejemplo:
- `padre(X, Y) :- hijo(Y, X).`
- Esta regla se divide en dos partes y se leería de la siguiente forma: "Si Y es hijo de X, entonces X es padre de Y".
- Haciendo el simil en programación estructurada: "if(hijo(Y,X)) then padre(X, Y)."
- Podemos ver que la condición es lo que está a la derecha del operador ":-".
- Si se cumplen los predicados que están a la derecha, entonces, se ejecuta la parte izquierda.

Reglas y consultas

- Siendo nuestra base de conocimiento:

Reglas y consultas

- Siendo nuestra base de conocimiento:
- ```
  hijo("jon", "eddard").
  padre(X, Y) :- hijo(Y, X).
  ```
  ```

Reglas y consultas

- Siendo nuestra base de conocimiento:
- ```
  hijo("jon", "eddard").
  padre(X, Y) :- hijo(Y, X).
  ```
- Si preguntamos al intérprete:
  ```

Reglas y consultas

- Siendo nuestra base de conocimiento:
- ```
  hijo("jon", "eddard").
  padre(X, Y) :- hijo(Y, X).
  ```
- Si preguntamos al intérprete:
- ```
  :- padre(X, "jon").
  ```

Reglas y consultas

- Siendo nuestra base de conocimiento:
- ```
  hijo("jon", "eddard").
  padre(X, Y) :- hijo(Y, X).
  ```
- Si preguntamos al intérprete:
- ```
  :- padre(X, "jon").
  ```
- El potencial de Prolog es que podemos tranquilamente poner las variables donde queramos, pudiendo preguntar tranquilamente:

Reglas y consultas

- Siendo nuestra base de conocimiento:
- ```
  hijo("jon", "eddard").
  padre(X, Y) :- hijo(Y, X).
  ```
- Si preguntamos al intérprete:
- ```
  :- padre(X, "jon").
  ```
- El potencial de Prolog es que podemos tranquilamente poner las variables donde queramos, pudiendo preguntar tranquilamente:
- ```
  :- padre(X, Y).
  ```
  ```

Reglas y consultas

- Siendo nuestra base de conocimiento:
- ```
  hijo("jon", "eddard").
  padre(X, Y) :- hijo(Y, X).
  ```
- Si preguntamos al intérprete:
- ```
  :- padre(X, "jon").
  ```
- El potencial de Prolog es que podemos tranquilamente poner las variables donde queramos, pudiendo preguntar tranquilamente:
- ```
  :- padre(X, Y).
  ```
- ```
  X="eddard", Y="jon".
  ```

Como se codifica un AND en Prolog?

- Podemos incluir N predicados en el antecedente de una regla separados por comas. Cada "," hará las funciones del operador AND.

Como se codifica un AND en Prolog?

- Podemos incluir N predicados en el antecedente de una regla separados por comas. Cada "," hará las funciones del operador AND.
- Es decir, para que el consecuente sea cierto, se deben verificar todos los predicados del antecedente. Veamos un ejemplo:

Como se codifica un AND en Prolog?

- Podemos incluir N predicados en el antecedente de una regla separados por comas. Cada "," hará las funciones del operador AND.
- Es decir, para que el consecuente sea cierto, se deben verificar todos los predicados del antecedente. Veamos un ejemplo:
- `familia_stark(X, Y) :- familia(X, Y), stark(X), stark(Y).`

Como se codifica un AND en Prolog?

- Podemos incluir N predicados en el antecedente de una regla separados por comas. Cada "," hará las funciones del operador AND.
- Es decir, para que el consecuente sea cierto, se deben verificar todos los predicados del antecedente. Veamos un ejemplo:
- `familia_stark(X, Y) :- familia(X, Y), stark(X), stark(Y).`
- Esta regla se lee como: "Si X e Y son de la misma familia y X es un Stark e Y es un Stark, entonces, X e Y pertenecen a la familia Stark".

Como se codifica un AND en Prolog?

- Podemos incluir N predicados en el antecedente de una regla separados por comas. Cada "," hará las funciones del operador AND.
- Es decir, para que el consecuente sea cierto, se deben verificar todos los predicados del antecedente. Veamos un ejemplo:
- `familia_stark(X, Y) :- familia(X, Y), stark(X), stark(Y).`
- Esta regla se lee como: "Si X e Y son de la misma familia y X es un Stark e Y es un Stark, entonces, X e Y pertenecen a la familia Stark".
- Solo podemos incluir una condición AND en el antecedente, al igual que en programación estructurada.

Como se codifica un OR en Prolog?

- Prolog permite incluir el operador OR mediante la inclusión de varias reglas con el mismo consecuente.

Como se codifica un OR en Prolog?

- Prolog permite incluir el operador OR mediante la inclusión de varias reglas con el mismo consecuente.
- ```
  familia_stark("eddard", "jon").
  familia_ASOIAF(X, Y) :- familia_mormont(X, Y).
  familia_ASOIAF(X, Y) :- familia_lannister(X,
  Y).
  familia_ASOIAF(X, Y) :- familia_stark(X, Y).
  ```
  ```

Como se codifica un OR en Prolog?

