# PRÁCTICAS DE MATEMÁTICAS 1

## 2018-2019


**INDETERMINISMO**

Comportamientos que cambian en ejecución

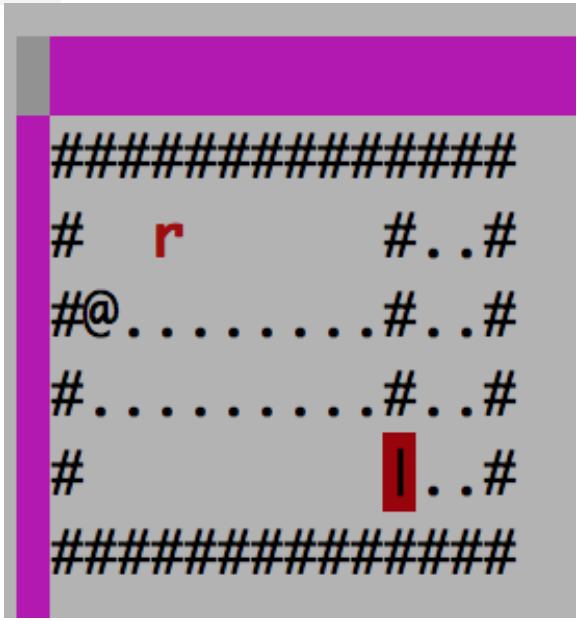
**PREDICADOS DINÁMICOS**

**De PROLOG**

¿Qué son?

¿Para qué sirven?

**Resuelve mapas Fase3**




La llave roja puede **aparecer** en cualquier sitio:

**Indeterminismo**

La solución debe funcionar **siempre**

**Descarga script launch**

https://logica.i3a.ua.es/downloads/launch.zip

**$ ./launch 10  mapa.pl  solucion.pl**

*launch ejecuta n veces solución.pl "cambiando" posiciones /comportamiento de los objetos indeterministas del mapa*

```
$ ./launch 10 maps/fase3/mapa0.pl sol.pl r
Ejecucion 1: MAPA SUPERADO 100%
Ejecucion 2: MAPA SUPERADO 100%
Ejecucion 3: MAPA SUPERADO 100%
Ejecucion 4: LIMITE DE MOVIMIENTOS SUPERADO
Ejecucion 5: MAPA SUPERADO 100%
Ejecucion 6: LIMITE DE MOVIMIENTOS SUPERADO
Ejecucion 7: MAPA SUPERADO 100%
Ejecucion 8: LIMITE DE MOVIMIENTOS SUPERADO
Ejecucion 9: MAPA SUPERADO 100%
Ejecucion 10: MAPA SUPERADO 100%
-------------------------------------------------------
-- RESULTADO FINAL:
-------------------------------------------------------
Total de ejecuciones superadas al 100%: 7 de 10 (70%)
```


# INDETERMINISMO, ¿ Qué es ?

Mapas de Fase 3, Fase 4

En cada ejecución de un mapa:

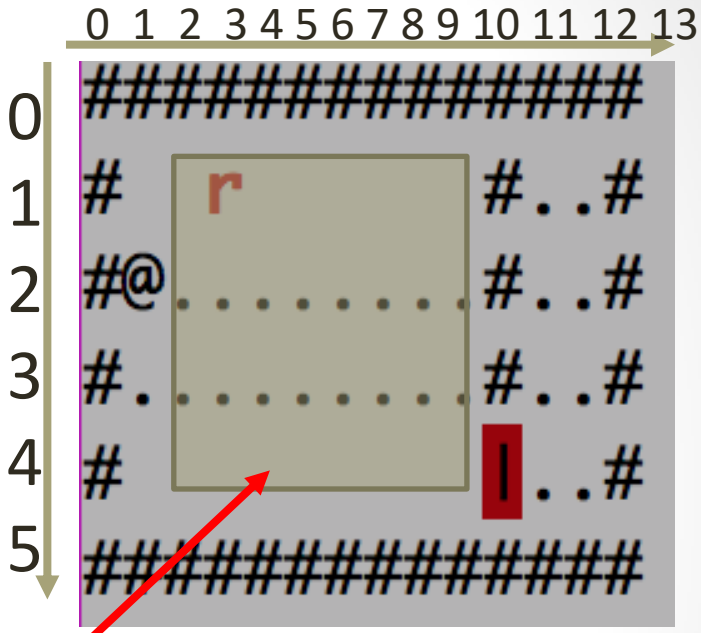
- Un objeto puede estar en **distinta posición.**
- Un enemigo puede **comportarse diferente.**
- Plman podría **empezar** en un sitio diferente.
- Los cocos pueden **aparecer / desaparecer.**




