

# Especificacion de TADs

## Ejercicio 1

```
TAD NumeroRac{
  obs num:   $\mathbb{Z}$ 
  obs den:   $\mathbb{Z}$ 

  proc nuevoRac (in n,d: $\mathbb{Z}$ ):NumeroRac {
    requiere {d  $\neq$  0}
    asegura {res.num = n  $\wedge$  res.den = d}
  }

  proc suma (inout a:NumeroRac, in b:NumeroRac):NumeroRac {
    requiere {a = A0}
    asegura {a.num = A0.num * b.den + A0.den * b.num}
  }

  proc resta (inout a:NumeroRac, in b:NumeroRac):NumeroRac {
    requiere {a = A0}
    asegura {a.num = A0.num * b.den - A0.den * b.num}
  }

  proc multiplicacion (inout a:NumeroRac, in b:NumeroRac):NumeroRac {
    requiere {a = A0}
    asegura {a.num = A0.num * b.num  $\wedge$  a.den = A0.den * b.den}
  }

  proc division (inout a:NumeroRac, in b:NumeroRac):NumeroRac {
    requiere {a = A0}
    asegura {a.num = A0.num * b.den  $\wedge$  a.den = A0.den * b.num}
  }

  proc iguales (in a,b:NumeroRac):Bool {
    requiere {True}
    asegura {res = True  $\leftrightarrow$  a.num * b.den = a.den * b.num}
  }
}
```

## Ejercicio 2

```
TAD Rectangulo2D{
  obs vsi:  tupla( $\mathbb{R}$ ,  $\mathbb{R}$ )
  obs vsd:  tupla( $\mathbb{R}$ ,  $\mathbb{R}$ )
  obs vii:  tupla( $\mathbb{R}$ ,  $\mathbb{R}$ )
  obs vid:  tupla( $\mathbb{R}$ ,  $\mathbb{R}$ )
  aux resta (in a,b:tupla( $\mathbb{R}$ ,  $\mathbb{R}$ )):tupla( $\mathbb{R}$ ,  $\mathbb{R}$ )=(a0 - b0, a1 - b1)
  aux prod (in a,b:tupla( $\mathbb{R}$ ,  $\mathbb{R}$ )):  $\mathbb{R}$ =a0 * b0 + a1 * b1
  pred noRepes (in t:seq( $\mathbb{R}$ )) {
    ( $\forall i,j:\mathbb{Z}$ )(0  $\leq$  i, j < |t|  $\wedge$  i  $\neq$  j  $\rightarrow$  t[i]  $\neq$  t[j])
  }
}
```

```

proc nuevoRectangulo (in a,b,c,d:ℝ):Rectangulo2D {
  requiere {noRepes(⟨a,b,c,d⟩)}
  asegura {prod(resta(a.vsi,b.vsd),resta(a.vsi,c.vii)) = 0}
  asegura {prod(resta(d.vid,b.vsd),resta(d.vid,c.vii)) = 0}
  asegura {prod(resta(c.vii,a.vsi),resta(c.vii,d.vid)) = 0}
  asegura {prod(resta(b.vsd,a.vsi),resta(b.vsd,d.vid)) = 0}
}

proc mover (inout r:Rectangulo2D,in dx,dy:ℝ):Rectangulo2D {
  requiere {r = R0}
  asegura {r.vsi = (R0.vsi0 + dx, R0.vsi1 + dy)}
  asegura {r.vsi = (R0.vsd0 + dx, R0.vsd1 + dy)}
  asegura {r.vsi = (R0.vii0 + dx, R0.vii1 + dy)}
  asegura {r.vsi = (R0.vid0 + dx, R0.vid1 + dy)}
}

```

### Ejercicio 3

a) TAD Cola $\langle T \rangle$

```

obs s: seq⟨T⟩

proc nuevaCola (in s:seq⟨T⟩):cola⟨T⟩ {
  requiere {True}
  asegura {|res.s| = 0}
}

proc estaVacía (in c:cola⟨T⟩):Bool {
  requiere {True}
  asegura {res = True ↔ |c.s| = 0}
}

proc encolar (inout c:cola⟨T⟩,e:T):cola⟨T⟩ {
  requiere {c = C0}
  asegura {c.s = concat(C0.s,{e})}
}

proc desencolar (inout c:cola⟨T⟩):T {
  requiere {c = C0}
  asegura {c.s = tail(C0.s) ∧ res = head(C0.s)}
}