- Prolog permite incluir el operador OR mediante la inclusión de varias reglas con el mismo consecuente.
- ```
  familia_stark("eddard", "jon").
  familia_ASOIAF(X, Y) :- familia_mormont(X, Y).
  familia_ASOIAF(X, Y) :- familia_lannister(X,
  Y).
  familia_ASOIAF(X, Y) :- familia_stark(X, Y).
  ```
- Dado un fichero de hechos y reglas: **Prolog lo ejecuta de arriba a abajo, en cada línea de izquierda a derecha**, ejecutando el algoritmo de unificación.
  ```

Como se codifica un OR en Prolog?

- Prolog permite incluir el operador OR mediante la inclusión de varias reglas con el mismo consecuente.
- ```
  familia_stark("eddard", "jon").
  familia_ASOIAF(X, Y) :- familia_mormont(X, Y).
  familia_ASOIAF(X, Y) :- familia_lannister(X,
  Y).
  familia_ASOIAF(X, Y) :- familia_stark(X, Y).
  ```
- Dado un fichero de hechos y reglas: **Prolog lo ejecuta de arriba a abajo, en cada línea de izquierda a derecha**, ejecutando el algoritmo de unificación.
- Si algun antecedente de una regla no se cumple, Prolog pasa a la siguiente. Esa es la clave del OR.
  ```

Variables anónimas

- Imaginemos que queremos saber un solo nombre de una persona que pertenezca a una familia de ASOIAF. El otro miembro de la tupla no nos interesa:

Variables anónimas

- Imaginemos que queremos saber un solo nombre de una persona que pertenezca a una familia de ASOIAF. El otro miembro de la tupla no nos interesa:
- `:- familia_ASOIAF(X, _).`

Variables anónimas

- Imaginemos que queremos saber un solo nombre de una persona que pertenezca a una familia de ASOIAF. El otro miembro de la tupla no nos interesa:
- `:- familia_ASOIAF(X, _).`
- _ es un operador que describe una variable anónima, variable de la cual no queremos saber su valor.

Variables anónimas

- Imaginemos que queremos saber un solo nombre de una persona que pertenezca a una familia de ASOIAF. El otro miembro de la tupla no nos interesa:
- `:- familia_ASOIAF(X, _).`
- _ es un operador que describe una variable anónima, variable de la cual no queremos saber su valor.
- Dada esta consulta, Prolog lee el hecho `familia_stark("eddard", "jon").` Prolog sabe que `familia_stark("eddard", "jon")` es cierto.

Variables anónimas

- Imaginemos que queremos saber un solo nombre de una persona que pertenezca a una familia de ASOIAF. El otro miembro de la tupla no nos interesa:
- `:- familia_ASOIAF(X, _).`
- _ es un operador que describe una variable anónima, variable de la cual no queremos saber su valor.
- Dada esta consulta, Prolog lee el hecho `familia_stark("eddard", "jon").` Prolog sabe que `familia_stark("eddard", "jon")` es cierto.
- Ejecuta la siguiente línea `familia_ASOIAF(X, Y) :- familia_mormont(X, Y).` No hay hechos `familia_mormont(X, Y)` luego pasa a la siguiente instrucción.

Variables anónimas

- Imaginemos que queremos saber un solo nombre de una persona que pertenezca a una familia de ASOIAF. El otro miembro de la tupla no nos interesa:
- `:- familia_ASOIAF(X, _).`
- _ es un operador que describe una variable anónima, variable de la cual no queremos saber su valor.
- Dada esta consulta, Prolog lee el hecho `familia_stark("eddard", "jon").` Prolog sabe que `familia_stark("eddard", "jon")` es cierto.
- Ejecuta la siguiente línea `familia_ASOIAF(X, Y) :- familia_mormont(X, Y).` No hay hechos `familia_mormont(X, Y)` luego pasa a la siguiente instrucción.
- Cuando llega a `familia_ASOIAF(X, Y) :- familia_stark(X, Y)` unifica las variable `X="eddard"`, validando `familia_stark(X,Y)`. Luego entonces el antecedente `familia_ASOIAF(X, Y)` es cierto y es una respuesta a la consulta u objetivo `:- familia_ASOIAF(X, _).`

Resumiendo AND y OR

- Prolog por tanto resuelve que X="brandon".