Para saber dónde puede aparecer la llave roja

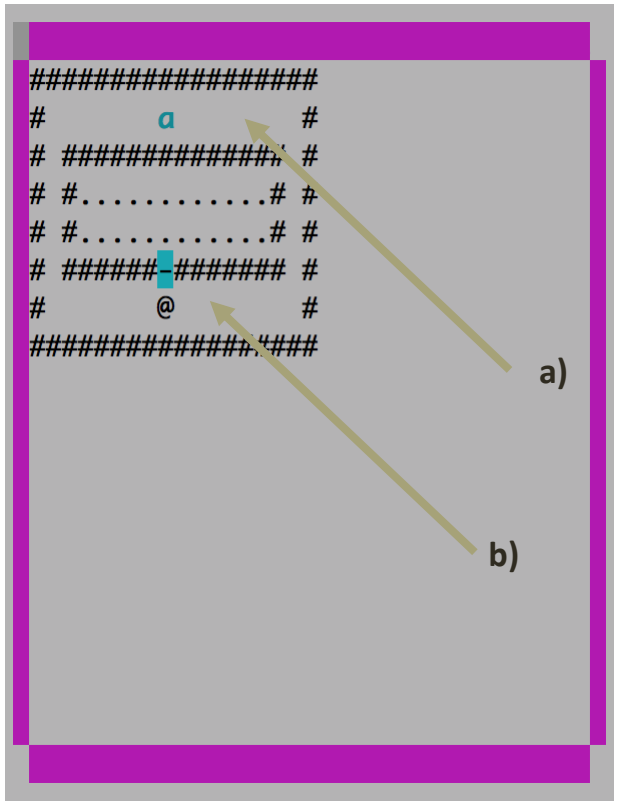
Lee el código

Cada mapa es una matriz




```
map_size(14, 6).
num_dots(25).
pacman_start(1, 2).
initMap:-
    addSolidObject('#'),
    createGameEntity(OID_K, 'r', object, rnd(2,9), rnd(1,4), inactive, norule,
            [name(llave_roja), solid(false), static(false), use_rule(basicDoorKey),
            description('Llave que abre la puerta roja'), appearance(attribs(bold, red, default))]),
```

# Hay mapas que son deterministas pero tiene condiciones adversas




**maps/fase3/mapa1.pl**

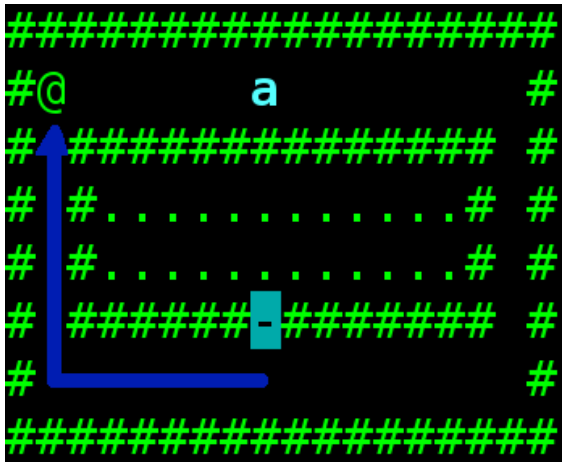
Situación idéntica en las zonas a) y b)
Tic-Tac- derecha / izquierda

¿Cómo resolverlo?

¿ con havingObject ????

