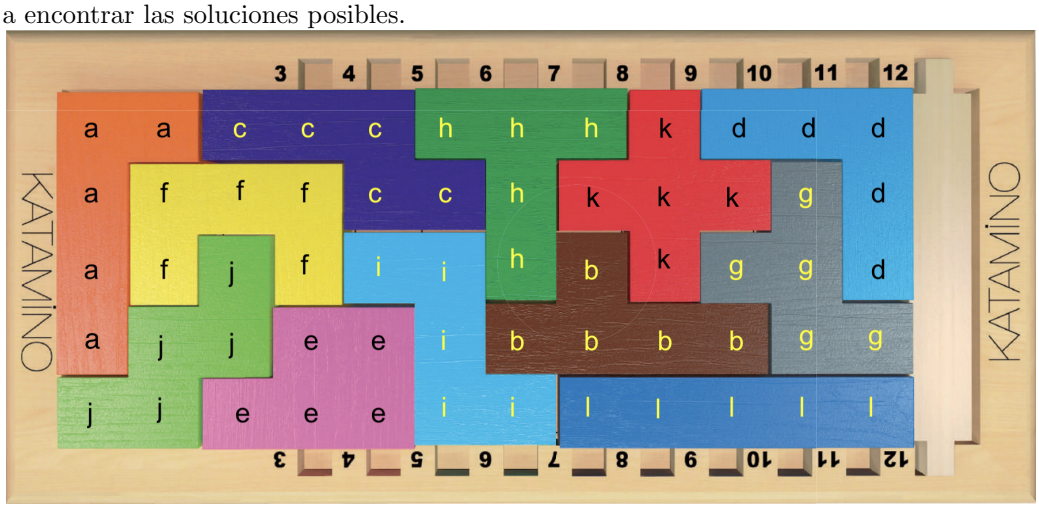
Trabajo Práctico 2
Programación Lógica
versión 1.2

Paradigmas ~~de Lenguajes~~ de Programación
1er cuatrimestre 2025

Fecha de entrega: jueves 19 de junio

1. Introducción

El Katamino es un juego que contiene 12 piezas que hay que acomodar ocupando todos los espacios del tablero. El tablero es de 5 filas y entre 3 y 12 columnas. En este trabajo vamos a implementar un programa que nos ayude a encontrar las soluciones posibles.



Archivos base

Disponemos del archivo `piezas.pl` con algunas funciones ya definidas. Dicho archivo no debe ser modificado. El archivo `katamino.pl` es donde van a definir los predicados pedidos. El archivo `katamino.pl` incluye a `piezas.pl`. Deben estar situados en el mismo directorio.

```
% cat katamino.pl
:- use_module(piezas).


% swipl katamino.pl
?-
```

Piezas

Las piezas van a estar identificadas por una letra `a,..., l`. Ya disponemos de un predicado `nombrePiezas/1` que nos indica los nombres de las piezas.

```
?- nombrePiezas(L).
L = [a, b, c, d, e, f, g, h, i, j, k, l].
```

También disponemos del predicado `pieza(+Identificador, -Forma)` que nos permitirá obtener cada pieza como una matriz como lista de filas.

```
?- pieza(a, F).
F = [[a,a],
     [a,_],
     [a,_],
     [a,_]] .
```

```
?- pieza(b, F).
F = [[_, b, _, _],
     [b, b, b, b]] .
```

En cada matriz, hay elementos instanciados con el identificador de la pieza donde la pieza ocupa espacio y valores no instanciados donde la pieza no ocupa espacio.