Resumiendo AND y OR

- Prolog por tanto resuelve que X="brandon".
- Si observamos, al programar varias reglas con el mismo consecuente, hemos hecho un OR de los antecedentes.

Resumiendo AND y OR

- Prolog por tanto resuelve que X="brandon".
- Si observamos, al programar varias reglas con el mismo consecuente, hemos hecho un OR de los antecedentes.
- El consecuente debe ser el mismo.

Resumiendo AND y OR

- Prolog por tanto resuelve que X="brandon".
- Si observamos, al programar varias reglas con el mismo consecuente, hemos hecho un OR de los antecedentes.
- El consecuente debe ser el mismo.
- Repasando, se programa un AND con hechos enlazados por comas en los antecedentes y un OR con varias reglas con el mismo consecuente.

Bucles en Prolog: Recursión

- Supongamos que queremos codificar la regla abuelo, podemos hacerlo con varias reglas padre:

Bucles en Prolog: Recursión

- Supongamos que queremos codificar la regla abuelo, podemos hacerlo con varias reglas padre:
- `grandfather(X, Y) :- father(X, Z), father(Z, Y)..`

Bucles en Prolog: Recursión

- Supongamos que queremos codificar la regla abuelo, podemos hacerlo con varias reglas padre:
- `grandfather(X, Y) :- father(X, Z), father(Z, Y).`.
- A veces, una relación involucra una secuencia indefinida de relaciones entre hechos.

Bucles en Prolog: Recursión

- Supongamos que queremos codificar la regla abuelo, podemos hacerlo con varias reglas padre:
- `grandfather(X, Y) :- father(X, Z), father(Z, Y).`.
- A veces, una relación involucra una secuencia indefinida de relaciones entre hechos.
- Pongamos un ejemplo. Consideremos la relación `ancestor(X, Y).`

Bucles en Prolog: Recursión

- Supongamos que queremos codificar la regla abuelo, podemos hacerlo con varias reglas padre:
- `grandfather(X, Y) :- father(X, Z), father(Z, Y).`.
- A veces, una relación involucra una secuencia indefinida de relaciones entre hechos.
- Pongamos un ejemplo. Consideremos la relación `ancestor(X, Y).`
- El miembro del clan X es antepasado del miembro Y si una secuencia de longitud indefinida de relaciones `parent(X, Z), parent(Z, A), ..., parent(K, Y)`, es verdadera.

Bucles en Prolog: Recursión

- Supongamos que queremos codificar la regla abuelo, podemos hacerlo con varias reglas padre:
- `grandfather(X, Y) :- father(X, Z), father(Z, Y).`.
- A veces, una relación involucra una secuencia indefinida de relaciones entre hechos.
- Pongamos un ejemplo. Consideremos la relación `ancestor(X, Y).`
- El miembro del clan X es antepasado del miembro Y si una secuencia de longitud indefinida de relaciones `parent(X, Z), parent(Z, A), ..., parent(K, Y)`, es verdadera.
- Con las herramientas vistas hasta ahora, no podemos definir dicha secuencia, pues su longitud es indefinida.

Recursión

- Para elaborar un bucle de evaluación de relaciones, podemos emplear la recursión.

Recursión

- Para elaborar un bucle de evaluación de relaciones, podemos emplear la recursión.
- En la recursión diferenciamos entre caso base, o condición que pone fin a la recursión y paso de la recursión.

Recursión

- Para elaborar un bucle de evaluación de relaciones, podemos emplear la recursión.
- En la recursión diferenciamos entre caso base, o condición que pone fin a la recursión y paso de la recursión.
- En este caso, el caso base es que dos términos compartiesen la relación parent.

Recursión

- Para elaborar un bucle de evaluación de relaciones, podemos emplear la recursión.
- En la recursión diferenciamos entre caso base, o condición que pone fin a la recursión y paso de la recursión.
- En este caso, el caso base es que dos términos compartiesen la relación parent.
- En este caso, hemos encontrado un enlace en la cadena recursiva, y volvemos.

Recursión

- Para elaborar un bucle de evaluación de relaciones, podemos emplear la recursión.
- En la recursión diferenciamos entre caso base, o condición que pone fin a la recursión y paso de la recursión.
- En este caso, el caso base es que dos términos compartiesen la relación parent.
- En este caso, hemos encontrado un enlace en la cadena recursiva, y volvemos.
- El paso de la recursión es encontrar un nuevo término que comparta la relación parent con el término actual y volver a llamar a la recursión.

