Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información.
CI–3661 – Laboratorio de Lenguajes de Programación
Septiembre–Diciembre 2024

Proyecto II: La Leyenda de Celda

Hace apenas unos días que salió al público el nuevo videojuego de la reconocida empresa ***No–entiendo***. El más reciente juego de una exitosa saga épica, llamado:

"La Leyenda de Celda: El Secreto de la Matriz Traspuesta".

En esta oportunidad la princesa Celda, de la gran matriz, ha sido secuestrada por el maligno vector Zero. El nulísimo villano desea utilizar a el poder escondido en Celda para trasponer la gran matriz, trayendo caos y miseria al mundo. Ante esta terrible situación, el valiente *Hiperlink* se propone salir y derrotar al malvado vector de una vez por todas, rescatar a la princesa secuestrada y traer paz nuevamente al mundo.

Siendo fanático de la saga, fuiste de los primeros en comprar y comenzar el juego. Como siempre, el videojuego estaba lleno de rompecabezas y de calabozos (o *templos*) que debían que superarse para seguir avanzando. Investigando uno de estos templos, te encontraste ante un salón enorme con lo que parecía ser un laberinto intrincado. Despues de un par de vistazos más, concluíste que parecía demasiado complicado como para resolverse facilmente. Te negabas a buscar la respuesta en internet. ¡Como detestabas a esos falsos fanáticos! Aunque, muy en tu interior, las ganas de ir a revisar no te faltaban.

Estabas a punto de rendirte y dejar de jugar por el día, cuando lograste encontrar un cofre de tesoro a la entrada del laberinto. Al revisarlo, te diste cuenta que contiene nada menos que un mapa de todo el laberinto. El mapa marcaba cada pasillo con una letra minúscula, algunas extrañamente de cabeza. Confiado, ignoraste las inusuales marcas y, sin pensarlo dos veces, usaste el mapa para intentar cruzar el laberinto. Pero el mapa no te mostró que alguno de los pasillos eran trampas y fuiste testigo de como *Hiperlink* cayó, desprevenido. Unos segundos después apareciste al inicio del laberinto, un corazón menos en tu estátus, listo para intentarlo de nuevo.

Inspeccionando mejor el lugar, notaste que en una esquina había un panel de control escondido con algunas palancas verticales, colocadas en fila. Cada una de estas palancas tenía una marca grabada, correspondiente a las mismas letras minúsculas que aparecen en el mapa. A un lado de éstas se encontraba una pequeña lápida con algunas inscripciones. Aunque difíciles de leer, lograste descifrar el siguiente mensaje:

"Aquel que desee cruzar el laberinto, debe probar primero su sabiduría y determinación. Cada palanca accionará un comportamiento sobre los pasillos del laberinto que su marca compartan. Si la palanca mira hacia arriba, los pasillos afectados serán seguros. Si la palanca mira hacia abajo, los pasillos afectados serán trampas. Mas, cuidado, pues aquellos pasillos que tengan la marca de cabeza recibirán el efecto contrario. Mientras los pasillos con cierta marca estén seguros, aquellos con la misma marca de cabeza serán trampas y viceversa."

Parecía ser una tarea ardua el colocar las palancas en una posición tal que se pueda cruzar seguramente, pero tenías una ventaja sin igual a tu favor. Eras un estudiante de Ingeniería en Computación que, al estar viendo el Laboratorio de Lenguajes de Programación, se dió cuenta que puede utilizar el lenguaje *Prolog* para ayudarlo a resolver este acertijo.

Cruzar o no cruzar

Decidiste entonces definir el predicado `cruzar\3`, con la siguiente forma:

```
cruzar(Mapa, Palancas, Seguro)
```

Donde `Mapa` es la especificación del laberinto en cuestión, `Palancas` es la configuración de las palancas y `Seguro` es el estado de éxito de la combinación (Si existe un camino para cruzar el laberinto seguramente o no).

`Mapa` será una estructura que puede tomar alguna de las siguientes formas, donde cada `SubMapa` presente tiene la misma estructura posible que `Mapa`:

- `pasillo(X, Modo)` : Un pasillo del laberinto, donde `X` es la letra asociada al pasillo (Por ejemplo: `a`, `b`, `c`, etc.) y `Modo` corresponde a que el caracter esté regular o de cabeza (`regular` y `de_cabeza`, respectivamente). Para poder cruzar este pasillo, la palanca correspondiente al caracter en `X` debe estar arriba (si `Modo` es `regular`) o abajo (si `Modo` es `de_cabeza`).
- `junta(SubMapa1, SubMapa2)`: Es la secuencia de dos submapas. Para poder cruzar esta junta, debe poder cruzarse primero `SubMapa1` y luego, igualmente, `SubMapa2`.
- `bifurcación(SubMapa1, SubMapa2)`: Es la bifurcación del camino en dos mapas. Para poder cruzar esta bifurcación, basta con poder cruzar `SubMapa1` o, equivalentemente, `SubMapa2`.

`Palancas` será una lista de asociaciones (pares ordenados) de la forma: `(X, Posicion)`. Donde, `X` es cada una de las letras que aparecen en `Mapa` y `Posicion` es la posición de la palanca correspondiente a la letra en `X`, que puede ser `arriba` o `abajo`.

`Seguro` será simplemente uno de dos valores: `seguro` si existe alguna manera de cruzar el laberinto o `trampa`, de lo contrario.

