Contents

1) (OK)

```
implement Lista of T where

    type Node of T = tuple
      elem: T
      next: pointer to (Node of T)
    end tuple
    type Lista of T = pointer to Node of T

    proc concat(in/out l: List of T, in l0: List of T )
      var p: pointer to Node of T
      p:= l
      while (p->next ≠ Null) do
        p := p->next
      od
      p->next = l0
    end proc
```

```
fun index(l: List of T, n: nat) ret e : T
  var p: pointer to Node of T
  p:= l
  for i := 1 to n do
    p:= p->next
  od
  e:= p->elem
end fun
proc take(in/out l: List of T, in n: nat)
  var p: pointer to Node of T
  p:= l
  for i := 1 to n do
    p:= p->next
  od
  p->next = null
end proc
proc drop(in/out l: List of T, in n: nat)
  var p: pointer to Node of T
  p:= l
  for i := 1 to n do
    p:= p->next
  od
  l:= p->next
end proc
fun copy_list(l1: List of T) ret l2 : List of T
  var p: pointer to Node of T
  p:= l
  while (p->next ≠ Null) do
    l2->elem:= p->elem
    l2->next:= p->next
    p := p->next
  od
  p->next = l0
end proc
```

2) (OK?)

implement Lista **of** T **where**

```
type Lista of T = tuple
  elems: array[1..N] of T
  used: Nat
end tuple
```

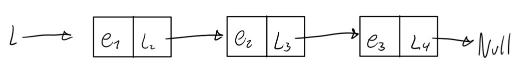
```
fun empty() ret l : List of T
  var a: array[1..N] of T
  l->elems:= a
  l->used := 0
end fun
proc addl(in e: T, in/out l: List of T)
  for i := 2 to l->used do
    l->elems[i-1] = l->elems[i]
  od
  l->elems[1] = e
end proc
fun is_empty(l: List of T) ret b : bool
  b:= (l->used = 0)
end fun
fun head(l: List of T) ret e : T
  e:= l->elems[1]
end fun
proc tail(in/out l: List of T)
  for i := 2 to l->used do
    l->elems[i-1] = l->elems[i]
  od
end proc
proc addr(in/out l: List of T, in e: T)
  l->elems[l->used+1]:= e
end proc
fun length(l: List of T) ret n : Nat
  n:= l->used
end fun
proc concat(in/out l: List of T, in l0: List of T )
  var largo: Nat
  largo:= length(l)
  for i := 1 to length(l0) do
    l->elems[i+largo] = l0->elems[i]
  od
end proc
fun index(l: List of T, n: nat) ret e : T
  e:= l->elems[n]
end fun
proc take(in/out l: List of T, in n: nat)
  l->used:= n
end proc
proc drop(in/out l: List of T, in n: nat)
  for i := 1 to n do
    l->elems[i] = l->elems[i+n]
  od
  l->elems[1] = e
end proc
```

```
fun copy_list(l1: List of T) ret l2 : List of T
   var p: pointer to Node of T
   p:= l
   while (p->next ≠ Null) do
      l2->elem:= p->elem
      l2->next:= p->next
      p := p->next
   od
   p->next = l0
end proc
```

Esta implementacion impone la restriccion de que el tamaño de la lista no será mutable, es decir, solo puede tener un numero N de elementos.

3) (OK)

```
proc add_at(in/out l: Lista of T, n: nat, e: T)
   var end: List of T
   end:= copy_list(l)
   take(l, n)
   drop(end, n)
   addr(l, e)
   concat(l, end)
end proc
```



4)

a) (OK)

spec Tablero **where**

```
constructors
  {- start of match -}
  fun start() ret r : Tablero
  {- new point A -}
  proc newPointA(in/out t: Tablero)
  {- new point B -}
  proc newPointB(in/out t: Tablero)
operations
  {- isCounter0 -}
  fun isCounter0(t: Tablero) ret r : bool
  {- hasAScored -}
  fun hasAScored(t: Tablero) ret r : bool
  {- hasBScored -}
  fun hasBScored(t: Tablero) ret r : bool
  {- AisWinning -}
  fun isAWinning(t: Tablero) ret r : bool
  {- BisWinning -}
  fun isBWinning(t: Tablero) ret r : bool
  {- isTie -}
  fun isTie(t: Tablero) ret r : bool
  {- giveApoints -}
  proc giveAPoints(t: Tablero)
  {- giveBpoints -}
  proc giveBPoints(t: Tablero)
  {- substractApoints -}
  proc substractAPoints(t: Tablero)
  {- substractBpoints -}
  proc substractBPoints(t: Tablero)
```