Recursión

- Estas dos condiciones se pueden expresar por dos relaciones unidas por el operador OR. X es ancestro de Z si:

Recursión

- Estas dos condiciones se pueden expresar por dos relaciones unidas por el operador OR. X es ancestro de Z si:
- `parent(X, Z).`

Recursión

- Estas dos condiciones se pueden expresar por dos relaciones unidas por el operador OR. X es ancestro de Z si:
- `parent(X, Z).`
- `parent(X, Y), ancestor(Y, Z).`

Recursión

- Estas dos condiciones se pueden expresar por dos relaciones unidas por el operador OR. X es ancestro de Z si:
- `parent(X, Z).`
- `parent(X, Y), ancestor(Y, Z).`
- **Es importante poner arriba el caso base, sino, se entra en un bucle infinito!**

Listas

- Hasta ahora hemos trabajado con términos que comparten relaciones.

Listas

- Hasta ahora hemos trabajado con términos que comparten relaciones.
- En ocasiones un conjunto potencialmente indeterminado de elementos comparten relaciones.

Listas

- Hasta ahora hemos trabajado con términos que comparten relaciones.
- En ocasiones un conjunto potencialmente indeterminado de elementos comparten relaciones.
- Para representar un número de términos que comparten propiedades usamos listas.

Listas

- Hasta ahora hemos trabajado con términos que comparten relaciones.
- En ocasiones un conjunto potencialmente indeterminado de elementos comparten relaciones.
- Para representar un número de términos que comparten propiedades usamos listas.
- `[naranja, platano, pera].`

Listas

- Hasta ahora hemos trabajado con términos que comparten relaciones.
- En ocasiones un conjunto potencialmente indeterminado de elementos comparten relaciones.
- Para representar un número de términos que comparten propiedades usamos listas.
- `[naranja, platano, pera].`
- Una lista vacía es una lista: `[]`.

Listas

- Hasta ahora hemos trabajado con términos que comparten relaciones.
- En ocasiones un conjunto potencialmente indeterminado de elementos comparten relaciones.
- Para representar un número de términos que comparten propiedades usamos listas.
- `[naranja, platano, pera].`
- Una lista vacía es una lista: `[].`
- Una lista puede tener elementos o listas: `[[x,y,[z,k]],[a,b,[[[m]]]]].`

Listas

- Hasta ahora hemos trabajado con términos que comparten relaciones.
- En ocasiones un conjunto potencialmente indeterminado de elementos comparten relaciones.
- Para representar un número de términos que comparten propiedades usamos listas.
- `[naranja, platano, pera].`
- Una lista vacía es una lista: `[]`.
- Una lista puede tener elementos o listas: `[[x,y,[z,k]],[a,b,[[[m]]]]].`
- Puede tener también relaciones: `[comida(pasta, pescado), familia(pepe, pedro)]`

Recorrer listas

- Podemos direccionar los elementos de una lista con el operador | unificando en variables cabeza y cola:

Recorrer listas

- Podemos direccionar los elementos de una lista con el operador | unificando en variables cabeza y cola:
- `[Cabeza|Cola].`

Recorrer listas

- Podemos direccionar los elementos de una lista con el operador | unificando en variables cabeza y cola:
- `[Cabeza|Cola].`
- `[1,2,3,4]` → `Cabeza==1, Cola==[2,3,4].`

Recorrer listas

- Podemos direccionar los elementos de una lista con el operador | unificando en variables cabeza y cola:
- `[Cabeza|Cola].`
- `[1,2,3,4]` → `Cabeza==1, Cola==[2,3,4].`
- Si la lista tiene un elemento `[x]` → `Cabeza==x, Cola==[].`

Recorrer listas

- Podemos direccionar los elementos de una lista con el operador | unificando en variables cabeza y cola:
- `[Cabeza|Cola].`
- `[1,2,3,4]` → `Cabeza==1, Cola==[2,3,4].`
- Si la lista tiene un elemento `[x]` → `Cabeza==x, Cola==[].`
- La lista vacía no puede ser dividida en cabeza y cola.

Recorrer listas

- Podemos direccionar los elementos de una lista con el operador | unificando en variables cabeza y cola:
- `[Cabeza|Cola].`
- `[1,2,3,4]` → `Cabeza==1, Cola==[2,3,4].`
- Si la lista tiene un elemento `[x]` → `Cabeza==x, Cola==[].`
- La lista vacía no puede ser dividida en cabeza y cola.
- Util para recursión! La lista se puede expresar como `L=[1,2,3,4]==[1|[2,3,4]]==`
  `==[1|[2|[3,4]]]==[1|[2|[3|[4]]]]==[1|[2|[3|[4|[]`

Recorrer listas

- Podemos direccionar los elementos de una lista con el operador | unificando en variables cabeza y cola:
- `[Cabeza|Cola].`
- `[1,2,3,4]` → `Cabeza==1, Cola==[2,3,4].`
- Si la lista tiene un elemento `[x]` → `Cabeza==x, Cola==[].`
- La lista vacía no puede ser dividida en cabeza y cola.
- Util para recursión! La lista se puede expresar como `L=[1,2,3,4]==[1|[2,3,4]]==`
  `==[1|[2|[3,4]]]==[1|[2|[3|[4]]]]==[1|[2|[3|[4|[]`
- `[F,S|R] to F==1, S==2, R==[3,4]`

Ejemplo de recursión. Operador is.