```

b) TAD Pila $\langle T \rangle$

```

obs s: seq⟨T⟩

proc nuevaPila (in s:seq⟨T⟩):pila⟨T⟩ {
  requiere {True}
  asegura {|res.s| = 0}
}

```

```

proc estaVacía (in c : pila(T)) : Bool {
  requiere {True}
  asegura {res = True  $\leftrightarrow$  |c.s| = 0}
}

proc apilar (inout c : pila(T), e : T) : pila(T) {
  requiere {c = C0}
  asegura {c.s = concat({e}, C0.s)}
}

proc desapilar (inout c : pila(T)) : T {
  requiere {c = C0}
  asegura {c.s = tail(C0.s)  $\wedge$  res = head(C0.s)}
}

```

c) TAD *dobleCola*(T){

```

  obs elems: seq(T)

```

```

proc nuevaDobleCola ( ) : dobleCola(T) {
  requiere {True}
  asegura {|res.elems| = 0}
}

```

```

proc estaVacía (in c : dobleCola(T)) : Bool {
  requiere {True}
  asegura {|res.elems| = 0}
}

```

```

proc encolarAdelante (inout c : dobleCola(T), in e : T) : {
  requiere {c = C0}
  asegura {c.elems = {e} ++ C0.elems}
}

```

```

proc encolarAtras (inout c : dobleCola(T), in e : T) : {
  requiere {c = C0}
  asegura {c.elems = C0.elems ++ {e}}
}

```

```

proc desencolar (inout c : dobleCola(T)) : T {
  requiere {c = C0}
  asegura {res = C0.elems[|C0.elems|/2]  $\wedge$  c.elems =
subseq(C0.elems, 0, |C0.elems|/2) ++ subseq(C0.elems, |C0.elems|/2, |C0.elems|)}
}

```

## Ejercicio 4

```

a) TAD Diccionario $\langle K, V \rangle$  {
    obs elem: conj $\langle \text{tupla}\langle K, V \rangle \rangle$ 

    proc nuevoDict ( ): Diccionario $\langle K, V \rangle$  {
        asegura {res.pares = 0}
    }

    proc definir (inout d: Diccionario $\langle K, V \rangle$ , in k: K, in t: T): {
        requiere {d = D0}
        asegura {(∃t': tupla $\langle K, V \rangle$ )(t' ∈ D0 ∧ t'0 = k) ⇔ d.pares = (D0.pares −  $\langle t' \rangle$ ) ∪  $\langle \langle k, t \rangle \rangle$ }
        asegura {¬(∃t': tupla $\langle K, V \rangle$ )(t' ∈ D0 ∧ t'0 = k) ⇔ d.pares = D0.pares ∪  $\langle \langle k, t \rangle \rangle$ }
    }

    proc obtener (in d: Diccionario $\langle K, V \rangle$ , in k: K): T {
        requiere {claveInDict(d, k)}
        asegura { $\langle k, res \rangle \in d.pares$ }
    }

    proc esta (in d: Diccionario $\langle K, V \rangle$ , in k: K): Bool {
        requiere {True}
        asegura {res = True ⇔ claveInDict(d, k)}
    }

    proc borrar (inout d: Diccionario $\langle K, V \rangle$ , in k: K): {
        requiere {d = D0 ∧ claveInDict(D0, k)}
        asegura {(∃t: T)( $\langle k, t \rangle \in D_0.pares$ ) ⇔ d.pares = D0.pares −  $\langle \langle k, t \rangle \rangle$ }
    }

    pred claveInDict (in d: Diccionario $\langle K, V \rangle$ , in k: K) {
        (∃t: tupla $\langle K, V \rangle$ )(t ∈ d.pares ∧ t0 = k)
    }

```

## Ejercicio 5

```

b) TAD conjunto $\langle T \rangle$  {
    obs esta(x: T): Bool

    proc nuevoConj ( ): conjunto $\langle T \rangle$  {
        asegura {(∀x: T)(¬res.esta(x))}
    }

    proc agregar (inout c: conjunto $\langle T \rangle$ , in e: T): {
        requiere {c = C0}
        asegura {C0.esta(e) ⇔ (∀x: T)(C0.esta(x) → c.esta(x))}
        asegura {¬C0.esta(e) ⇔ (∀x: T)(C0.esta(x) ∧ x ≠ e → c.esta(x) ∧ c.esta(e))}
    }

