Especificacion de TADs

Ejercicio 1 Especificar en forma completa el TAD `NumeroRacional` que incluya al menos las operaciones aritmeticas basicas

TAD `NumeroRacional` {

`obs n:` $\mathbb{Z}$

`obs d:` $\mathbb{Z}$

`proc suma (in a,b: NumeroRacional) : NumeroRacional`

`requiere` $\{true\}$

`asegura` $\{res = \frac{a.n*b.d+a.d*b.n}{b.d*a.d}\}$

`proc resta (in a,b: NumeroRacional) : NumeroRacional`

`requiere` $\{true\}$

`asegura` $\{res = \frac{a.n*b.d-a.d*b.n}{b.d*a.d}\}$

`proc multiplicacion (in a,b: NumeroRacional) : NumeroRacional`

`requiere` $\{true\}$

`asegura` $\{res = \frac{a.n*b.n}{b.d*a.d}\}$

`proc division (in a,b: NumeroRacional) : NumeroRacional`

`requiere` $\{b \neq 0\}$

`asegura` $\{res = \frac{a.n*b.d}{b.n*a.d}\}$

}

Ejercicio 2: Especificar TADs para las siguientes figuras geometricas. Tiene que contener las operaciones rotar, trasladar y escalar, y una mas propuesta por usted.

- Rectangulo (2D)
- Esfera (3D)

TAD `Rectangulo` {

`obs alto:` $\mathbb{R}$

`obs ancho:` $\mathbb{R}$

`obs posicion:` $\langle \mathbb{R} \times \mathbb{R} \rangle$

`obs angulo:` $\mathbb{R}$

`proc nuevoRectangulo (in h:` $\mathbb{R}$`, in w: float, in x:`$\mathbb{R}$`,in y:`$\mathbb{R}$`) : Rectangulo`

`requiere` $\{w \geq 0 \wedge h \geq 0\}$

`asegura` $\{res.ancho = w \wedge res.alto = h \wedge res.posicion = (x, y) \wedge r.angulo = 0\}$

`proc escalar (inout r: Rectangulo, mh:`$\mathbb{R}$`, mw:`$\mathbb{R}$`)`

`requiere` $\{r = R_0\}$

`asegura` $\{(mw \geq 0 \wedge mh \geq 0) \wedge_L r.posicion = R_0.posicion\}$

`asegura` $\{r.alto = |R_0.alto * mh| \wedge r.ancho = |R_0.ancho * mw|\}$

`asegura` $\{mw < 0 \wedge_L r.posicion_1 = R_0.posicion_1 + R_0.ancho * mw\}$

`asegura` $\{mh < 0 \wedge_L r.posicion_2 = R_0.posicion_2 + R_0.alto * mh\}$

`proc rotar (inout r: Rectangulo, in rad:` $\mathbb{R}$`)`

`requiere` $\{r = R_0\}$

`asegura` $\{r.angulo = R_0.angulo + rad\}$

`proc trasladar (inout r: Rectangulo, in pos:` $\langle x, y \rangle$`)`

`requiere` $\{r = R_0\}$

`asegura` $\{r.posicion = R_0.posicion + pos\}$

`proc area (in r: Rectangulo) :` $\mathbb{R}$

`asegura` $\{res = r.alto \cdot r.ancho\}$

}

TAD Esfera {

obs radio: $\mathbb{R}$

obs centro: $\langle \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rangle$

obs semieje: $\langle \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rangle$

proc nuevaEsfera (in r: $\mathbb{R}$, in pos:$\langle \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rangle$) : Esfera

asegura $\{res.radio = |r|\}$

asegura $\{res.centro = pos\}$

proc escalar (inout esfera: Esfera, in e: $\mathbb{R}$)

requiere $\{esfera = esfera_0\}$

asegura $\{esfera.radio = ESFERA_0.radio * |e|\}$

proc trasladar (inout esfera: Esfera, in pos:$\langle x, y, z \rangle$)

requiere $\{esfera = ESFERA_0\}$

asegura $\{esfera.centro = ESFERA_0.centro + pos\}$

proc rotar (inout esfera: Esfera, in angulosrad: $\langle \alpha, \beta, \gamma \rangle$)

requiere $\{esfera = ESFERA_0\}$

asegura $\{esfera.semieje = ESFERA_0.semieje + angulosrad\}$

}

Ejercicio 3. Especifique el TAD DobleCola$\langle T \rangle$, en el que los elementos pueden insertarse al principio o al final y se eliminan por el medio.

