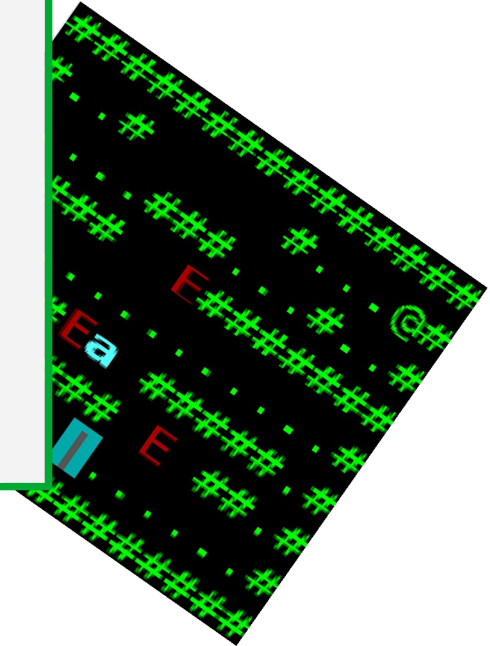
Vamos a aprender…...

Prolog

SWI Prolog



Introducción PL-MAN

http://bit.ly/Matematicas1

SEGUIMOS CON EL JUEGO…

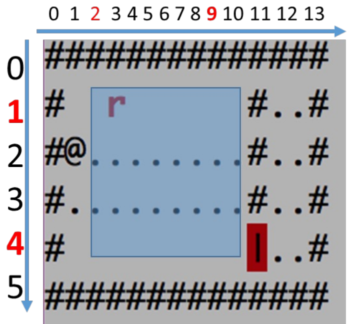
Fase 3

- Mapas con comportamientos ¨cambiantes¨: indeterminismo.
- Ejecutar el código n-veces con el script **launch.sh**
- Declaración de hechos dinámicos.
- Modulariza el código con diferentes estados
- Usaremos en la explicación los mapas :

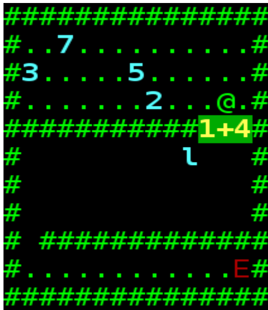
  plman/maps/fase3/**mapa0.pl**, **mapa6.pl**

  plman/maps/fase3/**mapa4.pl**

- Resuelve otros mapas de entrenamiento de la colección plman/maps/fase3



maps/fase3/mapa0.pl



maps/fase3/mapa6.pl

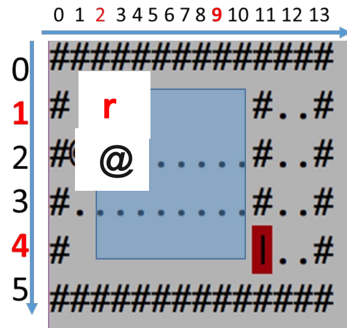
En los mapas de las fases 3 y 4 los objetos pueden

cambiar

o bien de **posición** (llave ’r’, pistola ‘l’, números…)

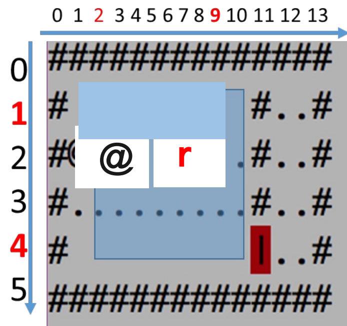
o bien de **comportamiento** (operación aritmética)

maps/fase3/mapa0.pl



PIman ve y coge la llave que se encuentra en una posición inmediata. En este ejemplo sería arriba:

do(get(up)) :- s(up, r) (1)



Tenemos que añadir más reglas y al salir y volver a entrar en el mapa la llave ha cambiado de posición

do(get(????)) :- s(?????, r) (2)

La regla (1) no sirve…
La regla (2) la podemos completar con la nueva posición de 'r', pero esto no nos garantiza que sea la dirección correcta en otra ejecución del mapa.
Que hacemos ???

maps/fase3/mapa0.pl

Descarga maps/**fase3/mapa0.pl** y ejecuta este código **(1) (sol1.pl).**
Si la llave 'r' está arriba, funciona…

```
do(ACT) :- not(havingObject),  do1(ACT).
do(ACT) :- havingObject,        do2(ACT).

do1(get(up))          :- s(up,r).
do1(move(up))         :- s(up, '.').
do1(move(down))       :- s(down, '.').
do1(move(right))      :- s(right, '.').
do1(move(right))      :- s(right, ' ').

do2(use(right)) :- s(right, '|').
do2(move(up))        :- s(up, '.').
do2(move(down))      :- s(down, '.').
do2(move(right))     :- s(right, '.').
do2(move(right))     :- s(right, ' ').
do2(move(down)).
```

