Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información.
CI–3661 – Laboratorio de Lenguajes de Programación I
Enero–Marzo 2012

Proyecto III

(Prolog - 20 %)

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. Zero desea utilizar a 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. Investigando uno de estos templos, te encuentras ante un salón enorme con lo que parece ser un laberinto intrincado. A la entrada de este laberinto, sin embargo, te encuentras con un tesoro. Al revisarlo, te das cuenta de que contiene un mapa de todo el laberinto. Este mapa muestra la estructura completa del mismo, marcando cada pasillo con una letra minúscula, algunas extrañamente de cabeza. Confiado, ignoras las inusuales marcas y usas el mapa para intentar cruzar el laberinto, sin perder tiempo meditando. Pero el mapa no muestra que alguno de los pasillos son trampas e *Hiperlink* muere desprevenido, devolviéndote al inicio del laberinto para intentarlo de nuevo.

Inspeccionando mejor el lugar, notas que en una esquina hay un panel de control escondido con algunas palancas verticales, colocadas en fila. Cada una de estas palancas tiene una marca grabada, correspondiente a las mismas letras minúsculas que aparecen en el mapa. A un lado de éstas 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 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."

Parece ser una tarea ardua el colocar las palancas 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.

Decides 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 `muerte`, 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 `muerte` 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 = muerte.

?- 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,
       muerte
   ).
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,
       muerte
   ).
Palancas = [(a, arriba)];
Palancas = [(a, abajo)].
```

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

Ahora puedes cruzar el laberinto fácilmente. Sin embargo, sospechas que sin importar la disposición de las palancas, 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 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.
```

Entrega:

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`.

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

La entrega del proyecto será para el día Miercoles, 21 de Marzo, hasta las 11:59pm. La entrega será por correo electrónico, al correo oficial de su profesor correspondiente: *rmonascal@ldc.usb.ve* para Ricardo Monascal (Sección 1) y *armarpc@gmail.com* para Carlos Gómez (Sección 2).

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