TAD DobleCola {

obs elems: $seq\langle T \rangle$

proc DobleColaVacia () : DobleCola

asegura $\{res = \langle\rangle\}$

proc encolarFinal (inout doblecola: DobleCola, in e: T)

requiere $\{doblecola = DOBLECOLA_0\}$

asegura $\{doblecola.elems = concat(DOBLECOLA_0.elems, \langle e \rangle)\}$

proc encolarInicio (inout doblecola: DobleCola, in e: T)

requiere $\{doblecola = DOBLECOLA_0\}$

asegura $\{doblecola.elems = concat(\langle e \rangle, DOBLECOLA_0.elems)\}$

proc desencolar (inout doblecola: DobleCola) : T

requiere $\{doblecola = DOBLECOLA_0\}$

requiere $\{cola.elems \neq \langle\rangle\}$

asegura $\{res \notin doblecola.elems\}$

asegura $\{res = DOBLECOLA_0.elems\left[\left\lceil \frac{|DOBLECOLA_0|}{2} \right\rceil\right]\}$

}

Ejercicio 4. Especifique el TAD DiccionarioConHistoria. El mismo guarda para cada clave, todos los valores que se asociaron con la misma a lo largo del tiempo (en orden)

TAD `DiccionarioConHistoriaT`,$seq\langle K\rangle$ {

`obs data: dict<T,`$seq\langle K\rangle$`>`

`proc DiccionarioConHistoriaT,`$seq\langle K\rangle$ `Vacio () : DiccionarioConHistoriaT,`$seq\langle K\rangle$
`asegura` $\{res.data = \{\}\}$

`proc estaLaLlave (in dic:DiccionarioConHistoriaT,`$seq\langle K\rangle$ `, in e: T) : Bool`
`asegura` $\{res = true \longleftrightarrow e \in dic.data\}$

`proc definir (inout dic: DiccionarioConHistoriaT,`$seq\langle K\rangle$ `,in k: T, in e: K)`
`requiere` $\{DIC_0 = dic \wedge_L k \in RES_0.data\}$
`asegura` $\{dic.data[k][0] = setKey(DIC_0.data, k, concat(DIC_0.data[k], e))\}$

`proc consultarHistorial (in dic: DiccionarioConHistoriaT,`$seq\langle K\rangle$ `, in k: T) :` $seq\langle K\rangle$
`asegura` $\{res = res.data[k]\}$

`proc borrar (inout dic DiccionarioConHistoriaT,`$seq\langle K\rangle$ `,in k: T)`
`requiere` $\{dic = DIC_0\}$
`requiere` $\{k \in dic.data\}$
`asegura` $\{dic.data = delKey(DIC_0, k)\}$

`proc tamaño (in dic: DiccionarioConHistoriaT,`$seq\langle K\rangle$ `) :` $\mathbb{Z}$
`asegura` $\{res = |dic.data|\}$

}

Ejercicio 5. Modifique el TAD ColaDePrioridad¡T¿para que, si hay muchos valores iguales al maximo, la operacion desapilarMax los desapile a todos.

TAD `ColaDePrioridad<T>` {

`---Toda la implementacion normal de ColaDePrioridad<T>---`

`proc desapilarMax (inout cola:ColaDePrioridad<T>)` : $seq\langle T\rangle$
`requiere` $\{cola = COLA_0\}$
`asegura` $\{\nexists\ e \in cola.data, \exists e' \in COLA_0.data : tienePriMax(COLA_0.data, e') \wedge_L cola.data[e] = COLA_0.data[e']\}$
`asegura` $\{(\forall e \in COLA_0.data)\ (tienePriMax(COLA_0.data, e) \wedge_L e \in res) \vee$
$(\neg tienePriMax(COLA_0.data, e) \wedge_L e \notin res)\}$
`asegura` $\{|res| = |COLA_0.data| - |cola.data|\}$

}

No estoy seguro de si se puede usar el operando $\in$ con diccionarios, ni si estoy asegurando que TODOS los elementos que saco de la cola sean incluidos en la sequencia de resultado, por ejemplo si tengo dos elementos en la cola con el mismo nombre y la misma prioridad, ambos van a la secuencia con la definicion que di? creo que no, que como uno ya pertenece el otro no necesariamente se agrega. ¿Comparar la cantidad que saco y el largo de la secuencia puede ser una opcion valida?