Si vemos 'r' en otra posición (inmediata, claro) código (1) falla. Cambiamos código (1) por este código **(2) (sol2.pl).**

```
do(ACT) :- not(havingObject), do1(ACT).
do(ACT) :- havingObject,        do2(ACT).

dir(up).
dir(down).
dir(left).
dir(right).

do1(get(D))      :- dir(D), s(D, r).
…

do2(use(right)) :- s(right, '|').
…
```

- En mapas indeterministas, con elementos “cambiantes”, debemos comprobar que la solución funciona, **siempre.**
- Para ello usamos el script **launch.sh** con el que podemos **ejecutar una solución n-veces**
- **launch.sh** guardará en la carpeta plman/**logs** las soluciones que **fallan**.

… plman **$ ./launch n mapa.pl solucion.pl**

Si quieres **reproducir** un log de una ejecución que falla, por ejemplo la “x” , escribe:

… plman **$ ./plman -r logs/log_ejecución_x**

Al reproducir el log de una ejecución anterior podemos ver paso a paso lo ocurrido.

Teclas en la reproducción:
P: Plman avanza; **O:** Plman retrocede;
1-9 se cambia la velocidad de reproducción.
ESC para salir.

- launch.sh también se puede usar en mapas deterministas

Ejemplo de cómo se utiliza el **script launch.sh**

Ejecutamos **10 veces** la solución del mapa maps/fase3/**map0.pl** implementada en los códigos (1) y (2) con ficheros solución sol1.pl y sol2.pl, respectivamente

... plman **$ ./launch 10 mapa0.pl sol1.pl**

Reproducimos alguna ejecución de las que ha fallado (por ej., la 3):

... plman **$ ./plman -r logs/log_ejecución_3**

idem con sol2.pl

Teclas en la reproducción:
P: Plman avanza; **O:** Plman retrocede;
1-9 se cambia la velocidad de reproducción.
ESC para salir.

Los códigos (1) y (2) del mapa 0.pl se han modulado teniendo en cuenta el predicado "havingObject".

En fases 3 y 4 también podemos modular usando **hechos dinámicos**.

Consideraremos a cada conjunto de subreglas (cada módulo) como un **estado.**

Cambiaremos de estado cada vez que haya un cambio de comportamiento.

Para cambiar de estado usaremos los **predicados dinámicos** de Prolog.

Importante : en cada momento de la resolución tenemos que ¨controlar¨ el estado en que se **encuentra** Plman.

Introducimos este nuevo concepto resolviendo el mapa4.pl

Implementación del mapa plman/maps/fase3/**mapa4.pl** con **¨estados dinámicos¨**

En este mapa **el objeto ¨cambiante¨** es Plman

Estrategia para la resolución:

1º. En cada ejecución situar a Plman en la esquina inferior izquierda.

2º. Recorrer fila impar

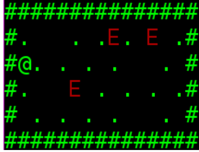
3º. Recorrer fila par

Implementaremos módulos (subreglas) para cada situación

Cada módulo será un **estado**

Código de plman/maps/fase3/**mapa4.pl** con ¨**estados dinámicos**¨

1º. En cada ejecución situar a Plman en la esquina inferior izquierda.

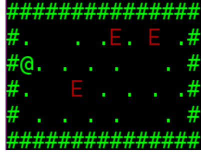



```
:-use_module("pl-man-game/main").

% situación inicial, en este estado movemos a Plman a esquina_izda

estado(esquina_izda).

do(ACT) :- estado(esquina_izda), doEi(ACT).

doEi(move(none))  :- write('voy a esquina_izda').

…
```

Al incluir el **hecho** **estado(esquina_izda)** el predicado do/1 se ejecuta pq cumple todas las condiciones del cuerpo de la regla do/1.

Código de plman/maps/fase3/**mapa4.pl** con ¨**estados dinámicos**¨

2º. Recorrer fila impar

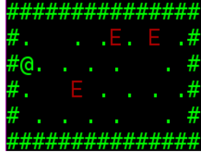



```
% Reglas para recorrer fila impar

estado(fila_impar).

do(ACT) :- estado(fila_impar), dofi(ACT).


dofi( move( none)) :- write('voy a esquina_izda').
...
```

Al incluir el **hecho estado(fila_impar)** el predicado do/1 se ejecuta pq cumple todas las condiciones del cuerpo de la regla.