El predicado `pieza/2` nos **va a enumerar también todas las posibles orientaciones de la pieza**. La enumeración es sin repetidos.

```
?- pieza(e, F).
F = [[_, e, e],
     [e, e, e]] ;

F = [[e, e],
     [e, e],
     [_, e]] ;
...
```

Mostrar

Se dispone de `mostrar(+M)` que imprime en pantalla tanto una pieza como un tablero. Cada ficha ya tiene un color/patrón asignado.

```
?- pieza(a, P), mostrar(P).
```

```
P = [[a, a], [a, _], [a, _], [a, _]] ;
```

```
P = [[a, _, _, _], [a, a, a, a]] ;
```

2. Ejercicios

Ejercicio 1: Sublista

Escribir un predicado `sublista(+Descartar, +Tomar, +L, -R)` que sea cierto cuando `R` es la sublista de longitud `Tomar` luego de descartar los primeros `Descartar` elementos de `L`. Para este predicado se debe usar `append/3` y no puede ser recursivo. El siguiente ejemplo muestra la única solución posible para el caso de descartar 2 elementos y tomar 3 de la lista `[a,b,c,d,e,f]`.

```
?- sublista(2,3,[a,b,c,d,e,f],R).
R = [c, d, e].
```

Ejercicio 2: Tablero

Escribir un predicado `tablero(+K, -T)` que genera un tablero vacío de $K > 0$ columnas. El tablero será una matriz de $5 \times K$ representada como lista de filas. Cada casilla del tablero debe ser una variable no instanciada distinta.

```
?- tablero(3, T)
T = [[_,_,_],
     [_,_,_],
     [_,_,_],
     [_,_,_],
     [_,_,_]].
```

Ejercicio 3: Tamaño

Dada una matriz `M` representada como lista de filas. El predicado `tamaño(+M, -F, -C)` será verdadero cuando `M` tenga `F` filas y `C` columnas. Este predicado va a ser usado tanto para piezas como para tableros.

```
?- tablero(3, T), tamaño(T, F, C).
F=5, C=3.

?- pieza(e, E), tamaño(E, F, C).
```

```
F = 2, C = 3 ;
F = 3, C = 2 ;
F = 2, C = 3 ;
...
```

Ejercicio 4: Coordenadas

Dado un tablero `T` de $5 \times K$, `coordenadas(+T, -IJ)` debe ser verdadero para todo par `IJ` que sea una coordenada de elementos del tablero. Los índices irán de $1 \dots 5$ y $1 \dots K$ donde $(1, 1)$ es la esquina superior izquierda. Puede dar los elementos en cualquier orden pero sin repetidos.

```
?- tablero(3, T), coordenadas(T, IJ).
IJ = (1,1) ;
IJ = (1,2) ;
...
IJ = (5,3).
```

Ejercicio 5: K-Piezas

Debemos cubrir todos los espacios del tablero de 5 filas y K columnas. Para esto sabemos que vamos a necesitar exactamente K piezas ya que todas las piezas ocupan 5 espacios. `kPiezas(+K, -PS)` debe ser verdadero cuando `PS` es una lista de longitud K de identificadores de piezas. **Es importante** que no repita soluciones (ni permutaciones) y que corte lo antes posible. Para esto haremos que la lista `PS` siempre tenga los elementos ordenados. Utilizar `nombrePiezas/1` para obtener el identificador posible de las piezas.

```
?- kPiezas(3, PS).
PS = [a,b,c] ;
PS = [a,b,d] ;
PS = [a,b,e] ;
PS = [a,b,f] ;
...
PS = [b,c,d] ;
...

?- kPiezas(12, PS).
PS = [a,b,c,d,e,f,g,h,i,j,k,l].
```

Tip:

- Cada pieza puede ser elegida o no.
- A cada paso podemos verificar si nos quedan suficientes piezas para completar la selección. Ej: Si tenemos que elegir 5, pero sólo nos quedan 4 candidatas ya sabemos que no podemos satisfacer la selección.