Ejercicio 6. Especifique los TADS indicados a continuacion pero utilizando los observadores propuestos

- Diccionario<K,V> observando con conjuntos (de tuplas)
- Conjunto<T> observando con funciones
- Pila<T> observando con diccionarios
- Punto observando con coordenadas polares

TAD Diccionario<K,V> {
obs datos: conj$\langle K \times V \rangle$

proc diccionarioVacio () : Diccionario<K,V>
requiere $\{true\}$
asegura $\{res.datos = \{\}\}$

proc agregar (inout dic: Diccionario<K,V>, in k: K, in v: V)
requiere $\{dic = DIC_0\}$
asegura $\{(k, v) \in dic.datos\}$

---Se puede hacer siquiera esto? Otro observador con otro conjunto de tuplas y apariciones? raro

}

TAD Conjunto<T> {
obs pertenece: Bool
obs largo: $\mathbb{Z}$

proc conjVacio () : Conjunto<T>
requiere $\{true\}$
asegura $\{res.largo = 0\}$

proc agregar (inout c: Conjunto<T>, in e: T)
requiere $\{c = C_0\}$
asegura $\{c.pertenece(e)\}$

proc sacar (inout c: Conjunto<T>, in e: T)
requiere $\{c = C_0\}$
asegura $\{\neg c.pertenece(e)\}$

proc unir (inout c: Conjunto<T>, in c': Conjunto<T>)
requiere $\{c = C_0\}$
asegura $\{\forall e \in T, c'.pertenece(e) : c.pertenece(e)\}$

proc restar (inout c: Conjunto<T>, in c': Conjunto<T>)
requiere $\{c = C_0\}$
asegura $\{\forall e \in T, c'.pertenece(e) : \neg c.pertenece(e)\}$

proc intersecar (inout c: Conjunto<T>, in c': Conjunto<T>)
requiere $\{c = C_0\}$
asegura $\{\forall e \in T, C_0.pertenece(e) \wedge c'.pertenece(e) \leftrightarrow c.pertenece(e)\}$

proc tamaño (in c: Conjunto<T>) : $\mathbb{N}$
requiere $\{true\}$
asegura $\{\exists! m \in \mathbb{Z} : \sum_{i=1}^{m} \text{if } c.pertenece(e) \text{ then } 1 \text{ else } 0 \text{ fi} = res, e \in T\}$
--- acá cree el observador largo porque muchas cosas raras tenia que hacer
asegura $\{res = c.largo\}$

}

TAD Pila<T> {
obs elems: dict<$\mathbb{N}$,T>

proc pilaVacia () : Pila<T>
requiere $\{true\}$
asegura $\{res.elems = \{\}\}$

proc vacia (in pila: Pila<T>) : Bool
requiere $\{true\}$
asegura $\{res = true \leftrightarrow pila.elems = \{\}\}$

proc apilar (inout pila: Pila<T>, in e: T)
requiere $\{pila = PILA_0\}$
asegura $\{|PILA_0.elems|+1 = |pila.elems| \wedge |pila.elems| \in pila.elems \wedge_L setKey(pila.elems, |pila.elems|, e)\}$

```
    proc desapilar (inout pila: Pila<T>) : T
          requiere {pila = PILA_0}
          requiere {pila.elems ≠ {}}
          asegura {|pila.elems| = |PILA_0.elems| − 1 ∧ |PILA_0.elems| ∉ pila.elems}
          asegura {res = PILA_0.elems[|PILA_0.elems|]}

    proc tope (in pila: Pila<T>) : T
          requiere {pila.elems ≠ {}}
          asegura {res = pila.elems[|pila.elems| − 1]}
}

TAD Punto {
    obs coords: ⟨ℝ × ℝ⟩
    pred igualdad (p1, p2: Punto) {
        (∃k ∈ ℤ : p1.coords_2 = p2.coords_2 + 2kπ) ∧ (p1.coords_1 = p2.coords_1)
    }
    proc crearPunto (in r: ℝ, in α: ℝ) : Punto
          requiere {r ≥ 0}
          asegura {res.coords = (r, α)}

    proc mover (inout p: Punto, in newcoords: ⟨ℝ × ℝ⟩)
          requiere {true}
          asegura {p.coords_1 = newcoords_1 ∧ p.coords_2 = newcoords_2}
}
```