Resuelve y comprueba

**maps/fase3/mapa1.pl**

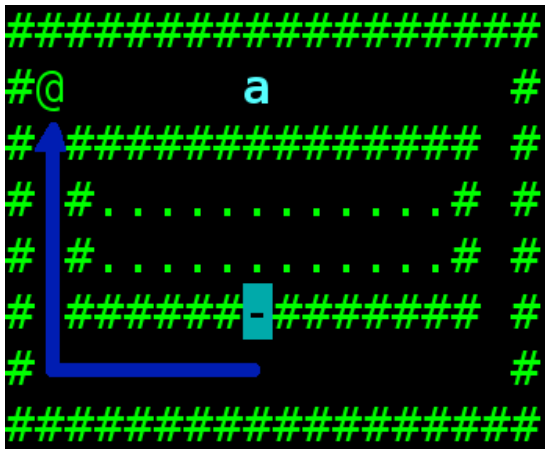


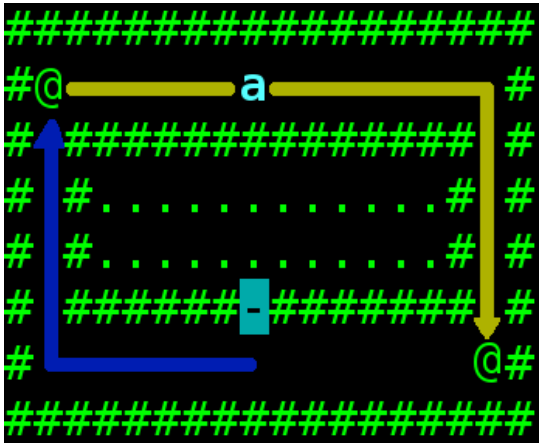

solución.pl

```
:- use_module('pl-man-game/main').
s(D,O) :- see(normal, D, O).

do(move(left))  :-  s(left, ' ').
do(move(up))    :-  s( up, ' ').
do(move(right)).
```

*Arriba encontramos un "problema" al movernos a la derecha…*

maps/fase3/mapa1.pl




## Usamos subreglas…

```
:- use_module('pl-man-game/main').
s(D,O) :- see(normal, D, O).

% Regla que lanza subreglas
do(ACT)   :-  doSubir(ACT).

% Subreglas para subir por la izda
doSubir(move(left))    :-  s( left, ' ').
doSubir(move(up))      :-  s( up, ' ').
```

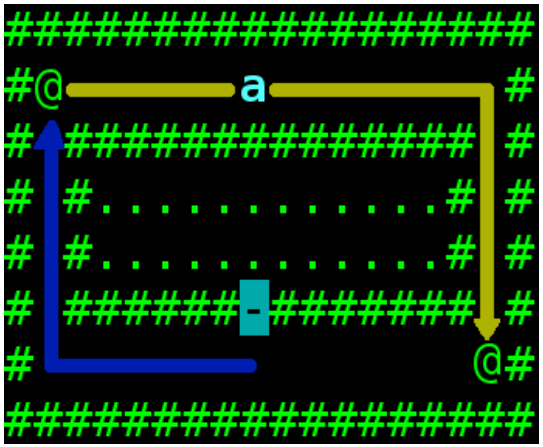
***Si ahora añadimos:***

```
doSubir(move(right))   :- s( right, ' ' ).
```

***Aparece TIC-TAC …***

***Esto no funciona, seguimos…***

Añadimos subreglas para la parte de arriba




maps/fase3/mapa1.pl

```
:- use_module('pl-man-game/main').
s(D,O) :- see(normal, D, O).

% Regla que lanza subreglas
do(ACT)   :-  doSubir(ACT).
do(ACT)   :-  doBajar(ACT).

% Subreglas para subir por la izda
doSubir(  move(left))  :- s( left, '  ').
doSubir(   move(up))   :- s( up, '  ').

% Subreglas para bajar por la dcha
doBajar(  move(right))  :-  s( right, '  ' ).
doBajar(     get(right))  :-  s( right, 'a ').
doBajar( move(down))  :- s( down, '  ').
```

Pero…¿ cómo se "pasa" de las subreglas de subir a las de arriba ?

