Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información.
CI–3661 – Laboratorio de Lenguajes de Programación I
Septiembre–Diciembre 2017

Proyecto II

(Prolog - 20 %)

Hace apenas unos días que salió al público el nuevo videojuego de la reconocida empresa ***Nœntiendo***. 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*. *Zero* desea utilizar los poderes de *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 el juego y comenzar a jugarlo. Como siempre, el videojuego está lleno de rompecabezas y de calabozos (o *templos*) que tienen que superarse para seguir avanzando. Explorando uno de estos templos, te encuentras ante un salón enorme con lo que parece ser un intrincado laberinto. Cerca de la entrada de este laberinto te encuentras con un cofre de tesoro. Al revisarlo, te das cuenta de que contiene un mapa detallado de todo el laberinto, mostrando la estructura completa del mismo y marcando cada pasillo con una letra minúscula, algunas extrañamente invertidas (de cabeza). Confiado, ignoras las inusuales marcas y, sin perder tiempo meditando, usas el mapa para intentar cruzar el laberinto. Pero el mapa no advierte que alguno de los pasillos son trampas e *Hiperlink* muere desprevenido, devolviéndolo al inicio del laberinto para intentarlo de nuevo.

Inspeccionando mejor el lugar, notas que en una esquina hay un panel de control escondido con algunos interruptores colocados en fila. Cada uno de estos interruptores tiene una marca grabada, correspondiente a las mismas letras minúsculas que aparecen en el mapa. A un lado de éstos se encuentra una pequeña lápida con algunas inscripciones. Aunque difíciles de leer, logras descifrar el siguiente mensaje:

"Aquel que desee cruzar el laberinto, debe probar primero su sabiduría y determinación. Cada interruptor accionará un comportamiento sobre los pasillos del laberinto que su marca compartan. Si el interruptor está encendido, los pasillos afectados serán seguros. Si el interruptor está apagado, los pasillos afectados serán trampas. Mas cuidado aventureros, pues aquellos pasillos que tengan la marca invertida recibirán el efecto contrario. Mientras los pasillos con cierta marca estén seguros, aquellos con la misma marca invertida serán trampas y viceversa."

Parece ser una tarea ardua el colocar los interruptores en una posición tal que se pueda cruzar seguramente, pero tienes una ventaja: eres un estudiante de Ingeniería en Computación, que al estar viendo el Laboratorio de Lenguajes de Programación, se da cuenta que puede utilizar el lenguaje `Prolog` para ayudar a resolver esta tarea.

1. cruzar\3

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

```
cruzar(Mapa, Interruptores, Seguro)
```

Donde, `Mapa` es la especificación del laberinto en cuestión, `Interruptores` es la configuración de los interruptores 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 potencial 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 invertido (`regular` y `invertido`, respectivamente). Para poder cruzar este pasillo, el interruptor correspondiente al caracter en `X` debe estar encendido (si `Modo` es `regular`) o apagado (si `Modo` es `invertido`).
- `secuencia(SubMapa1, SubMapa2)`: Es la secuencia de dos submapas. Para poder cruzar esta secuencia, debe poder cruzarse primero `SubMapa1` y luego, igualmente, `SubMapa2`.
- `division(SubMapa1, SubMapa2)`: Es la división del camino en dos mapas. Para poder cruzar esta división, basta con poder cruzar `SubMapa1` o, equivalentemente, `SubMapa2`.

`Interruptores` será una lista de asociaciones (pares ordenados) de la forma: `(X, Estado)`. Donde, `X` es cada una de las letras que aparecen en `Mapa` y `Estado` es el estado del interruptor correspondiente a la letra en `X`, que puede ser `encencido` o `apagado`.