Ejercicio 7. Especificar TADs para las siguientes estructuras:

- Multiconjunto<T> - Es igual a un conjunto pero con duplicados. Cada elemento tiene asociada una multiplicidad, que es la cantidad de veces que este aparece en la estrucutra. Tiene las mismas operaciones que un conjunto y ademas una que indica la multiplicidad del elemento.

```
TAD Multiconjunto {
      obs elems: dict<T,ℕ>

      proc conjuntoVacio () : Multiconjunto
            requiere {true}
            asegura {res.elems = {}}

      proc agregar (inout mc: Multiconjunto, in e: T)
            requiere {mc = MC_0}
            asegura {e ∈ MC_0.elems ∧_L mc.elems[e] = MC_0.elems[e] + 1}
            asegura {e ∉ MC_0.elems ∧_L setKey(mc, e, 1)}

      proc eliminar (inout mc: Multiconjunto)
            requiere {mc = MC_0}
            asegura {e ∈ MC_0.elems ∧_L MC_0.elems[e] = 1 ∧_L mc.elems = delKey(MC_0, e)}
            asegura {e ∈ M_0.elems ∧_L MC_0.elems[e] > 1 ∧_L mc.elems = setKey(MC_0, e, MC_0.elems[e] − 1)}

      proc multiplicidad (in mc: Multiconjunto, in e: T) : ℤ
            requiere {true}
            asegura {e ∈ mc ∧_L res = mc.elems[e]}
            asegura {e ∉ mc ∧_L res = 0}

      --- Hay mas operaciones pero creo que estas son las mas relevantes
}
```

- Multidict<K,V>: Misma idea pero para diccionarios, cada clave puede estar asociada a multiples valores. Los valores se definen de a uno, pero la operacion obtener debe devolver todos los valores asociados a una determinada clave.

  Sobre la nota(que no copie): una posible implementacion que se me ocurre es la de un taller en el que las keys son los operarios y los values son listas en las que estan los trabajos pendientes que le corresponden a cada uno, en una implementacion mas completa se podria hacer procs para mover los trabajos de la cola (porque la secuencia va a ser basicamente una cola en su comportamiento default) pero que tambien se pueda elegir trabajos especificos que hacer para darles prioridad.

  TAD `Multidict<K,V>` {

  `obs elems: dict<K,`$seq\langle V\rangle$`>`

  `proc multidictVacio () : Multidict<K,V>`

  `requiere` $\{true\}$

  `asegura` $\{res.elems = \{\}\}$

  `proc agregar (inout md: Multidict<K,V>, in k: K, in v: V)`

  `requiere` $\{md = MD_0\}$

  `asegura` $\{k \in MD_0.elems \wedge_L setKey(md.elems, k, concat(\langle v\rangle, MD_0.elems[k]))\}$

  `asegura` $\{k \notin MD_0.elems \wedge_L setKey(md.elems, k, \langle v\rangle)\}$

  `proc borrar (inout md: Multidict<K,V>, in k: K)`

  `requiere` $\{md = MD_0\}$

  `asegura` $\{k \notin MD_0 \wedge_L md = MD_0\}$

  `asegura` $\{k \in MD_0.elems \wedge |MD_0.elems| = 1 \wedge_L delKey(md.elems, k)\}$

  `asegura` $\{k \in MD_0.elems \wedge |MD_0.elems| > 0 \wedge_L$
  $setKey(md.elems, k, subseq(MD_0.elems, 1, |MD_0.elems|[k]))\}$

  `--- Creo que el resto de la implementacion es trivial, podria agregarse un booleando como parametr`
  `de borrar que determine si se borra la key completamente o si se borra valor a valor`

  }

Ejercicio 8. Especifique el TAD contadores que, dada una lista de eventos, permite contar la cantidad de veces que se produjo cada uno de ellos. La lista de eventos es fija. El TAD debe tener una operacion para incrementar el contador asociado a un evento y una operacion para conocer el valor actual del contador de dicho evento.