```

```

proc eliminar (inout c:conjunto⟨T⟩, in e:T): {
  requiere {c = C0 ∧ C0.esta(e)}
  asegura {(∀x:T)(C0.esta(x) ∧ x ≠ e ↔ c.esta(x) ∧ ¬c.esta(e))}
}

```

```

proc intersec (in a,b:conjunto⟨T⟩):conjunto⟨T⟩ {
  requiere {True}
  asegura {(∀x:T)(a.esta(x) ∧ b.esta(x) ↔ res.esta(x))}
}

```

```

proc union (in a,b:conjunto⟨T⟩):conjunto⟨T⟩ {
  requiere {True}
  asegura {(∀x:T)(a.esta(x) ∨ b.esta(x) ↔ res.esta(x))}
}

```

d) TAD *punto*{

obs *rad*: ℝ

obs *ang*: ℝ

```

proc crearPunto (in r,a:ℝ):punto {
  asegura {res.rad = r ∧ res.ang = a}
}

```

```

proc dist (in p1,p2:punto):ℝ {
  requiere {True}
  asegura {res = norma(x(p1) - x(p2), y(p1) - y(p2))}
}

```

```

proc distOrig (in p:punto):ℝ {
  requiere {True}
  asegura {norma(x(p), y(p))}
}

```

```

proc mover (inout p:punto, in dx,dy:ℝ): {
  requiere {p = P0}
  asegura {(∃r,a:ℝ)(r * cos(a) = P0.rad * cos(P0.ang) + dx ∧ r * sen(a) =
P0.rad * sen(P0.ang) + dy) ↔ p.rad = r ∧ p.ang = a}
}

```

aux x (in p:punto):ℝ=cos(p.ang) \* p.rad

aux y (in p:punto):ℝ=sen(p.ang) \* p.rad

aux norma (in x,y:ℝ):ℝ= $\sqrt{x^2 + y^2}$

## Ejercicio 6

a) TAD  $multiConjunto\langle T \rangle$

obs  $esta(x:T): Bool$

obs  $mult(x:T): \mathbb{Z}$

```
proc nuevoMultiConjunto ( ):multiConjunto⟨T⟩ {
  asegura  $\{(\forall t:T)(\neg res.esta(t) \leftrightarrow res.mult(t) = 0)\}$ 
}
```

```
proc agregar (inout c:multiConjunto⟨T⟩, in t:T): {
  requiere  $\{c = C_0\}$ 
  asegura  $\{(\forall x:T)(C_0.esta(x) \wedge x \neq t \leftrightarrow c.esta(x) \wedge c.esta(t) \wedge C_0.mult(x) = c.mult(x) \wedge c.mult(t) = C_0.mult(t) + 1)\}$ 
}
```

```
proc quitar (inout c:multiConjunto⟨T⟩, in t:T): {
  requiere  $\{c = C_0\}$ 
  asegura  $\{C_0.mult(t) > 1 \wedge (\forall x:T)(C_0.esta(x) \wedge x \neq t \leftrightarrow c.esta(x) \wedge c.esta(t) \wedge C_0.mult(x) = c.mult(x) \wedge c.mult(t) = C_0.mult(t) - 1)\}$ 
  asegura  $\{C_0.mult(t) \leq 1 \wedge (\forall x:T)(C_0.esta(x) \wedge x \neq t \leftrightarrow c.esta(x) \wedge \neg c.esta(t) \wedge C_0.mult(x) = c.mult(x) \wedge c.mult(t) = 0)\}$ 
}
```

```
proc pertenece (in c:multiConjunto⟨T⟩, in t:T):Bool {
  requiere  $\{True\}$ 
  asegura  $\{res = c.esta(t)\}$ 
}
```

```
proc multiplicidad (in c:multiConjunto⟨T⟩, in t:T): $\mathbb{Z}$  {
  requiere  $\{True\}$ 
  asegura  $\{res = c.mult(t)\}$ 
}
```