Ejercicio 6: SeccionTablero

Para ubicar una pieza vamos a seleccionar una sección del tablero. El predicado `seccionTablero(+T, +ALTO, +ANCHO, +IJ, ?ST)` será verdadero cuando `ST` sea una sección de tamaño `ALTO` $\times$ `ANCHO` del tablero `T` a partir de la coordenada `IJ`. Aprovechar `sublista/4`.

```
?- tablero(3, T), seccionTablero(T, 3, 2, (1,2), ST).
T  = [[_C11,_C12,_C13],
      [_C21,_C22,_C23],
      [_C31,_C32,_C33],
      [_C41,_C42,_C43],
      [_C51,_C52,_C53]],

ST =  [[_C12,_C13],
       [_C22,_C23],
       [_C32,_C33]].
```

Si no hay posible sección de tablero de este tamaño entonces el predicado da `false`.

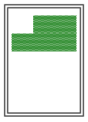
```
?- tablero(3, T), seccionTablero(T, 3, 3, (1,2), ST).
false.
```

¿Para qué va a servir esto? ¡Para saber si una pieza encaja! Con solo unificar `ST` con una pieza del tamaño correcto podemos ubicarla. Notar la última `E` en la siguiente expresión:

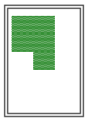
```
?- tablero(3, T), pieza(e, E), tamaño(E, F, C), seccionTablero(T, F, C, (1,1), E).
T  = [[_,e,e],
      [e,e,e],
      [_,_,_],
      [_,_,_],
      [_,_,_]] ;
...
T  = [[e,e,_],
      [e,e,_],
      [_,e,_],
      [_,_,_],
      [_,_,_]] ;
...
```

El predicado `mostrar/1` puede usarse para ver el estado del tablero.

```
?- tablero(3, T), pieza(e, E), tamaño(E, F, C), seccionTablero(T, F, C, (1,1), E), mostrar(T).
```



```
T = [[_A, e, e], [e, e, e], [_, _, _], [_, _, _], [_, _, _]],
E = [[_A, e, e], [e, e, e]],
F = 2,
C = 3 ;
```



```
T = [[e, e, _], [e, e, _], [_A, e, _], [_, _, _], [_, _, _]],
E = [[e, e], [e, e], [_A, e]],
F = 3,
C = 2 ;
```

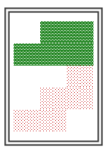
Además como cada pieza tiene un identificador distinto ¡la unificación nos va a evitar poner una pieza sobre otra!

Ejercicio 7: Ubicar pieza

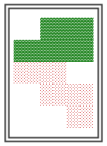
Dado un tablero `ubicarPieza(+Tablero, +Identificador)` debe generar todas las posibles ubicaciones de la pieza dada en el tablero. El `Tablero` va a ser una matriz con estructura instanciada pero con casillas posiblemente no instanciadas para indicar casillas libres. Soluciones similares via rotaciones de tablero se consideran distintas. Aprovechar `seccionTablero/5`.

```
?- tablero(3, T), ubicarPieza(T, e).
T  = [[_,e,e],              T  = [[_,_,_],
      [e,e,e],                    [_,e,e],
      [_,_,_],                    [e,e,e],
      [_,_,_],                    [_,_,_],
      [_,_,_]] ;                  [_,_,_]] ;
...                         ...
```

```
?- tablero(3, T), ubicarPieza(T, e), ubicarPieza(T, j), mostrar(T).
```



```
T = [[_, e, e], [e, e, e], [_, _, j], [_, j, j], [j, j, _]] ;
```



```
T = [[_, e, e], [e, e, e], [j, j, _], [_, j, j], [_, _, j]] ;
```

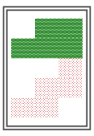
Ejercicio 8: Ubicar piezas

Similar a `ubicarPieza/2`, el predicado `ubicarPiezas(+Tablero, +Poda, +Identificadores)` debe listar todas las posibles opciones de ubicar todas las piezas mencionadas en `Identificadores`. Asumir que la lista `Identificadores` va a estar ordenada. El valor de `Poda` indica qué estrategia usar para podar el espacio de búsqueda. El `Tablero` va a ser una matriz con estructura instanciada pero con casillas posiblemente no instanciadas para indicar casillas libres.