Código de plman/maps/fase3/**mapa4.pl** con ¨**estados dinámicos**¨

3º. Recorrer fila par




```
% Reglas para recorrer fila par

estado(fila_par).

do(ACT) :- estado(fila_par), dofp(ACT).


dofp(move( none)) :- write('estoy en fila par').
…
```

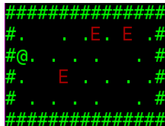
Al incluir el **hecho** **estado(fila_par)** el predicado do/1 se ejecuta pq cumple todas las condiciones del cuerpo de la regla.

Teniendo en cuenta cómo Prolog recorre código ¿ qué pasaría con esta implementación ?

Qué do/1 se ejecutaría (1), (2) o (3) ?

Si respondes (1) > CORRECTO, pero en qué casos crees que se ejecutaría (2) o (3) ?

```
:-use_module("pl-man-game/main").

(H1) estado(esquina_izda).
(H2) estado(fila_impar).
(H3) estado(fila_par).

(1) do(ACT) :- estado(esquina_izda), doEi(ACT).
(2) do(ACT) :- estado(fila_impar),    dofi(ACT).
(3) do(ACT) :- estado(fila_par),      dofp(ACT).

% Reglas para mover a Plman a esquina_izda
doEi(move(none))  :- write('voy a esquina_izda').
…
% Reglas para recorrer fila impar
dofi( move( none)) :- write('estoy en fila impar').

…
% Reglas para recorrer fila par
dofp(move( none)) :- write('estoy en fila par').
…
```



```
:-use_module("pl-man-game/main").

(H1) % estado(esquina_izda).
(H2) estado(fila_impar).
(H3) estado(fila_par).

(1) do(ACT) :- estado(esquina_izda), doEi(ACT).
(2) do(ACT) :- estado(fila_impar),     dofi(ACT).
(3) do(ACT) :- estado(fila_par),        dofp(ACT).

% Reglas para mover a Plman a
doEi(move(none))  :- write('voy a
...
% Reglas para recorrer fila impa
dofi( move( none)) :- write('estoy

...
% Reglas para recorrer fila par
dofp(move( none)) :- write('estoy en fila par').
...
```

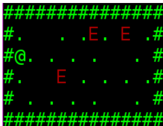
Borramos (H1) para pasar al hecho (H2) y ejecutar la regla (2)

Ahora qué **do/1 se ejecutaría (1), (2) o (3) ?**

Si respondes (1) > ERROR, en regla falla hecho (H1)

Si respondes (2) > OK, pero cuándo se ejecutaría (3) ?

Estamos en las mismas que antes ….sólo se ejecutará (3) si se quita (H2)



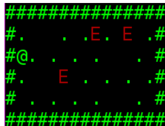
```
:-use_module("pl-man-game/main").

(H1) % estado(esquina_izda).
(H2) % estado(fila_impar).
(H3) estado(fila_par).

(1) do(ACT) :- estado(esquina_izda), doEi(ACT).
(2) do(ACT) :- estado(fila_impar),   dofi(ACT).
(3) do(ACT) :- estado(fila_par),     dofp(ACT).

% Reglas para mover a Plman a es
doEi(move(none)) :- write('voy a e
…
% Reglas para recorrer fila impar
dofi( move( none)) :- write('estoy en fila impar').

…
% Reglas para recorrer fila par
dofp(move( none)) :- write('estoy en fila par').
…
```

Borramos (H2) para pasar al hecho (H3) y ejecutar la regla (3)

Ahora qué **do/1 se ejecutaría (1), (2) o (3) ?**

Si respondes (1) o (2) > ERROR, en estas reglas fallan (H1) (H2)

Se ejecuta (3)



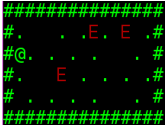
plman/maps/fase3/**mapa4.pl** con ¨**estados dinámicos**¨

```
:-use_module("pl-man-game/main").

(H1) % estado(esquina_izda).
(H2) % estado(fila_impar).
(H3) estado(fila_par).

(1) do(ACT) :- estado(esquina_izda), do
(2) do(ACT) :- estado(fila_impar),     d
(3) do(ACT) :- estado(fila_par),       do

% Reglas para mover a Plman a esquina_izda
doEi(move(none))  :- write('voy a esquina_izda').
…
% Reglas para recorrer fila impar
dofi( move( none)) :- write('estoy en fila impar').

…
% Reglas para recorrer fila par
dofp(move( none)) :- write('estoy en fila par').
…
```

Si queremos volver a ejecutar fila_impar…?