b) TAD  $multiDict\langle K, V \rangle$

obs  $data: seq\langle tupla\langle K, V \rangle \rangle$

```
proc nuevoMultiDict ( ):multiDict⟨K, V⟩ {
  asegura  $\{|res.data| = 0\}$ 
}
```

```
proc definir (inout md:multiDict⟨K, V⟩, in k:K, in t:T): {
  requiere  $\{md = Md_0\}$ 
  asegura  $\{cantApariciones(\langle K, V \rangle, md.data) = cantApariciones(\langle K, V \rangle, Md_0.data) + 1\}$ 
}
```

```

proc obtener (in md : multiDict⟨K, V⟩, in k : K) : seq⟨V⟩ {
  requiere {True}
  asegura {(∀t : tupla⟨K, V⟩)(t ∈ md.data ∧ t0 = k ↔ t1 ∈ res)}
}

proc pertenece (in md : multiDict⟨K, V⟩, in k : K) : Bool {
  requiere {True}
  asegura {res = True ↔ claveInDict(md, k)}
}

proc borrar (inout md : multiDict⟨K, V⟩, in k : K) : {
  requiere {md = Md0}
  asegura {(∀t : tupla⟨K, V⟩)(t ∈ Md0.data ∧ t0 = k ↔ ¬(t ∈ md.data))}
}

pred claveInDict (in md : multiDict⟨K, V⟩, in k : K) {
  (∃t : tupla⟨K, V⟩)(t ∈ md.data ∧ t0 = k)}

aux cantApariciones (in t : T, in s : seq⟨T⟩) : ℤ =  $\sum_{i=0}^{|s|-1} ifThenElse(s[i] = t, 1, 0)$ 

```

## Ejercicio 7

i) TAD *contador*{

```

obs list: seq⟨T⟩
obs cont(x : T): ℤ

```

```

proc nuevoContador (in s : seq⟨T⟩) : contador {
  asegura {(∀t : T)(t ∈ s → res.cont(t) = cantApariciones(t, s))}
}

```

```

proc cantEventos (in c : contador, in t : T) : ℤ {
  requiere {True}
  asegura {res = c.cont(t)}
}

```

```

proc incrementarEvento (inout c : contador, in t : T) : {
  requiere {t ∈ c.list ∧ c = C0}
  asegura {c.list = C0.list + +⟨t⟩ ∧ c.cont(t) = C0.cont(t) + 1}
}

```

ii) TAD *contador*{

```

obs list: seq⟨T⟩
obs cont(x : T): dict(ℤ, ℤ)

```

```

proc nuevoContador (in s : seq⟨T⟩, in fecha : ℤ) : contador {
  asegura {(∀t : T)(t ∈ s ↔ setKey(res.cont(t), fecha, cantApariciones(t, s)))}
  asegura {(∀t : T)(t ∈ s ↔ t ∈ c.list)}
}

```

```

proc cantEventos (in c : contador, in t : T, in fecha :  $\mathbb{Z}$ ) :  $\mathbb{Z}$  {
  requiere {True}
  asegura { $(\exists f : \mathbb{Z})(f \in c.cont(t) \wedge (\forall f' : \mathbb{Z})(f' \in c.cont(t) \Leftrightarrow f' \leq f \leq fecha)) \Leftrightarrow$ 
    res = d[f]}
}

proc incrementarEvento (inout c : contador, in t : T, in fecha :  $\mathbb{Z}$ ) : {
  requiere { $t \in c.list \wedge c = C_0$ }
  asegura { $c.list = C_0.list + +\langle t \rangle$ }
  asegura { $cantApariciones(t, c.list) = cantApariciones(t, C_0.list) + 1$ }
  asegura { $setKey(c.cont(t), fecha, cantApariciones(t, C_0.list) + 1)$ }
}

proc eventosPorFecha (in c : contador, in t : T, in fecha :  $\mathbb{Z}$ ) :  $\mathbb{Z}$  {
  requiere { $fecha \in c.cont(t)$ }
  asegura { $res = c.cont(t)[fecha]$ }
}

```

## Ejercicio 8

i) TAD  $CacheFIFO\langle K, V \rangle$

```

obs data: dict(K, V)
obs time: dict(K,  $\mathbb{Z}$ )
obs capacidad:  $\mathbb{Z}$ 