Al momento contamos con una única estrategia `sinPoda` que no poda. Y equivale a ubicar una pieza tras otra sin hacer chequeos adicionales entre una y otra. Definir y usar el predicado `poda/2` donde `poda(+Poda, +Tablero)` sea verdadero si el tablero satisface la poda. Por ahora el único valor para `Poda` será `sinPoda`.

```
poda(sinPoda, _).
```

```
?- tablero(3, T), ubicarPiezas(T, sinPoda, [e, j]), mostrar(T).
```

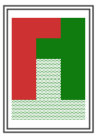


```
T = [[_, e, e], [e, e, e], [_, _, j], [_, j, j], [j, j, _]] ;
```

Ejercicio 9: Llenar Tablero

Dada la cantidad de columnas de un tablero `llenarTablero(+Poda, +Columnas, -Tablero)` debe enumerar todas las formas distintas de llenar un tablero con la cantidad de columnas (y piezas) indicada. El valor de `Poda` determina la estrategia de poda que `ubicarPiezas/3` va a usar. Notar que todas las casillas del tablero van a estar instanciadas en `Tablero`. Aprovechar predicados anteriores. Nuevamente `mostrar/1` puede usarse para ver el estado del tablero.

```
?- llenarTablero(sinPoda, 3, T), mostrar(T).
```



```
T = [[a, a, b], [a, b, b], [a, h, b], [a, h, b], [h, h, h]] ;
```

```
T = [[h, h, h], [b, h, a], [b, h, a], [b, b, a], [b, a, a]] ;
```

Es posible que para valores `K >= 5` este predicado tarde más de lo que esperan.

Ejercicio 10: Medición

Definir el predicado `cantSoluciones/3` que calculará cuántas soluciones distintas hay para un tablero con la cantidad de columnas dada de la siguiente manera.

```
cantSoluciones(Poda, Columnas, N) :-
 findall(T, llenarTablero(Poda, Columnas, T), TS),
 length(TS, N).
```

Medir para las distintas cantidades de columnas cuánto tarda el predicado `cantSoluciones/3` en dar respuesta. Usar el predicado `time/1` para medir el tiempo de ejecución. Anotar en un comentario en el código las mediciones obtenidas para `K=3` y `K=4`. Si bien los tiempos dependen del programa y máquina que usen, los valores de N deben coincidir.

```
?- time(cantSoluciones(sinPoda, 3, N)).
% 22,559,913 inferences, 1.056 CPU in 1.066 seconds (99% CPU, 21364222 Lips)
N = 28.

?- time(cantSoluciones(sinPoda, 4, N)).
% 859,463,345 inferences, 38.800 CPU in 39.024 seconds (99% CPU, 22151106 Lips)
N = 200.
```

Ejercicio 11: Optimización

Actualmente, cada vez que se ubica una pieza puede que el tablero quede de tal forma que nunca va a poder completarse. Ej: Si alguno de los grupos de lugares libres no es divisible por 5. Definir una estrategia de poda `podaMod5` de forma tal que `ubicarPiezas(podaMod5, ES, T)` realice el chequeo de dicha poda a medida que va ubicando cada pieza. Para esto se debe agregar otra regla a `poda/2` y definir luego el predicado `todosGruposLibresModulo5(+T)`.

```
poda(sinPoda, _).
poda(podaMod5, T) :- todosGruposLibresModulo5(T).
```

El predicado `todosGruposLibresModulo5(+Tablero)` recibe un `Tablero` que va a ser una matriz con estructura instanciada pero con casillas posiblemente no instanciadas para indicar casillas libres. Debe ser verdadero si las casillas libres agrupadas son todas de tamaño módulo 5. **Es importante** que el predicado `todosGruposLibresModulo5/1` y todos los auxiliares usados para definirlo **no sean recursivos**.

