Trabajo Práctico Nº2: JML y KeY

Ejercicio 1:

Demuestre las siguientes fórmulas con papel y lápiz:

1. $(\forall x\ p(x) \lor \forall x\ c(x)) \rightarrow \forall x\ (p(x) \lor c(x))$
2. $\neg(\exists x\ p(x)) \rightarrow \forall x\ (\neg p(x))$
3. $(\forall x\ p(x) \rightarrow \exists x\ q(x)) \rightarrow \exists x\ (p(x) \rightarrow q(x))$
4. $(\forall x\ f(g(x)) = g(x)) \land (g(a) = b) \rightarrow (f(b) = g(a))$

Escriba ahora las fórmulas en el formato utilizado por KeY y encuentre las pruebas mediante la aplicación manual de reglas en la herramienta (es decir, sin utilizar la funcionalidad *"start/stop automated proof search"* provista por la herramienta).

Ejercicio 2:

Considere dos arreglos, b y c, con n componentes enteras. Sean j y k dos índices tales que $0 \leq j < k < n$. A continuación, escribimos "`b[j..k]`" para denotar un segmento del arreglo b (note, sin embargo, que no se puede usar esta notación en JML). Escriba expresiones en JML que describan precisamente las siguientes condiciones:

a) Todas las componentes en el segmento `b[j..k]` tienen valor cero.

b) Todas las componentes que tienen valor cero en `b` también tienen valor cero en el segmento `c[j..k]`.

c) No es el caso que todas las componentes que tienen valor cero en `b` están en el segmento `b[j..k]`.

d) El arreglo `b` contiene al menos dos componentes con valor cero.

e) El arreglo `b` contiene exactamente dos componentes con valor cero.

Ejercicio 3:

Considere la clase `Lista`, con el siguiente *esqueleto*:

```
/* Una clase simple para listas. */
public class Lista {
      Object cabeza;
      Lista cola;

      Lista(Object cabeza, Lista cola) {

            this.cabeza = cabeza;
            this.cola = cola;
      }

      // Algunos métodos...
      int largo() {
            // ...
      }
}
```

Escriba especificaciones para el método `largo` usando JML.

Ejercicio 4:

Considere el siguiente *esqueleto* de código para una clase `Cola` que implementa una cola mediante un arreglo:

```
public class Cola {
	Object[] arreglo;
	int largo;
	int primero;
	int proximo;

	Cola(int tope) {
		// ...
	}

	public int largo() {
		// ...
	}

	public void agregarElemento(Object x) {
		// ...
	}

	public Object sacarElemento() {
		// ...
	}
}
```

Escriba especificaciones JML para el código provisto (invariantes de clase y pre/post condiciones para cada método). *Observación: No es necesario implementar la clase*.

Ejercicio 5:

El objetivo de este ejercicio es especificar una clase que implementa un laberinto, el cual será implementado mediante un arreglo de dos dimensiones (por lo tanto, cada fila tiene la misma cantidad de componentes).

La clase `Laberinto` declara los siguientes elementos:

- Constantes de la forma `MOVER_X` que codifican las diferentes direcciones en las cuales puede moverse el jugador (arriba, abajo, izquierda y derecha).
- Constantes `SALIDA`, `LIBRE` y `PARED`, que codifican el tipo de cada celda en el laberinto.
- Dos campos de tipo entero, `jugadorFila` y `jugadorColumna`, con la posición actual del jugador.

Utilice el siguiente esqueleto para la clase `Laberinto`:

```
public class Laberinto {
	// CONSTANTES DE MOVIMIENTO
	public final static int MOVER_ARRIBA = 0;
	public final static int MOVER_ABAJO = 1;
	public final static int MOVER_IZQ = 2;
	public final static int MOVER_DER = 3;
```

```
    // CONSTANTES DE CELDA
    public final static int LIBRE = 0;
    public final static int PARED = 1;
    public final static int SALIDA = 2;

    /* CAMPO DE JUEGO:
    El campo de juego esta dado por un rectángulo donde las paredes se
    representan mediante entradas con valor Laberinto.PARED ('1'), la
    salida (de la cual hay una sola) es una entrada con valor Laberin-
    to.SALIDA ('2') y todas las demás tienen valor Laberinto.LIBRE
    ('0').
    El primer valor determina la fila y el segundo la columna (comen-
    zando desde cero).
    */
    private int[][] lab;

    /*
    Posicion del jugador:
    La posicion del jugador debera siempre estar dentro del laberinto,
    y no puede haber una pared en esa misma posicion.
    */
    private int jugadorFila, jugadorColumna;

    public Laberinto(int[][] lab, int filaComienzo, int
                columnaComienzo) {
          this.lab = lab;
          // Poner al jugador en la posicion de comienzo
          this.jugadorFila = filaComienzo;
          this.jugadorColumna = columnaComienzo;
    }

    /*
    Devuelve verdadero si el jugador ha llegado a la salida.
    */
    public boolean gano() {
          // AUN NO IMPLEMENTADO...
    }

    /*
    Un movimiento a la posicion (nuevaFila,nuevaColumna) es posible si
    y solo si la celda esta dentro del laberinto y esta libre.
    */
    public boolean esPosible(int nuevaFila, int nuevaColumna) {
          // AUN NO IMPLEMENTADO...
    }

    /*
    Toma una direccion y ejecuta el movimiento si es posible. Sino, el
    jugador permanence en la misma posicion. La direccion debe ser una
    de las definidas por las constantes MOVER_X. El valor devuelto in-
    dica si el movimiento tuvo exito.
    */
    public boolean mover(int direccion) {
          // AUN NO IMPLEMENTADO...
    }
}
```

Escriba las siguientes especificaciones en JML:

1. Invariantes que especifiquen las condiciones:
   a) El laberinto no es `null`; es decir, el campo de juego en sí y sus componentes no son `null`.
   b) Cada fila del laberinto tiene la misma cantidad de columnas.
   c) La posición ocupada por el jugador no es una pared.
   d) Cada celda del laberinto es o bien una pared, la salida, o está libre.
   e) Hay exactamente una salida.

2. Especificaciones `normal_behavior` para cada método que reflejen los comentarios en castellano lo más cercanamente posible.

Ejercicio 6:

¿Qué hace el siguiente método, suponiendo que `x` es un número entero positivo?

```
public int x;

public int ejercicio6() {
       int  y = x; int  z = 0;
       while (y > 0) {
              z = z + x;
              y = y - 1;
       return z;        }
```

a) Escriba especificaciones en JML para el método e invariantes para la clase.
b) Demuestre correctitud parcial del método mediante un invariante de bucle.
c) ¿Es posible demostrar correctitud total? Justifique su respuesta.

Ejercicio 7:

Considere la siguiente clase con un sólo método:

```
public class AyudaArreglo {
/*@ public normal_behavior
  @ requires nuevoElem > arreglo[pos];
  @ ensures arreglo[pos] == nuevoElem;
  @
  @ also
  @
  @ public normal_behavior
  @ requires nuevoElem <= arreglo[pos];
  @ ensures true;
  @ assignable \nothing;
@*/
public static void reemplazarSiMayor(int nuevoElem,
                                      int pos, int[] arreglo) {
      if (nuevoElem > arreglo[pos]) {
          arreglo[pos] = nuevoElem;
      }
}}
```

a) Utilice KeY para verificar si el método `reemplazarSiMayor` respeta sus contratos.
b) Analice la prueba que arroja la herramienta y explique lo que está sucediendo.

c) Arregle el error en la especificación o la implementación y verifique los contratos del método nuevamente.