```

```

proc nuevoCacheFIFO (in cap :  $\mathbb{Z}$ ) :  $CacheFIFO\langle K, V \rangle$  {
  requiere { $cap \geq 0$ }
  asegura { $res.capacidad = cap$ }
  asegura { $|res.data| = 0 \wedge |res.time| = 0$ }
}

proc esta (in c :  $CacheFIFO\langle K, V \rangle$ , in k : K) : Bool {
  requiere {True}
  asegura { $res = True \Leftrightarrow k \in c.data \wedge k \in c.time$ }
}

proc obtener (in c :  $CacheFIFO\langle K, V \rangle$ , in k : K) : V {
  requiere { $k \in c.data \wedge k \in c.time$ }
  asegura { $res = c.data[k]$ }
}

proc definir (inout c :  $CacheFIFO\langle K, V \rangle$ , in k : K, in v : V) : {
  requiere { $c = C_0$ }
  asegura { $C_0.cap = 0 \Leftrightarrow c.cap = 0 \wedge c.data = \{\} \wedge c.time = \{\}$ }
  asegura { $C_0.cap = |C_0.data| \wedge k \in C_0.data \Leftrightarrow setKey(c.data, k, v) \wedge$ 
     $setKey(c.time, k, horaActual())$ }
  asegura { $C_0.cap = |C_0.data| \wedge \neg(k \in C_0.data) \Leftrightarrow setKey(delKey(c.data, m), k, v) \wedge$ 
     $setKey(delKey(c.time, m), k, horaActual()) \wedge claveMasVieja(C_0, m)$ }
}

```



```

    asegura  $\{C_0.cap > |C_0.data| \leftrightarrow setKey(c.data, k, v) \wedge setKey(c.time, k, horaActual())\}$ 
    asegura  $\{c.cap = C_0.cap\}$ 
}

```

```

pred claveMasVieja ( $c : CacheFIFO\langle K, V \rangle, k : K$ ) {
   $(\forall p : K)(p \in c.time \leftrightarrow c.time[p] \geq c.time[k])$ 
}

```

ii) TAD  $CacheLRU\langle K, V \rangle$

```

obs data: dict( $K, V$ )
obs time: dict( $K, \mathbb{Z}$ )
obs access: dict( $K, \mathbb{Z}$ )
obs capacidad:  $\mathbb{Z}$ 

```

```

proc nuevoCacheLRU (in  $cap : \mathbb{Z}$ ) :  $CacheLRU\langle K, V \rangle$  {
  requiere  $\{cap \geq 0\}$ 
  asegura  $\{res.capacidad = cap\}$ 
  asegura  $\{|res.data| = 0 \wedge |res.time| = 0 \wedge |res.access| = 0\}$ 
}

```

```

proc esta (in  $c : Cache\langle K, V \rangle, in\ k : K$ ) :  $Bool$  {
  requiere  $\{True\}$ 
  asegura  $\{res = True \leftrightarrow k \in c.data \wedge k \in c.time\}$ 
}

```

```

proc obtener (in  $c : Cache\langle K, V \rangle, in\ k : K$ ) :  $V$  {
  requiere  $\{k \in c.data \wedge k \in c.time\}$ 
  asegura  $\{res = c.data[k] \wedge setKey(c.access, k, horaActual())\}$ 
}

```

```

proc definir (inout  $c : CacheLRU\langle K, V \rangle, in\ k : K, \in v : V$ ) : {
  requiere  $\{c = C_0\}$ 
  asegura  $\{C_0.cap = 0 \leftrightarrow c.cap = 0 \wedge c.data = \{\} \wedge c.time = \{\} \wedge c.access = \{\}\}$ 
  asegura  $\{C_0.cap = |C_0.data| \wedge k \in C_0.data \leftrightarrow c.data = setKey(c.data, k, v) \wedge$ 
     $c.time = setKey(c.time, k, horaActual()) \wedge (\forall k' : K)(k' \in C_0.access \Leftrightarrow k' \in c.access)\}$ 
  asegura  $\{C_0.cap = |C_0.data| \wedge \neg(k \in C_0.data) \leftrightarrow$ 
     $c.data = setKey(delKey(C_0.data, m), k, v) \wedge$ 
     $c.time = setKey(delKey(C_0.time, m), k, horaActual()) \wedge$ 
     $((oldestAccess(C_0, m) \wedge c.access = delKey(C_0.access, m)) \vee$ 
     $\neg oldestAccess(C_0, m) \wedge claveMasVieja(C_0, m))\}$ 
  asegura  $\{C_0.cap > |C_0.data| \leftrightarrow c.data = setKey(C_0.data, k, v) \wedge$ 
     $c.time = setKey(C_0.time, k, horaActual()) \wedge (\forall k' : K)(k' \in C_0.access \Leftrightarrow k' \in c.access)\}$ 
  asegura  $\{c.cap = C_0.cap\}$ 
}