- Programemos una función factorial. El factorial de un número es dado por el producto de todos los números inferiores a él.

Ejemplo de recursión. Operador is.

- Programemos una función factorial. El factorial de un número es dado por el producto de todos los números inferiores a él.
- 5!=5*4*3*2*1 = 120.

Ejemplo de recursión. Operador is.

- Programemos una función factorial. El factorial de un número es dado por el producto de todos los números inferiores a él.
- 5!=5*4*3*2*1 = 120.
- El caso base es 0!=1.

Ejemplo de recursión. Operador is.

- Programemos una función factorial. El factorial de un número es dado por el producto de todos los números inferiores a él.
- 5!=5*4*3*2*1 = 120.
- El caso base es 0!=1.
- Pues lo escribimos: `fact(0,1).`

Ejemplo de recursión. Operador is.

- Programemos una función factorial. El factorial de un número es dado por el producto de todos los números inferiores a él.
- 5!=5*4*3*2*1 = 120.
- El caso base es 0!=1.
- Pues lo escribimos: `fact(0,1).`
- En el caso recursivo relacionamos el factorial del número actual con el factorial del número anterior.

Ejemplo de recursión. Operador is.

- Programemos una función factorial. El factorial de un número es dado por el producto de todos los números inferiores a él.
- 5!=5*4*3*2*1 = 120.
- El caso base es 0!=1.
- Pues lo escribimos: `fact(0,1).`
- En el caso recursivo relacionamos el factorial del número actual con el factorial del número anterior.
- Pues lo escribimos: `fact(X,Y) :- Z is X-1, fact(Z,Q), Y is Q*X.`

Ejemplo de recursión. Operador is.

- Programemos una función factorial. El factorial de un número es dado por el producto de todos los números inferiores a él.
- 5!=5*4*3*2*1 = 120.
- El caso base es 0!=1.
- Pues lo escribimos: `fact(0,1).`
- En el caso recursivo relacionamos el factorial del número actual con el factorial del número anterior.
- Pues lo escribimos: `fact(X,Y) :- Z is X-1, fact(Z,Q), Y is Q*X.`
- Dos líneas solo lo consiguen. (Sin control de errores) Pruébalo!

Ejemplo de recursión. Evitar recursión infinita

- ```
  fact(0,1).
  fact(X,Y) :- Z is X-1, fact(Z,Q), Y is Q*X.
  ```
  ```

Ejemplo de recursión. Evitar recursión infinita

- `fact(0,1).`
  `fact(X,Y) :- Z is X-1, fact(Z,Q), Y is Q*X.`
- Podemos realizar una consulta: `fact(5,N).`
  `N=120`

Ejemplo de recursión. Evitar recursión infinita

- `fact(0,1).`
  `fact(X,Y) :- Z is X-1, fact(Z,Q), Y is Q*X.`
- Podemos realizar una consulta: `fact(5,N).`
  `N=120`
- Si pedimos al interprete que devuelva otra solución, entra en recursión infinita.

Ejemplo de recursión. Evitar recursión infinita

- ```
  fact(0,1).
  fact(X,Y) :- Z is X-1, fact(Z,Q), Y is Q*X.
  ```
- Podemos realizar una consulta: `fact(5,N).`
  ```
  N=120
  ```
- Si pedimos al interprete que devuelva otra solución, entra en recursión infinita.
- Una posible solución es incluir una condición de positividad:
  ```
  fact(0,1).
  fact(X,Y) :- X > 0, Z is X-1, fact(Z,Q), Y
  is Q*X.
  ```
  ```

Ejemplo de recursión. Evitar recursión infinita

- ```
  fact(0,1).
  fact(X,Y) :- Z is X-1, fact(Z,Q), Y is Q*X.
  ```
- Podemos realizar una consulta: `fact(5,N).`
  `N=120`
- Si pedimos al interprete que devuelva otra solución, entra en recursión infinita.
- Una posible solución es incluir una condición de positividad:
  ```
  fact(0,1).
  fact(X,Y) :- X > 0, Z is X-1, fact(Z,Q), Y
  is Q*X.
  ```
- Otra opción, usando el operador `!/0` para cortar en el caso base:
  ```
  fact(0,1) :-!.
  fact(X,Y) :- Z is X-1, fact(Z,Q), Y is Q*X.
  ```
  ```

Ejemplo de recursión: is y unificación