-Modifique el TAD para que sea posible preguntar el valor del contador en un determinado momento del pasado. Si necesita conocer la fecha y hora actual puede pasarla como parametro a los procedimientos. Asuma que las dechas son numeros enteros (por ejemplo la cantidad de segundos desde el 1ro de enero del '70)

TAD `Contadores` {

`obs elems: dict<T,`$\mathbb{Z}$`>`

`proc nuevosContadores (in l:` $seq\langle T\rangle$`) : Contadores`

`requiere` $\{true\}$

`asegura` $\{|c.elems| = |l|\}$

`asegura` $\{\forall e \in l : e \in res.elems \wedge_L res[e] = 0\}$

`proc aumentarContador (inout c: Contadores, in k: T)`

`requiere` $\{c = C_0\}$

`requiere` $\{k \in c.elems\}$

`asegura` $\{|c.elems| = |C_0.elems|\}$

`asegura` $\{\forall i \in c.elems : (i = k \wedge_L c.elems[k] = C_0.elems[k] + 1) \vee (i \neq k \wedge_L c.elems[k] = C_0.elems[k])\}$

`proc numeroEventos (in c: Contadores, in k: T) :` $\mathbb{Z}$

`requiere` $\{k \in c.elems\}$

`asegura` $\{res = c.elems[k]\}$

}

Contador con historial:

TAD ContadoresHistorial {
obs elems: dict<T,$\mathbb{Z}$>
obs tiempo: $\mathbb{Z}$

proc nuevosContadores (in l: $seq\langle T\rangle$,in fecha: $\mathbb{Z}$) : ContadoresHistorial
requiere $\{true\}$
asegura $\{|c.elems| = |l|\}$
asegura $\{\forall e \in l : e \in res.elems \wedge_L res.elems[e] = 0 \wedge res.tiempo = fecha\}$

proc aumentarContador (inout c: ContadoresHistorial, in k: T, in fecha: $\mathbb{Z}$)
requiere $\{c = C_0\}$
requiere $\{k \in c.elems\}$
asegura $\{|c.elems| = |C_0.elems|\}$
asegura $\{\forall i \in c.elems : (i = k \wedge_L c.elems[k] = C_0.elems[k] + 1 \wedge c.tiempo = fecha) \vee (i \neq k \wedge_L c.elems[k] = C_0.elems[k])\}$

proc numeroEventos (in c: ContadoresHistorial, in k: T) : $\mathbb{Z}$
requiere $\{k \in c.elems\}$
asegura $\{res = c.elems[k]\}$

}
- muy confuso, lo dejo para despues

Ejercicio 9. Supongamos que queremos utilizar el TAD contador en un sistema que procesa millones de eventos por segundo y no damos abasto para procesar todos los eventos. Una posible solución es hacer sampling: en lugar de contar cada evento, contamos un porcentaje (configurable) de ellos, por ejemplo, un 1 %. Es decir que de todas las llamadas al proc incrementar, sólo el 1 % de ellas efectivamente incrementa el contador.

En mi opinion es incorrecta porque no se deberia llamar a incrementar y ahi decir si se incrementa o no, en su lugar deberian seleccionarse solo el 1 % de los eventos y que esos eventos llamen a incrementar.
Entonces lo que haria seria dejar la especificacion normal de incrementar y descartar los eventos antes. Por ejemplo con un $\forall i \in \mathbb{Z}, 0 \leq i, i mod 99 = 0$ (porque son millones por segundo, en otro caso no se como se haria.)