```

```

pred oldestKey ( $c : CacheLRU\langle K, V \rangle, k : K$ ) {
   $(\forall p : K)(p \in c.time \leftrightarrow c.time[p] \geq c.time[k])$ 
}

```

```

pred oldestAccess ( $c : CacheLRU\langle K, V \rangle, k : K$ ) {
   $(\forall p : K)(p \in c.access \leftrightarrow c.access[p] \geq c.access[k])$ 
}

```

```

iii) TAD CacheTTL $\langle K, V \rangle$ {
  obs data: dict( $K, V$ )
  obs time: dict( $K, \mathbb{Z}$ )
  obs tmax:  $\mathbb{Z}$ 

  proc nuevoCacheTTL (in tiempoMax :  $\mathbb{Z}$ ) : CacheTTL $\langle K, V \rangle$  {
    requiere {cap  $\geq 0$ }
    asegura {res.tmax = tiempoMax}
    asegura {|res.data| = 0  $\wedge$  |res.time| = 0}
  }

  proc esta (in c : CacheTTL $\langle K, V \rangle$ , in k :  $K$ ) : Bool {
    requiere {True}
    asegura {res = True  $\leftrightarrow k \in c.data \wedge k \in c.time \wedge c.time[k] < c.tmax$ }
  }

  proc obtener (in c : CacheTTL $\langle K, V \rangle$ , in k :  $K$ ) :  $V$  {
    requiere { $k \in c.data \wedge k \in c.time \wedge c.time[k] < c.tmax$ }
    asegura {res = c.data[k]}
  }

  proc definir (inout c : CacheLRU $\langle K, V \rangle$ , in k :  $K$ , in v :  $V$ ) : {
    requiere {c =  $C_0$ }
    asegura {c.data = setKey(C0.data, k, v)  $\wedge$  c.time = setKey(C0.time, k, time)}}
    asegura {c.tmax = C0.tmax}
  }
}

```

## Ejercicio 9

```

TAD robot{
  obs pos: struct $\langle x : \mathbb{Z}, y : \mathbb{Z} \rangle$ 
  obs cont(p : struct $\langle x : \mathbb{Z}, y : \mathbb{Z} \rangle$ ):  $\mathbb{Z}$ 

  proc nuevoRobot (in posx, posy :  $\mathbb{Z}$ ) : robot {
    asegura {res.pos.x = posx  $\wedge$  res.pos.y = posy}
    asegura {( $\forall a, b : \mathbb{Z}$ )( $a \neq posx \wedge b \neq posy \leftrightarrow res.cont(\langle x : a, y : b \rangle) = 0$ )}}
    asegura {res.cont( $\langle x : posx, y : posy \rangle$ ) = 1}
  }

  proc arriba (inout r : robot): {
    requiere {r =  $R_0$ }
    asegura {r.pos.x = R0.pos.x  $\wedge$  r.pos.y = R0.pos.y + 1}
    asegura {r.cont( $\langle x : r.pos.x, y : r.pos.y \rangle$ ) = R0.cont( $\langle x : R_0.pos.x, y : R_0.pos.y \rangle$ ) + 1}
  }

  proc abajo (inout r : robot): {
    requiere {r =  $R_0$ }
    asegura {r.pos.x = R0.pos.x  $\wedge$  r.pos.y = R0.pos.y - 1}
    asegura {r.cont( $\langle x : r.pos.x, y : r.pos.y \rangle$ ) = R0.cont( $\langle x : R_0.pos.x, y : R_0.pos.y \rangle$ ) + 1}
  }
}