De entre los tres argumentos de `cruzable\3`, `Mapa` siempre debe estar instanciado. Los otros argumentos, `Palancas` y `Seguro`, pueden estarlo o no. Si `Palancas` está instanciado, entonces `Seguro` debe unificar con `seguro` de ser cierto que la disposición de las palancas crea algún camino seguro para cruzar el laberinto; o `trampa` de lo contrario. Si `Seguro` está instanciado, entonces `Palancas` debe unificar con cada lista de asociaciones, para cada letra (en el formato expresado anteriormente), tal que la existencia de un camino seguro o no corresponda con lo indicado en `Seguro`.

Como ejemplo, considere las siguientes consultas con sus unificaciones esperadas:

```
?- cruzar(
       pasillo(a, regular),
       [(a, arriba)],
       Seguro
   ).
Seguro = seguro.

?- cruzar(
       pasillo(a, de_cabeza),
       [(a, arriba)],
       Seguro
   ).
Seguro = trampa.
```

```
?- cruzar(
       junta(
           pasillo(a, regular),
           bifurcacion(
               pasillo(b, regular),
               pasillo(c, de_cabeza)
           )
       ),
       Palancas,
       seguro
   ).
Palancas = [(a, arriba), (b, arriba), (c, arriba)];
Palancas = [(a, arriba), (b, arriba), (c, abajo)];
Palancas = [(a, arriba), (b, abajo), (c, abajo)].

?- cruzar(
       junta(
           pasillo(a, regular),
           bifurcacion(
               pasillo(b, regular),
               pasillo(c, de_cabeza)
           )
       ),
       Palancas,
       trampa
   ).
Palancas = [(a, arriba), (b, abajo), (c, arriba)];
Palancas = [(a, abajo), (b, arriba), (c, arriba)];
Palancas = [(a, abajo), (b, arriba), (c, abajo)];
Palancas = [(a, abajo), (b, abajo), (c, arriba)];
Palancas = [(a, abajo), (b, abajo), (c, abajo)].

?- cruzar(
       junta(
           pasillo(a, regular),
           pasillo(a, de_cabeza)
       ),
       Palancas,
       seguro
   ).
false.

?- cruzar(
       junta(
           pasillo(a, regular),
           pasillo(a, de_cabeza)
       ),
       Palancas,
       trampa
   ).
Palancas = [(a, arriba)];
Palancas = [(a, abajo)].
```

Con este nuevo predicado, podías conseguir todas las disposiciones posibles de las palancas, con las cuales se puede cruzar el laberinto, ya sea seguramente o cayendo en una trampa. Así mismo, podías verificar la seguridad de una disposición ya dada.

Sin miedo y sin pausa

Ahora podías cruzar el laberinto fácilmente. Sin embargo, sospechaste que, sin importar la disposición de las palancas, siempre había una manera de cruzar el laberinto. Para confirmar esta sospecha, decideste entonces implementar el predicado `siempre_seguro\1`, con la siguiente forma:

```
siempre_seguro(Mapa)
```

Donde `Mapa` tiene la misma semántica y dominio que en `cruzar\3`, e igualmente debe estar siempre instanciado. Este predicado triunfa, si indiferentemente de la disposición de palancas, el laberinto en `Mapa` puede cruzarse de forma segura.

Como ejemplo, considere las siguientes consultas con sus unificaciones esperadas:

```
?- siempre_seguro(
       pasillo(a, regular)
   ).
false.

?- siempre_seguro(
       bifurcacion(
           pasillo(b, regular),
           pasillo(b, de_cabeza)
       )
   ).
true.

?- siempre_seguro(
       junta(
           pasillo(b, regular),
           pasillo(b, de_cabeza)
       )
   ).
false.

?- siempre_seguro(
       bifurcacion(
           pasillo(a, regular),
           pasillo(b, de_cabeza)
       )
   ).
false.

?- siempre_seguro(
       bifurcacion(
           bifurcacion(
               pasillo(a, de_cabeza)
               pasillo(b, de_cabeza)
           ),
           junta(
               pasillo(a, regular)
               pasillo(b, regular)
           )
       )
   ).
true.
```

Detalles de la Entrega

La entrega debe incluir:

- Un repositorio git privado (preferiblemente Github) con el código fuente en Prolog. Dicho repositorio debe ser compartido con el profesor del curso (https://github.com/rmonascal). Todo el código debe estar debidamente documentado.

  Debe implementar los predicados `cruzar\3` y `siempre_seguro\1`, descritos anteriormente. Además, debe implementar un predicado `leer\1`, de tal forma que `leer(Mapa)` pida un nombre de archivo al usuario y lea, a partir del contenido de dicho archivo, la estructura de un laberinto (con el mismo formato con el que se escribiría en el intérprete de *SWI–Prolog*). Finalmente, dicho laberinto debe quedar unificado en `Mapa`.

- Un *breve* README, a modo de informe explicando la formulación/implementación de sus predicadod y justificando todo aquello que considere necesario. Es recomendable que escriba este informe usando el lenguaje `Markdown`, para que se renderice bien desde el navegador.

Nota: No puede utilizar el predicado `findall\3` en la implementación de este proyecto.

Fecha de Entrega: Lunes, 09 de Diciembre (Semana 12), hasta las 11:59 pm.