tl;dr: No sé

Ejercicio 10. Un caché es una capa de almacenamiento de datos de alta velocidad que almacena un subconjunto de datos, normalmente transitorios, de modo que las solicitudes futuras de dichos datos se atienden con mayor rapidez que si se debe acceder a los datos desde la ubicación de almacenamiento principal. El almacenamiento en caché permite reutilizar de forma eficaz los datos recuperados o procesados anteriormente. Esta estructura comunmente tiene una interface de diccionario: guarda valores asociados a claves, con la diferencia de que los datos almacenados en un cache pueden desaparecer en cualquier momento, en función de diferentes criterios. Especificar caches genéricos (con claves de tipo K y valores de tipo V) que respeten las operaciones indicadas y las siguientes políticas de borrado automático. Si necesita conocer la fecha y hora actual, puede pasarla como parámetro a los procedimientos o bien puede asumir que existe una función externa horaActual() : Z que retorna la fecha y hora actual. Asuma que las fechas son números enteros (por ejemplo, la cantidad de segundos desde el 1 de enero de 1970).

TAD Cache<K,V> {
proc esta (in c:Cache<K,V>) : Bool

proc obtener (in c:Cache<K,V>, in k: K) : V

proc definir (inout c:Cache<K,V>, in k: K) : V

}

Fifo o first-in-first-out
El cache tiene una capacidad maxima. si se alcanza esa capacidad maxima se borra automaticamente la clave que fue definida por primera vez hace mas tiempo.

TAD Cache<K,V> {
obs elems: struct<dict:dict<K,V>, tiempo:$\mathbb{Z}$>
obs llena: Bool

proc esta (in c:Cache<K,V>, in k: K) : Bool
requiere $\{true\}$
asegura $\{res = true \leftrightarrow k \in c.elems_{dict}\}$

proc obtener (in c:Cache<K,V>, in k: K) : V
requiere $\{k \in c.elems_{dict}\}$
asegura $\{res = c.elems_{dict}[k]\}$

proc definir (inout c:Cache<K,V>, in k: K, in v: V)
requiere $\{c = C_0\}$
asegura $\{(C_0.llena \wedge_L (\exists e \in C_0.elems : e_{tiempo} = min(C_0.elems_{tiempo}) \wedge_L c.elems_{dict} = setKey(delKey(c.elems_{dict}, e)), k, v))) \vee_L (\neg C_0.llena \wedge_L setKey(c.elems_{dict}, k, v))\}$
asegura $\{res.elems_{tiempo} = fechaActual()\}$

}

————Flashe una banda, mejor ni terminarlo y arrancar denuevo. (Defini mal mi observador y solo tengo una fecha jdlksjf)

TAD Cache<K,V> {
obs datos: dict<K,struct$\langle valor : V, tiempo : \mathbb{Z}\rangle$>
obs lleno: Bool

proc esta (in c:Cache<K,V>, in k: K) : Bool
asegura $\{res = true \leftrightarrow k \in c.datos\}$

proc obtener (in c:Cache<K,V>, in k: K) : V
requiere $\{k \in c.datos\}$
asegura $\{res = c.datos[k]_{valor}\}$

proc definir (inout c:Cache<K,V>, in k: K, in v: V)
requiere $\{c = C_0\}$
asegura $\{C_0.lleno \wedge_L \exists e \in C_0.datos : (\forall e' \in C_0.datos : e_{tiempo} \leq e'_{tiempo}) \wedge_L c.datos = setKey(delKey(C_0.datos, e), k, \langle v, fechaActual() \rangle)\}$
asegura $\{\neg C_0.lleno \wedge_L c.datos = setKey(C_0.datos, k, \langle v, fechaActual() \rangle)\}$

} LRU o last-recently-used

TAD Cache<K,V> {
obs datos: dict<K,struct$\langle valor : V, tiempo : \mathbb{Z}\rangle$>
obs lleno: Bool

proc esta (in c:Cache<K,V>, in k: K) : Bool
asegura $\{res = true \leftrightarrow k \in c.datos\}$

proc obtener (inout c:Cache<K,V>, in k: K) : V
requiere $\{c = C_0\}$
requiere $\{k \in C_0.datos\}$
asegura $\{res = C_0.datos[k]_{valor}\}$
asegura $\{c.datos[k]_{tiempo} = fechaActual()\}$

proc definir (inout c:Cache<K,V>, in k: K, in v: V)
requiere $\{c = C_0\}$
asegura $\{C_0.lleno \wedge_L \exists e \in C_0.datos : (\forall e' \in C_0.datos : e_{tiempo} \leq e'_{tiempo}) \wedge_L c.datos = setKey(delKey(C_0.datos, e), k, \langle v, fechaActual() \rangle)\}$
asegura $\{\neg C_0.lleno \wedge_L c.datos = setKey(C_0.datos, k, \langle v, fechaActual() \rangle)\}$

}

Entiendo el concepto del TTL pero no veo como aplicar algo que deberia ser automatico a lenguaje especificacion. Sobre los primeros dos creo que estan bien ahora si.