No podemos estar borrando/escribiendo hechos del programa de forma estática, para cada situación…

Prolog permite **borrar/añadir** hechos en **tiempo de ejecución** declarándolos como **dinámicos.**



Vamos escribir el programa paso a paso con **hechos dinámicos**

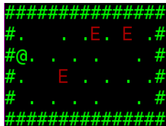
El predicado ¨estado/1¨ será el que indicará la situación en la que nos encontramos a partir del valor de su argumento.

Declaramos ¨estado/1¨como **dinámico**

Directiva de Prolog para declarar un predicado como dinámico:

:- dynamic predicado/n.

```
:-use_module("pl-man-game/main").

:- dynamic estado/1.

estado(esquina_izda).
estado(fila_impar).
estado(fila_par).

do(ACT) :- estado(esquina_izda), doEi(ACT).
do(ACT) :- estado(fila_impar),    dofi(ACT).
do(ACT) :- estado(fila_par),      dofp(ACT).

% Reglas para mover a Plman a esquina_izda
% Reglas para recorrer fila impar
% Reglas para recorrer fila par
```



plman/maps/fase3/**mapa4.pl** con ¨**estados dinámicos**¨

Para comenzar la ejecución ponemos un valor inicial al argumento de ¨estado/1¨.

Con este valor se ejecuta (1), que es la regla que tiene este hecho como condición.

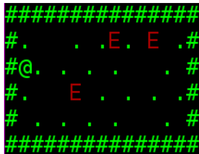
Se **activan** las subreglas doEi/1

Cuando se tiene que cambiar de situación, cambiamos de estado en las subreglas doEi

Si queremos que del estado(esquina_izda) pase al estado(fila_impar) …ver (4)

```
:-use_module("pl-man-game/main").

:- dynamic estado/1.

estado(esquina_izda).

(1) do(ACT) :- estado(esquina_izda),  doEi(ACT).
(2) do(ACT) :- estado(fila_impar),      dofi(ACT).
(3) do(ACT) :- estado(fila_par),          dofp(ACT).

% Reglas para mover a Plman a esquina_izda

doEi(move(none)) :- s(down,'#'), s(left, '#'),
(4)          retractall(estado(_)),
assert(estado(fila_impar)).

…

% Reglas para recorrer fila impar
% Reglas para recorrer fila par
```

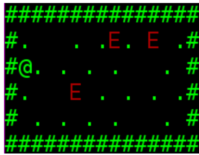
Sigue transparencia →



plman/maps/fase3/**mapa4.pl** con ¨**estados dinámicos**¨

La regla (4) permite que la ejecución pase a la regla (2) ya que la regla (1) fracasaría porque el hecho: estado(esquina_izda) no existe

```
:-use_module("pl-man-game/main").

:- dynamic estado/1.

estado(esquina_izda).   → se elimina

estado(esquina_izda).   → se añade


(1) do(ACT) :- estado(esquina_izda),  doEi(ACT).
(2) do(ACT) :- estado(fila_impar),     dofi(ACT).  → se ejecuta
(3) do(ACT) :- estado(fila_par),        dofp(ACT).

(4) % Reglas para mover a Plman a esquina_izda
doEi(move(none)) :- s(down,'#'), s(left, '#'),
        retractall(estado(_)), assert(estado(fila_impar)).
...
% Reglas para recorrer fila impar
.....
% Reglas para recorrer fila par
```



Sigue la explicación de estos predicados en las siguientes transparencias →

PREDICADOS DINÁMICOS DE PROLOG

Una característica muy útil de Prolog es la posibilidad de **añadir y eliminar** hechos o reglas (sólo veremos hechos) de los predicados presentes en el programa, y hacerlo en **tiempo de ejecución**.

Los predicados con esta posibilidad se denominan **predicados dinámicos**.

Se declaran mediante la directiva dynamic/1

:- dynamic predicado / n

Ejemplo declaramos el predicado ¨edad/1¨ como dinámico

:- dynamic edad/1.

>> Cuando usamos un predicado dinámico lo **que cambia es el valor de su argumento** en el transcurso de la ejecución del programa donde está declarado

PREDICADOS DINÁMICOS DE PROLOG

Predicados ISO_Standard de Prolog para **añadir /eliminar** hechos dinámicos son (entre otros):

Añadir hechos: assert/1

assert(hecho).

Ejemplo

assert(edad(23)).

Eliminar un hecho: retract/1

retract(hecho).

Ejemplo

retract(edad(23)).