**1º**

Añadimos un HECHO

para indicar por dónde queremos

empezar (subreglas para subir)

**2º:**

Ahora subir/bajar son argumentos

**doSubir(ACT) >> do1(EST,ACT)**

**Ejecución**:

empieza ejecutando reglas de "subir"

¿Cómo "pasamos" a las reglas de "bajar" ??

```
:- use_module('pl-man-game/main').
s(D,O) :- see(normal, D, O).

estado(subir).

% Regla que lanza subreglas
do(ACT) :-  estado(EST), do1(EST, ACT).

% Subreglas para subir por la izda
do1(subir,    move(left))  :- s( left, ' ').
do1(subir,     move(up))  :- s( up, ' ').

% Subreglas para bajar por la dcha
do1(bajar,   move(right))  :-  s( right, ' ' ).
do1(bajar,     get(right))  :-  s( right, 'a ').
do1(bajar, move(down)) :-  s( down, ' ').
```

*¿ Se arreglaría añadiendo el hecho*
**estado(bajar)** *?*


Seguimos sin “pasar” a subreglas de bajar

**Solución:**
Añadir / eliminar los hechos

**Predicados dinámicos en Prolog**

```
:- use_module(‘pl-man-game/main’).
s(D,O) :- see(normal, D, O).

estado(subir).
estado(bajar).

% Regla que lanza subreglas
do(ACT) :-  estado(EST), do1(EST, ACT).

% Subreglas para subir por la izda
do1(subir,    move(left))  :- s( left, ’  ’).
do1(subir,     move(up))  :- s( up, ’  ’).

% Subreglas para bajar por la dcha
do1(bajar,   move(right))  :-  s( right, ’  ’ ).
do1(bajar,      get(right))  :-  s( right, ’a ’).
do1(bajar,  move(down))  :-  s( down, ’  ’).
```



# Predicados dinámicos en Prolog

>> Permiten **añadir / eliminar** cláusulas (hechos, reglas) en la BC durante la ejecución**.**

>> Sólo trabajaremos con **hechos**

>> Se declaran mediante la directiva **dynamic/1**

**:- dynamic predicado /n**

Predicados ISO_Standard

>> Para **añadir** HECHOS dinámicos**:** **assert/ 1,** **assert(hecho)**.

>> Para **eliminar 1** HECHO dinámico. Si el hecho no existe→ fracaso

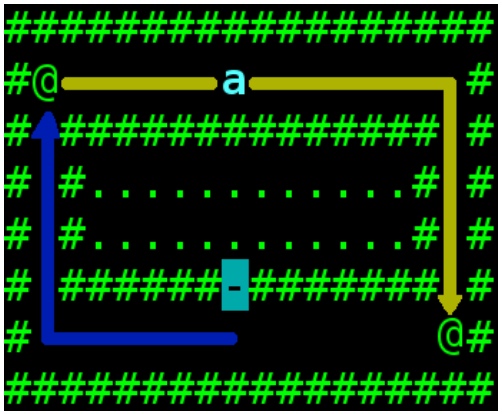
**retract/ 1, retract(hecho).**

>> Para **eliminar todos** los HECHOs dinámicos:

**retractall/1, retractall(hecho).**

Siempre tiene éxito, incluso cuando ningún hecho unifica con **HECHO**.

# Solución (incompleta) con Predicados dinámicos

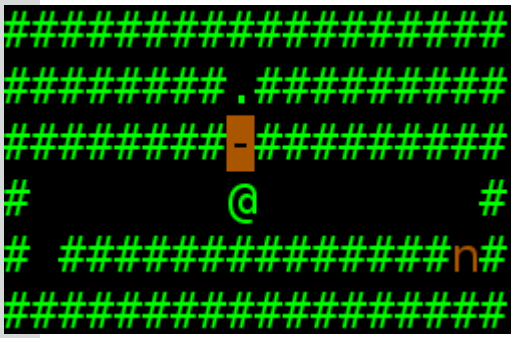



```
:- use_module('pl-man-game/main').
s(D,O) :- see(normal, D, O).
```