Se dispone del predicado `agrupar(+L, -G)` que dada una lista de coordenadas `(I,J)`, será verdadero si G es una lista de grupos de dichas posiciones. Dos casillas se agrupan si comparten un lado.

```
?- agrupar([(1,1),(2,2),(2,1),(3,3),(4,4),(3,4)],G).
G = [[(1, 1), (2, 2), (2, 1)], [(3, 3), (4, 4), (3, 4)]].
```

Tip:

- Definir un predicado que sea verdadero si una coordenada es libre en un tablero.
- Usar `findall/3` para obtener todas las coordenadas libres.

Repetir las mediciones del ejercicio anterior para comparar cómo mejora esta poda el espacio de búsqueda.

```
?- time(cantSoluciones(podaMod5, 3, N)).
% 11,586,733 inferences, 0.655 CPU in 0.737 seconds (89% CPU, 17690452 Lips)
N = 28.

?- time(cantSoluciones(podaMod5, 4, N)).
% 248,960,072 inferences, 14.674 CPU in 16.450 seconds (89% CPU, 16965505 Lips)
N = 200.
```

A continuación se muestran los resultados de nuestra solución para usar de referencia. Los tiempos y cantidad de inferencias pueden variar según la máquina y el programa que usen, pero los valores de N deben coincidir.

K	N	Tiempo sinPoda	tiempo podaMod5	inferencias sinPoda	inferencias podaMod5
3	28	1s	0.6s	22M	11M
4	200	39s	16s	859M	248M
5	856	15.8m	4.57m	21000M	3800M
6	2164	n/a	53m	n/a	41000M
7	5584	n/a	13.5h	n/a	361000M

Ejercicio 12: Reversibilidad

Se desea analizar si el predicado `sublista/4` puede ser usado para averiguar si una lista dada es una sublista de otra y en tal caso cuántos elementos hay que descartar. O sea, si es reversible en el primer y cuarto argumentos, y si puede ser usado como `sublista(-Descartar, +Tomar, +L, +R)`. Indicar en un comentario en el código de `sublista/4` si lo es o no y por qué. Recomendación: analizar casos que debieran tener una, múltiples y ninguna solución. Nota: no hace falta cambiar el código que ya está implementado.

3. Pautas de Entrega

Importante: Se espera que la elaboración de este trabajo sea 100 % de los estudiantes del grupo que realiza la entrega. Así que, más allá de que pueden tomar información de lo visto en las clases o consultar información en la documentación de SWI-Prolog u otra disponible en Internet, no se podrán utilizar herramientas para generar parcial o totalmente en forma automática la resolución del TP (e.g., ChatGPT, Copilot, etc). En caso de detectarse esto, el trabajo será considerado como un plagio, por lo que será gestionado de la misma forma que se resuelven las copias en los parciales u otras instancias de evaluación.

Se deberá subir subir el código fuente a la tarea respectiva en el Campus.

El código debe poder ser ejecutado en SWI-Prolog. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté adecuadamente comentado. Los objetivos a evaluar en la implementación de los predicados son:

- corrección,
- declaratividad,
- reutilización de predicados previamente definidos
- Uso de unificación, backtracking, generate and test y reversibilidad de los predicados que correspondan.
- No está permitido usar macros que colapsen cláusulas, como `;` y `->`.
- **Importante**: salvo donde se indique lo contrario, los predicados no deben instanciar soluciones repetidas. Vale aclarar que no es necesario filtrar las soluciones repetidas si la repetición proviene de las características de la entrada.

Importante: se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

4. Referencias y sugerencias

Como referencia se recomienda la bibliografía incluída en el sitio de la materia (ver sección *Bibliografía* → *Programación Lógica*).

Se recomienda que, siempre que sea posible, utilicen los predicados ISO y los de SWI-Prolog ya disponibles. Recomendamos especialmente examinar los predicados y metapredicados que figuran en la sección *Otros* de la página de la materia. Pueden hallar la descripción de los mismos en la ayuda de `SWI-Prolog` (a la que acceden con el predicado `help`). También se puede acceder a la documentación online de SWI-Prolog.