- ```
  fact(0,1).
  fact(X,Y) :- fact(X-1,Q), Y is Q*X.
  ```
  ```

Ejemplo de recursión: is y unificación

- ```
  fact(0,1).
  fact(X,Y) :- fact(X-1,Q), Y is Q*X.
  ```
- Este ejemplo no funciona. ¿Porqué?
  ```

Ejemplo de recursión: is y unificación

- ```
  fact(0,1).
  fact(X,Y) :- fact(X-1,Q), Y is Q*X.
  ```
- Este ejemplo no funciona. ¿Porqué?
- Si no utilizamos el operador `is`, la expresión `X-1` queda como una expresión simbólica y la operación aritmética no se realiza.
  ```

Ejemplo de recursión: is y unificación

- ```
  fact(0,1).
  fact(X,Y) :- fact(X-1,Q), Y is Q*X.
  ```
- Este ejemplo no funciona. ¿Porqué?
- Si no utilizamos el operador `is`, la expresión `X-1` queda como una expresión simbólica y la operación aritmética no se realiza.
- Otro ejemplo:

  ```
  temp(X,Y) :- Y = X-1.
  ```
  ```

Ejemplo de recursión: is y unificación

- ```
  fact(0,1).
  fact(X,Y) :- fact(X-1,Q), Y is Q*X.
  ```
- Este ejemplo no funciona. ¿Porqué?
- Si no utilizamos el operador `is`, la expresión `X-1` queda como una expresión simbólica y la operación aritmética no se realiza.
- Otro ejemplo:

  ```
  temp(X,Y) :- Y = X-1.
  ```
- ```
  ?- temp(2,Y).
  ```

Ejemplo de recursión: is y unificación

- ```
  fact(0,1).
  fact(X,Y) :- fact(X-1,Q), Y is Q*X.
  ```
- Este ejemplo no funciona. ¿Porqué?
- Si no utilizamos el operador `is`, la expresión `X-1` queda como una expresión simbólica y la operación aritmética no se realiza.
- Otro ejemplo:

  ```
  temp(X,Y) :- Y = X-1.
  ```
- ```
  ?- temp(2,Y).
  ```
- ```
  Y = 2-1
  ```
  ```

Ejemplo de recursión: is y unificación

- ```
  fact(0,1).
  fact(X,Y) :- fact(X-1,Q), Y is Q*X.
  ```
- Este ejemplo no funciona. ¿Porqué?
- Si no utilizamos el operador `is`, la expresión `X-1` queda como una expresión simbólica y la operación aritmética no se realiza.
- Otro ejemplo:
  ```
  temp(X,Y) :- Y = X-1.
  ```
- `?- temp(2,Y).`
- `Y = 2-1`
- Se ha realizado la unificación entre `Y` y `2-1`, sin realizar la operación aritmética.
  ```

Ejemplo de recursión: is y unificación

- ```
  fact(0,1).
  fact(X,Y) :- fact(X-1,Q), Y is Q*X.
  ```
- Este ejemplo no funciona. ¿Porqué?
- Si no utilizamos el operador `is`, la expresión `X-1` queda como una expresión simbólica y la operación aritmética no se realiza.
- Otro ejemplo:
  ```
  temp(X,Y) :- Y = X-1.
  ```
- `?- temp(2,Y).`
- `Y = 2-1`
- Se ha realizado la unificación entre `Y` y `2-1`, sin realizar la operación aritmética.
- El predicado `=/2` indica comparación o unificación, no asignación como en los lenguajes estructurados.
  ```

Ejemplo de recorrido de listas. Operador is.

- Supongamos ahora que recibimos una lista X como argumento y queremos programar el productorio $\prod_{i=1}^{N} x_i : x_i \in X$.

Ejemplo de recorrido de listas. Operador is.

- Supongamos ahora que recibimos una lista X como argumento y queremos programar el productorio $\prod_{i=1}^{N} x_i : x_i \in X$.
- El elemento neutro de la multiplicación es el 1. Luego el caso base es que el productorio de una lista vacía es 1.

Ejemplo de recorrido de listas. Operador is.

- Supongamos ahora que recibimos una lista X como argumento y queremos programar el productorio $\prod_{i=1}^{N} x_i : x_i \in X$.
- El elemento neutro de la multiplicación es el 1. Luego el caso base es que el productorio de una lista vacía es 1.
- Pues lo ponemos: `prod([], 1).`

Ejemplo de recorrido de listas. Operador is.