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

De entre los tres argumentos de `cruzar\3`, `Mapa` siempre debe estar instanciado. Los otros argumentos, `Interruptores` y `Seguro`, pueden estarlo o no. Si `Interruptores` está instanciado, entonces `Seguro` debe unificar con `seguro` de ser cierto que la disposición de los interruptores crea algún camino seguro para cruzar el laberinto; o `peligroso` de lo contrario. Si `Seguro` está instanciado, entonces `Interruptores` 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, encendido)],
       Seguro
   ).
Seguro = seguro.
```

```
?- cruzar(
       pasillo(a, invertido),
       [(a, encendido)],
       Seguro
   ).
Seguro = peligroso.

?- cruzar(
       secuencia(
           pasillo(a, regular),
           division(
               pasillo(b, regular),
               pasillo(c, invertido)
           )
       ),
       Interruptores,
       seguro
   ).
Interruptores = [(a, encendido), (b, encendido), (c, encendido)];
Interruptores = [(a, encendido), (b, encendido), (c, apagado)];
Interruptores = [(a, encendido), (b, apagado), (c, apagado)].

?- cruzar(
       secuencia(
           pasillo(a, regular),
           division(
               pasillo(b, regular),
               pasillo(c, invertido)
           )
       ),
       Interruptores,
       peligroso
   ).
Interruptores = [(a, encendido), (b, apagado), (c, encendido)];
Interruptores = [(a, apagado), (b, encendido), (c, encendido)];
Interruptores = [(a, apagado), (b, encendido), (c, apagado)];
Interruptores = [(a, apagado), (b, apagado), (c, encendido)];
Interruptores = [(a, apagado), (b, apagado), (c, apagado)].

?- cruzar(
       secuencia(
           pasillo(a, regular),
           pasillo(a, invertido)
       ),
       Interruptores,
       seguro
   ).
false.
```

```
?- cruzar(
       secuencia(
           pasillo(a, regular),
           pasillo(a, invertido)
       ),
       Interruptores,
       peligroso
   ).
Interruptores = [(a, encendido)];
Interruptores = [(a, apagado)].
```

Con este nuevo predicado, puedes conseguir todas las disposiciones posibles de los interruptores, con las cuales se puede cruzar el laberinto seguramente. Así mismo, podrías verificar la seguridad de una disposición ya dada.

2. siempre_seguro\1

Ahora puedes cruzar el laberinto fácilmente. Sin embargo, sospechas que sin importar la disposición de los interruptores, siempre habrá una manera de cruzar el laberinto. Para confirmar esta sospecha, decides 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 interruptores, 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(
       division(
           pasillo(b, regular),
           pasillo(b, invertido)
       )
   ).
true.

?- siempre_seguro(
       secuencia(
           pasillo(b, regular),
           pasillo(b, invertido)
       )
   ).
false.
```

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

3. leer\1

Para ahorrar tiempo cargando el mapa a tu programas, decides 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`.

Note que `leer\3` no es en realidad predicado, sino un procedimiento imperativo impuro (su intención no es la de hacer cumplir un predicado, sino la de alterar el estado del programa con información externa al mismo).

Detalles de la Entrega

La entrega del proyecto consistirá de un único archivo `p2_<carné1>&<carné2>.tar.gz`, donde `<carné1>` y `<carné2>` son los números de carné de los integrantes de su equipo. Por ejemplo, si el equipo está conformado por `00-00000` y `11-11111`, entonces su entrega debe llamarse: `p2_00-00000&11-11111.pdf`.

Tal archivo debe ser un directorio comprimido que contenga únicamente los siguientes elementos:

- Un archivo `celda.pl` con su implementación de los predicados `cruzar\3`, `siempre_seguro\1` y `leer\1`, descritos anteriormente.
- Un archivo `Readme` con los datos de su equipo (integrantes, carné, etc.) y los detalles relevantes de su implementación.

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

La tarea deberá ser entregada al prof. Ricardo Monascal *únicamente* a su dirección de correo electrónico oficial: (rmonascal@usb.ve) a más tardar el Viernes 9 de Diciembre, a las 11:59pm. VET.

Nota: Algunas implementaciones de SWI–Prolog *imprimen* `Yes` *y* `No`*, en vez de* `true` *y* `false`*. Esto es irrelevante a efectos de este proyecto.*