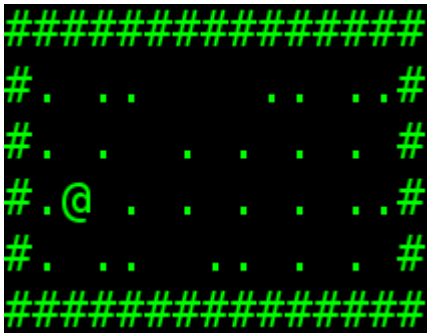
Completa solución

```
:- dynamic estado/1.

%  hecho inicial por donde quieres empezar
estado(subir).

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

% Regla que lanza subreglas
do(ACT) :-  estado(EST),   do1(EST, ACT).

% Reglas para subir por la izquierda
do1(subir,    move(left))  :- s( left, '  ').
do1(subir,     move(up))   :- s( up, '  ').
do1(subir,  move(right))   :-  change(bajar).

% Reglas para bajar por la derecha
do1(bajar,  move(right))  :-  s( right, '  ' ).
do1(bajar, move(down)) :-  s( down, '  ').
do1(bajar,     get(right))  :-  s( right, 'a ').
……
```

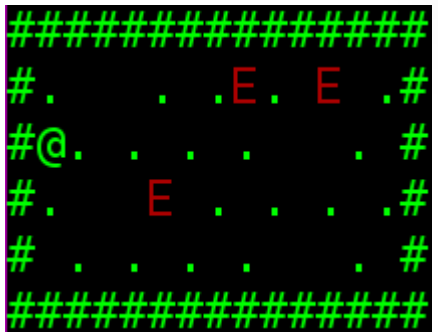
# HACER MAPAS ejemplos de Fase 3 (plman/maps/fase3)



mapa2

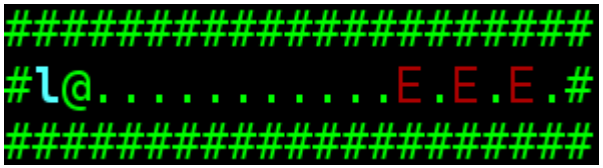


mapa3

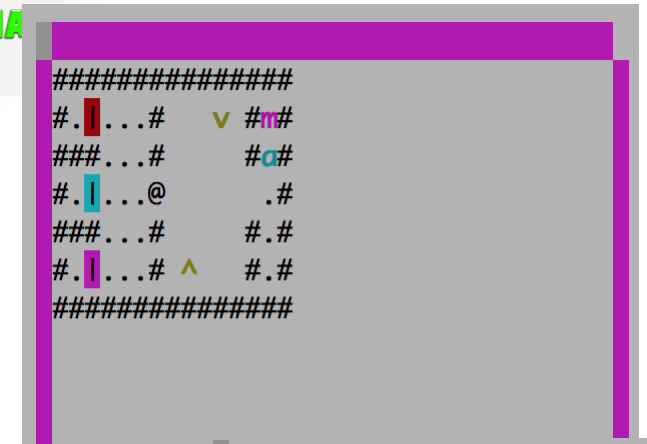


mapa4

mapa6



mapa5




maps/fase3/mapa6.pl

```
%
% Las 3 puertas de este mapa siempre aparecen en orden aleatorio. Las llaves para abrirlas
% están siempre en la cámara de la derecha, pero en posición aleatoria (pueden estar incluso
% varias de ellas en la misma casilla)
%
% Los arqueros automáticos están siempre uno arriba a la derecha y otro abajo a la izquierda,
% pero pueden aparecer en una columna aleatoria. El de abajo a la izquierda puede estar en las
% columnas 7 y 8, y el de arriba a la derecha puede estar en las columnas 10 y 11
%
```

# *Revisando fallos*

Para **revisar un fallo**, volver a ejecutar no sirve, el **mapa cambia**

Script **launch** permite **ejecutar** la solución de una mapa **n veces**

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

El Script launch guarda logs de las que fallan

Plman permite **guardar el log** de una ejecución

**$ ./plman mapa.pl solucion.pl -l archivoLog.log**

**Reproducir** un log de una ejecución

**$ ./plman -r archivoLog.log**

**Teclas:**

**P** se avanza;

**O** se retrocede;

**1-9** se cambia la velocidad de reproducción.

**ESC** para salir.