Ejercicio 12. Queremos modelar el funcionamiento de un vivero. El vivero arranca su actividad sin ninguna planta y con un monto inicial de dinero. Las plantas las compramos en un mayorista que nos vende la cantidad que deseemos pero solamente de a una especie por vez. Como vivimos en un país con inflación, cada vez que vamos a comprar tenemos un precio diferente para la misma planta. Para poder comprar plantas tenemos que tener suficiente dinero disponible, ya que el mayorista no acepta fiarnos. A cada planta le ponemos un precio de venta por unidad. Ese precio tenemos que poder cambiarlo todas las veces que necesitemos. Para simplificar el problema, asumimos que las plantas las vendemos de a un ejemplar (cada venta involucra un solo ejemplar de una única especie). Por supuesto que para poder hacer una venta tenemos que tener stock de esa planta y tenemos que haberle fijado un precio previamente. Además, se quiere saber en todo momento cuál es el balance de caja, es decir, el dinero que tiene disponible el vivero.

TAD `vivero` {

`obs dinero:` $\mathbb{R}$

`obs plantas: dict<E,`$\mathbb{N}_0$`>`

`obs catalogo: dict<E,`$\mathbb{R}$`>`

`obs preciosMayorista: dict<E,`$\mathbb{R}$`>`

`proc comprarPlantas (inout v: vivero, in especie: E, in cantidad:` $\mathbb{N}$`)`

`requiere` $\{v = V_0\}$

`requiere` $\{V_0.dinero > V_0.preciosMayorista[especie] * cantidad\}$

`asegura` $\{especie \in V_0.plantas \wedge_L v.plantas = setKey(V_0.plantas, especie, V_0.plantas[especie] + cantidad)\}$

`asegura` $\{especie \notin V_0.plantas \wedge_L v.plantas = setKey(V_0.plantas, especie, cantidad)\}$

`asegura` $\{v.dinero = V_0.dinero - V_0.preciosMayorista[especie] * cantidad\}$

`asegura` $\{especie \in V_0.catalogo \wedge_L$
$v.catalogo = setKey(V_0.catalogo, especie, max(V_0.catalogo[especie], V_0.preciosMayorista[especie]))\}$

`asegura` $\{especie \notin V_0.catalogo \wedge_L v.catalogo = setKey(V_0.catalogo, especie, 0))\}$

`asegura` $\{v.preciosMayorista > V_0.preciosMayorista\}$

`proc ponerPrecio (inout v: vivero, in especie: E, in precio:` $\mathbb{R}$`)`

`requiere` $\{v = V_0\}$

`asegura` $\{v.catalogo = setKey(V_0.catalogo, especie, precio)\}$

`asegura` $\{v.preciosMayorista = V_0.preciosMayorista\}$

`asegura` $\{v.plantas = V_0.plantas\}$

`asegura` $\{v.dinero = V_0.dinero\}$

`proc venderPlanta (inout v: vivero, in especie: E)`

`requiere` $\{v = V_0\}$

`requiere` $\{V_0.catalogo[especie] \neq 0\}$

`requiere` $\{V_0.plantas[especie] > 0\}$

`asegura` $\{v.dinero = V_0.dinero + V_0.catalogo[especie]\}$

`asegura` $\{v.plantas = setKey(V_0.plantas, especie, V_0.plantas[especie] - 1)\}$

`asegura` $\{v.preciosMayorista = V_0.preciosMayorista\}$

`asegura` $\{v.catalogo = V_0.catalogo\}$

`proc consultarCaja (in vivero: vivero) :` $\mathbb{R}$

`asegura` $\{res = vivero.dinero\}$

`--- el resto de consultas son triviales`

}