```

```

proc derecha (inout r : robot): {
  requiere {r = R0}
  asegura {r.pos.x = R0.pos.x + 1 ∧ r.pos.y = R0.pos.y}
  asegura {r.cont(⟨x : r.pos.x, y : r.pos.y⟩) = R0.cont(⟨x : R0.pos.x, y : R0.pos.y⟩) + 1}
}

proc izquierda (inout r : robot): {
  requiere {r = R0}
  asegura {r.pos.x = R0.pos.x - 1 ∧ r.pos.y = R0.pos.y}
  asegura {r.cont(⟨x : r.pos.x, y : r.pos.y⟩) = R0.cont(⟨x : R0.pos.x, y : R0.pos.y⟩) + 1}
}

proc masDerecha (in r : robot): ⟨x : ℤ, y : ℤ⟩ {
  requiere {True}
  asegura {res = c ↔ res.cont(c) > 0 ∧ (∀c' : ⟨x : ℤ, y : ℤ⟩)(res.cont(c') > 0 ↔ c'.x ≤ c.x)}
}

proc cuantasVecesPaso (in r : robot, in t : ⟨x : ℤ, y : ℤ⟩): ℤ {
  requiere {True}
  asegura {res = r.cont(t)}
}

```

## Ejercicio 10

```

TAD vivero{
  obs stock: dict(K, ℤ)
  obs precios: dict(K, ℝ)
  obs balance: ℝ

  proc nuevoVivero (in capital): Vivero {
    asegura {|res.data| = 0 ∧ |res.stock| = 0 ∧ res.balance = capital}
  }

  proc comprar (inout v : vivero, in planta : string, in cant : ℤ, in precio : ℝ): {
    requiere {capital ≥ cant * precio ∧ cant > 0 ∧ precio > 0}
    requiere {v = V0}
    asegura {planta ∈ v.stock ↔ v.stock = setKey(V0.stock, planta, V0.stock[planta] + cant)}
    asegura {planta ∉ v.stock ↔ v.stock = setKey(V0.stock, planta, cant)}
    asegura {v.balance = V0.balance - cant * precio}
    asegura {v.precios = V0.precios}
  }

  proc asignarPrecio (inout v : vivero, in planta : string, in p : precio): {
    requiere {v = V0 ∧ planta ∈ V0.stock ∧ p > 0}
    asegura {v.precios = setKey(V0.stock, planta, p) ∧ v.stock = V0.stock ∧
              v.balance = V0.balance}
  }

  proc venta (inout v : vivero, in planta : string): {

```

```

    requiere  $\{V_0.stock[planta] > 0 \wedge planta \in V_0.precios \wedge v = V_0\}$ 
    asegura  $\{v.stock[planta] = V_0.stock[planta] - 1 \wedge v.precios = V_0.precios\}$ 
    asegura  $\{v.balance = V_0.balance + v.precios[planta]\}$ 
  }

proc balance (in v : vivero) :  $\mathbb{R}$  {
  requiere  $\{True\}$ 
  asegura  $\{res = v.balance\}$ 
}