- Supongamos ahora que recibimos una lista X como argumento y queremos programar el productorio $\prod_{i=1}^{N} x_i : x_i \in X$.
- El elemento neutro de la multiplicación es el 1. Luego el caso base es que el productorio de una lista vacía es 1.
- Pues lo ponemos: `prod([], 1).`
- Por otro lado debemos declarar una variable acumulativa del producto de cada elemento de la lista por lo acumulado.

Ejemplo de recorrido de listas. Operador is.

- Supongamos ahora que recibimos una lista X como argumento y queremos programar el productorio $\prod_{i=1}^{N} x_i : x_i \in X$.
- El elemento neutro de la multiplicación es el 1. Luego el caso base es que el productorio de una lista vacía es 1.
- Pues lo ponemos: `prod([], 1).`
- Por otro lado debemos declarar una variable acumulativa del producto de cada elemento de la lista por lo acumulado.
- Usamos el operador | para ello:

Ejemplo de recorrido de listas. Operador is.

- Supongamos ahora que recibimos una lista X como argumento y queremos programar el productorio $\prod_{i=1}^{N} x_i : x_i \in X$.
- El elemento neutro de la multiplicación es el 1. Luego el caso base es que el productorio de una lista vacía es 1.
- Pues lo ponemos: `prod([], 1).`
- Por otro lado debemos declarar una variable acumulativa del producto de cada elemento de la lista por lo acumulado.
- Usamos el operador | para ello:
- `prod([X|Y], Z) :- prod(Y, K), Z is X*K.`

Ejemplo de recorrido de listas. Operador is.

- Supongamos ahora que recibimos una lista X como argumento y queremos programar el productorio $\prod_{i=1}^{N} x_i : x_i \in X$.
- El elemento neutro de la multiplicación es el 1. Luego el caso base es que el productorio de una lista vacía es 1.
- Pues lo ponemos: `prod([], 1).`
- Por otro lado debemos declarar una variable acumulativa del producto de cada elemento de la lista por lo acumulado.
- Usamos el operador | para ello:
- `prod([X|Y], Z) :- prod(Y, K), Z is X*K.`
- En la línea anterior iteramos hasta la lista vacía y luego vamos acumulando en la variable de salida cada elemento.

Cortes

- Prolog siempre se queda esperando a dar mas respuestas.

Cortes

- Prolog siempre se queda esperando a dar mas respuestas.
- Si tras satisfacer relaciones queremos interrumpir ese comportamiento usamos el operador corte !.

Cortes

- Prolog siempre se queda esperando a dar mas respuestas.
- Si tras satisfacer relaciones queremos interrumpir ese comportamiento usamos el operador corte !.
- Si en el factorial no ponemos `fact(0,1):-!.` en el caso base, Prolog al intentar realizar todas las unificaciones posibles generará más consultas.

Cortes

- Prolog siempre se queda esperando a dar mas respuestas.
- Si tras satisfacer relaciones queremos interrumpir ese comportamiento usamos el operador corte !.
- Si en el factorial no ponemos `fact(0,1):-!.` en el caso base, Prolog al intentar realizar todas las unificaciones posibles generará más consultas.
- Con el corte interrumpimos ese comportamiento.

Cortes

- Prolog siempre se queda esperando a dar mas respuestas.
- Si tras satisfacer relaciones queremos interrumpir ese comportamiento usamos el operador corte !.
- Si en el factorial no ponemos `fact(0,1):-!.` en el caso base, Prolog al intentar realizar todas las unificaciones posibles generará más consultas.
- Con el corte interrumpimos ese comportamiento.
- En una situación normal, para buscar mas unificaciones, usamos el operador ; en el intérprete de Prolog.

Gestión de errores, trazas.

- En algunas ocasiones, queremos programar el fracaso intencionado de una consulta.

Gestión de errores, trazas.

- En algunas ocasiones, queremos programar el fracaso intencionado de una consulta.
- Por ejemplo, en controles de error, queremos que si se introduce un numero negativo el programa falle.

Gestión de errores, trazas.

- En algunas ocasiones, queremos programar el fracaso intencionado de una consulta.
- Por ejemplo, en controles de error, queremos que si se introduce un numero negativo el programa falle.
- Podemos forzar esta situación con el operador `fail` en Prolog.

Gestión de errores, trazas.

- En algunas ocasiones, queremos programar el fracaso intencionado de una consulta.
- Por ejemplo, en controles de error, queremos que si se introduce un numero negativo el programa falle.
- Podemos forzar esta situación con el operador `fail` en Prolog.
- Si `fail` es procesado, la consulta fallara.

Gestión de errores, trazas.

- En algunas ocasiones, queremos programar el fracaso intencionado de una consulta.
- Por ejemplo, en controles de error, queremos que si se introduce un numero negativo el programa falle.
- Podemos forzar esta situación con el operador `fail` en Prolog.
- Si `fail` es procesado, la consulta fallara.
- Es útil usar la secuencia: `print('error'), !, fail.` La negación por fallo.