Eliminar todos los hecho: retractall/1

retractall(hecho).

Ejemplo

retractall(edad(_)).

Elimina todos los hechos edad/1 con cualquier valor en el argumento

PREDICADOS DINÁMICOS DE PROLOG

Ejemplo Sigue los pasos indicados para que veas cómo funcionan estos predicados

- Abre SWI-Prolog, teclea **$ swipl**

Escribe

```
? dynamic edad/1.
```

Añade un hecho del predicado edad/1

```
?- assert(edad(23)).
```

Añade otros hechos del predicado edad/1

```
?- assert(edad(55)).
?- assert(edad(90)).
?- assert(edad(10)).
```

Vemos el contenido de TODA la base de conocimiento

```
?- listing.
```

PREDICADOS DINÁMICOS DE PROLOG

Elimina hechos existente en la BC

?- **retract**(edad(10)).

?- **retract**(edad(55)).

Intenta borrar un hecho inexistente ¿?

?- **retract**(edad(8)).

Borra el primero que encuentre

?- **retract**(edad(_)).

Mira lo que queda…

?- **listing**(edad).

Borra todos los hechos del predicado edad/1

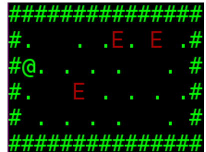
?- **retractall**(edad(_)).

plman/maps/fase3/**mapa4.pl** con ¨**estados dinámicos**¨

Programa (incompleto) del mapa4.pl declarando dinámico el predicado ¨estado/1¨

Hemos declarado una regla **change/1** que permite hacer el cambio de estado de tal forma que cada vez que queramos hacerlo incluimos en la subregla apropiada esta regla con el argumento del estado al que queremos acceder.

En este mapa cuando hemos considerado oportuno en las subreglas que recorre Plman para ir a esquina izda hemos cambiado al estado fila_impar




```
:-use_module("pl-man-game/main").

:- dynamic estado/1.

estado(esquina_izda).

change(Es) :- retractall(estado(_)), assert(estado(Es)).


do(ACT) :- estado(esquina_izda),  doEi(ACT).
do(ACT) :- estado(fila_impar),       dofi(ACT).
do(ACT) :- estado(fila_par),          dofp(ACT).

% Reglas para mover a Plman a esquina_izda

doEi(move(none)) :- s(down,'#'), s(left, '#'),
         change(fila_impar).

…
% Reglas para recorrer fila impar

dofi(move(none)) :- s(left, ' '),
           change(fila_par).
…
% Reglas para recorrer fila par
…
```

Descuidos al usar dynamic

```
dynamic estoy/1.
estoy(abajo).
```

```
:- dynamic estoy/2.
estoy(abajo).
```

```
%% dynamic no usado
estoy(abajo).
```

```
ERROR: retract/1: No permission to modify static procedure `estoy/1'
```

Forma correcta

```
:- dynamic estoy/1.
estoy(abajo).
```

Fallos al usar retract y assert

```
r(abajo)   :- retract(estoy),
              assert(estoy(arriba)).
```

```
1 ADVERTENCIA: La regla de control
de tu personaje ha fracasado!
```

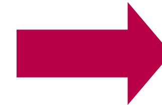
Se intenta borrar `estoy/0`, que no existe. `retract` fracasa.

```
r(abajo)   :- retract(estoy(arriba)),
              assert(estoy(arriba)).
```

```
1 ADVERTENCIA: La regla de control
de tu personaje ha fracasado!
```

Se intenta borrar `estoy(arriba)`, que no existe (existe `estoy(abajo)`). `retract` fracasa.

```
r(abajo)   :- assert(estoy(arriba)).
```



```
estoy(abajo).
estoy(arriba).
```

`assert` añade un hecho, pero no borra el anterior.
La Base de Conocimientos ahora tiene 2 hechos `estoy/1`.

Comprobar fallos escribiendo "mensajes" con write/1 y writeln/1

```
r :- estoy(E),
     write('Estoy '),
     writeln(E),
     r(E).
```

```
Estoy abajo
Estoy abajo
Estoy abajo
Estoy abajo
Estoy arriba
Estoy arriba
Estoy arriba
```

Para saber si la ejecución pasa por un punto o no

```
r(abajo)    :- write('Lanzamos retract...'),
               retract(estoy),
               write('Exito. Lanzamos assert...'),
               assert(estoy(arriba)).
```

```
Lanzamos retract...1 ADVERTENCIA: L
a regla de control de tu personaje
ha fracasado!
```

retract se lanza y fracasa.
La ejecución nunca llega al segundo **write**.