```

## Ejercicio 11

```

TAD dobleCola(T){
  obs cola1: seq(T)
  obs cola2: seq(T)
  obs lastCola:  $\mathbb{Z}$ 

  proc nuevaDobleCola (in c1, c2 : seq(T)) : dobleCola(T) {
    requiere  $\{noHayRepetidos(c1 ++ c2)\}$ 
    asegura  $\{res.col1 = c1 \wedge res.col2 = c2 \wedge lastCola = 2\}$ 
  }

  proc encolar (inout dc : dobleCola(T) in elem : T, in cola :  $\mathbb{Z}$ ) : {
    requiere  $\{dc = Dc_0 \wedge (cola = 1 \vee cola = 2)\}$ 
    asegura  $\{cola = 1 \leftrightarrow dc.col1 = Dc_0.col1 ++ \{elem\} \wedge dc.col2 = Dc_0.col2\}$ 
    asegura  $\{cola = 2 \leftrightarrow dc.col2 = Dc_0.col2 ++ \{elem\} \wedge dc.col1 = Dc_0.col1\}$ 
    asegura  $\{dc.lastCola = Dc_0.lastCola\}$ 
  }

  proc desencolar (inout dc : dobleCola(T)) : T {
    requiere  $\{dc = Dc_0 \wedge (|Dc_0.col1| > 0 \vee |Dc_0.col2| > 0)\}$ 
    asegura  $\{Dc_0.lastCola = 2 \wedge |Dc_0.col1| > 0 \leftrightarrow$ 
       $dc.col1 = tail(Dc_0.col1) \wedge res = head(Dc_0.col1) \wedge dc.lastCola = 1\}$ 
    asegura  $\{Dc_0.lastCola = 1 \wedge |Dc_0.col2| > 0 \leftrightarrow$ 
       $dc.col2 = tail(Dc_0.col2) \wedge res = head(Dc_0.col2) \wedge dc.lastCola = 2\}$ 
    asegura  $\{Dc_0.lastCola = 2 \wedge |Dc_0.col1| = 0 \wedge |Dc_0.col2| > 0 \leftrightarrow$ 
       $dc.col2 = tail(Dc_0.col2) \wedge res = head(Dc_0.col2) \wedge dc.lastCola = 1\}$ 
    asegura  $\{Dc_0.lastCola = 1 \wedge |Dc_0.col2| = 0 \wedge |Dc_0.col1| > 0 \leftrightarrow$ 
       $dc.col1 = tail(Dc_0.col1) \wedge res = head(Dc_0.col1) \wedge dc.lastCola = 2\}$ 
  }

  proc mudarElemento (inout dc : dobleCola(T), in elem : T) : {
    requiere  $\{elem \in (Dc_0.col1 ++ Dc_0.col2) \wedge dc = Dc_0\}$ 
    asegura  $\{esIndice(i, elem, Dc_0.col1) \leftrightarrow$ 
       $dc.col1 = subseq(Dc_0.col1, 0, i) ++ subseq(Dc_0.col1, i + 1, |Dc_0.col1|) \wedge$ 
       $dc.col2 = Dc_0.col2 ++ \{elem\}\}$ 
    asegura  $\{esIndice(i, elem, Dc_0.col2) \leftrightarrow$ 
       $dc.col2 = subseq(Dc_0.col1, 0, i) ++ subseq(Dc_0.col2, i + 1, |Dc_0.col2|) \wedge$ 
       $dc.col1 = Dc_0.col1 ++ \{elem\}\}$ 
  }
}

```

```

}

pred noHayRepetidos (s : seq⟨T⟩) {
  (∀i, j : T)(i ∈ s ∧ j ∈ s → i ≠ j)}
pred esIndice (i : ℤ, elem : T, s : seq⟨T⟩) {
  res = True ↔ 0 ≤ i < |s| ∧ s[i] = elem}

```

## Ejercicio 12

TAD *CallOfDuty : The Arrival of AED*{

obs *batallones*: dict(*K*, seq⟨ℤ⟩)

obs *money*: ℝ

obs *lands*: seq⟨T⟩

Nota: por comodidad se usara la equivalencia cod=CallOfDuty:The Arrival of AED

```

proc CoDInit (in m : ℝ, in land : K, in ) : cod {
  requiere {m ≥ 0}
  asegura {res.batallones = {} ∧ res.money = m ∧ res.lands = land}
}

proc contratarMercenario (inout c : cod, in bat : K, in mer : struct⟨power : ℤ, prize : ℝ⟩) :
{
  requiere {c = C0 ∧ C0.money ≥ mer.prize}
  requiere {mer.prize ≥ 0 ∧ mer.power ≥ 0}
  asegura {|C0.batallones[bat]| > 0 ↔
    c.batallones = setKey(C0.batallones, bat, C0.batallones[bat] + {mer.power})}
  asegura {|C0.batallones[bat]| = 0 ↔ c.batallones = setKey(C0.batallones, bat, {mer.power})}
  asegura {(∀k : K)(k ∈ C0.batallones ∧ k ≠ bat ↔ c.batallones[k] = C0.batallones[k])}
  asegura {c.money = C0.money - mer.prize}
  asegura {c.lands = C0.lands}
}

proc atacarTerritorio (inout c : cod, in bat : k, in terr : T, in power : ℤ) : {
  requiere {c = C0}
  requiere {terr ∉ C0.lands ∧ power ≥ 0 ∧ C0.money ≥ 500}
  asegura {poderBat(C0, bat) > power ↔ c.money = C0 - money + 1000 ∧
    c.lands = C0.lands ++ {terr}}
  asegura {poderBat(C0, bat) ≤ power ↔ c.money = C0 - money - 500 ∧ c.lands = C0.lands}
  asegura {c.batallones = C0.batallones}
}

proc cantTerritorios (in c : cod) : ℤ {
  asegura {res = |c.lands|}
}

aux poderBat (in c : cod, in bat : K) : ℤ = ∑i=0|c.batallones[bat]|-1 c.batallones[bat][i]

```