Gestión de errores, trazas.

- En algunas ocasiones, queremos programar el fracaso intencionado de una consulta.
- Por ejemplo, en controles de error, queremos que si se introduce un numero negativo el programa falle.
- Podemos forzar esta situación con el operador `fail` en Prolog.
- Si `fail` es procesado, la consulta fallara.
- Es útil usar la secuencia: `print('error'), !, fail.` La negación por fallo.
- `print` informa al usuario del error y hace que el intérprete no pregunte por mas soluciones.

Gestión de errores, trazas.

- En algunas ocasiones, queremos programar el fracaso intencionado de una consulta.
- Por ejemplo, en controles de error, queremos que si se introduce un numero negativo el programa falle.
- Podemos forzar esta situación con el operador `fail` en Prolog.
- Si `fail` es procesado, la consulta fallara.
- Es útil usar la secuencia: `print('error'), !, fail.` La negación por fallo.
- `print` informa al usuario del error y hace que el intérprete no pregunte por mas soluciones.
- Puedes introducir control de errores como condiciones en un OR que se verifican al principio.

Gestión de errores, trazas.

- En algunas ocasiones, queremos programar el fracaso intencionado de una consulta.
- Por ejemplo, en controles de error, queremos que si se introduce un numero negativo el programa falle.
- Podemos forzar esta situación con el operador `fail` en Prolog.
- Si `fail` es procesado, la consulta fallara.
- Es útil usar la secuencia: `print('error'), !, fail.` La negación por fallo.
- `print` informa al usuario del error y hace que el intérprete no pregunte por mas soluciones.
- Puedes introducir control de errores como condiciones en un OR que se verifican al principio.
- Puedes usar `print` para imprimir por terminal información.

Swipl

- Interprete de SWI-Prolog en linea de comandos invocado por `swipl.`

Swipl

- Interprete de SWI-Prolog en linea de comandos invocado por `swipl.`
- Para cargar una base de conocimiento hacemos `consult('prolog_file.pl').`

Swipl

- Interprete de SWI-Prolog en linea de comandos invocado por `swipl.`
- Para cargar una base de conocimiento hacemos `consult('prolog_file.pl').`
- Para detener el intérprete podemos hacer `halt.`

Swipl

- Interprete de SWI-Prolog en linea de comandos invocado por `swipl.`
- Para cargar una base de conocimiento hacemos `consult('prolog_file.pl').`
- Para detener el intérprete podemos hacer `halt.`
- Para debuggear Prolog se puede invocar a `trace.`

Swipl

- Interprete de SWI-Prolog en linea de comandos invocado por `swipl.`
- Para cargar una base de conocimiento hacemos `consult('prolog_file.pl').`
- Para detener el intérprete podemos hacer `halt.`
- Para debuggear Prolog se puede invocar a `trace.`
- `trace` es extremadamente útil, ilustrándonos paso por paso la ejecución del intérprete de Prolog.

Swipl

- Interprete de SWI-Prolog en linea de comandos invocado por `swipl.`
- Para cargar una base de conocimiento hacemos `consult('prolog_file.pl').`
- Para detener el intérprete podemos hacer `halt.`
- Para debuggear Prolog se puede invocar a `trace.`
- `trace` es extremadamente útil, ilustrándonos paso por paso la ejecución del intérprete de Prolog.
- Se puede usar también la interfaz web de Prolog SWISH.

Trace, ejemplo

```
[trace]  ?- prod([6,7,8,9],X).
   Call: (8) prod([6, 7, 8, 9], _11206) ? creep
   Call: (9) prod([7, 8, 9], _11454) ? creep
   Call: (10) prod([8, 9], _11454) ? creep
   Call: (11) prod([9], _11454) ? creep
   Call: (12) prod([], _11454) ? creep
   Exit: (12) prod([], 1) ? creep
   Call: (12) _11458 is 9*1 ? creep
   Exit: (12) 9 is 9*1 ? creep
   Exit: (11) prod([9], 9) ? creep
   Call: (11) _11464 is 8*9 ? creep
   Exit: (11) 72 is 8*9 ? creep
   Exit: (10) prod([8, 9], 72) ? creep
   Call: (10) _11470 is 7*72 ? creep
   Exit: (10) 504 is 7*72 ? creep
   Exit: (9) prod([7, 8, 9], 504) ? creep
   Call: (9) _11206 is 6*504 ? creep
   Exit: (9) 3024 is 6*504 ? creep
   Exit: (8) prod([6, 7, 8, 9], 3024) ? creep
X = 3024.
```