b) (OK)

```
implement Tablero of T where
type Tablero of T  = tuple
  A: Counter of T
  B: Counter of T
end tuple
```

```
{- start of match -}
fun start() ret r : Tablero
   r->A:= init()
   r->B:= init()
end fun
{- new point A -}
proc newPointA(in/out t: Tablero)
   incr(t->A)
end proc
{- new point B -}
proc newPointB(in/out t: Tablero)
   incr(t->B)
end proc
{- isCounter0 -}
fun isCounter0(t: Tablero) ret r : bool
   r:= is_init(t->A) ∧ is_init(t->B)
end fun
{- hasAScored -}
fun hasAScored(t: Tablero) ret r : bool
   r:= ¬ is_init(t->A)
end fun
{- hasBScored -}
fun hasBScored(t: Tablero) ret r : bool
   r:= ¬ is_init(t->B)
end fun
{- AisWinning -}
fun isAWinning(t: Tablero) ret r : bool
   r:= t->A > t->B
end fun
{- BisWinning -}
fun isBWinning(t: Tablero) ret r : bool
   r:= t->B > t->A
end fun
{- isTie -}
fun isTie(t: Tablero) ret r : bool
   r:= t->B = t->A
end fun
```

```
{- giveApoints -}
proc giveAPoints(t: Tablero, n: Nat )
   for i := 1 to n do
      incr(t->A)
   od
end proc
{- giveBpoints -}
proc giveBPoints(t: Tablero, n: Nat )
   for i := 1 to n do
      incr(t->B)
   od
end proc
{- substractApoints -}
proc giveAPoints(t: Tablero, n: Nat )
   var i: Nat
   i:= 0
   while (¬ is_init(t->A) ∧i < n) do
      decr(t->A)
      i = i + 1
   od
end proc
{- substractBpoints -}
proc giveBPoints(t: Tablero, n: Nat )
   var i: Nat
   i:= 0
   while (¬ is_init(t->B) ∧i < n) do
      decr(t->B)
      i = i + 1
   od
end proc
```

c) (OK)

```
implement Tablero of T where
type Tablero of T   = tuple
   A: nat
   B: nat
end tuple
```

```
{- start of match -}
fun start() ret r : Tablero
  r->A:= 0
  r->B:= 0
end fun
{- new point A -}
proc newPointA(in/out t: Tablero)
  t->A = t->A + 1
end proc
{- new point B -}
proc newPointB(in/out t: Tablero)
  t->B = t->B + 1
end proc
{- isCounter0 -}
fun isCounter0(t: Tablero) ret r : bool
  r:= (t->A = 0) ∧ (t->A = 0)
end fun
{- hasAScored -}
fun hasAScored(t: Tablero) ret r : bool
  r:= t->A > 0
end fun
{- hasBScored -}
fun hasBScored(t: Tablero) ret r : bool
  r:= t->B > 0
end fun
{- AisWinning -}
fun isAWinning(t: Tablero) ret r : bool
  r:= t->A > t->B
end fun
{- BisWinning -}
fun isBWinning(t: Tablero) ret r : bool
  r:= t->B > t->A
end fun
{- isTie -}
fun isTie(t: Tablero) ret r : bool
  r:= (t->B = t->A)
end fun
```

```
{- giveApoints -}
proc giveAPoints(t: Tablero, n: Nat )
  t->A = t->A + n
end proc
{- giveBpoints -}
proc giveBPoints(t: Tablero, n: Nat )
  t->B = t->B + n
end proc
{- substractApoints -}
proc substractAPoints(t: Tablero, n: Nat )
  if (t->A - n) > 0 then
    t->A = n
  else
    t->A = 0
  fi
end proc
{- substractBpoints -}
proc substractBPoints(t: Tablero, n: Nat )
  if (t->B - n) > 0 then
    t->B = n
  else
    t->B = 0
  fi
end proc
```

Las operaciones que usan el tipo del Contador quizas sean un poco mas sencillas de entender ya que se aplica un nuevo nivel de abstraccion

5) (OK)

```
spec Conjunto of T where
constructors
  {- Conjunto vacio -}
  fun empty_set() ret r : Conjunto
  {- Agregar elemento -}
  proc addc(in/out c: Conjunto)
operations
  {- isInSet -}
  fun isInSet(e: T, c: Conjunto) ret r : bool
  {- isEmpty -}
  fun isEmpty(c: Conjunto) ret r : bool
  {- Union -}
  proc union(in/out c1: Conjunto, in c2: Conjunto)
  {- Intersect -}
  proc intersection(in/out c1: Conjunto, in c2: Conjunto)
  {- Difference -}
  proc difference(in/out c1: Conjunto, in c2: Conjunto)
```

6) (INCOMPLETE)

a) (OK but INC)

implement Conjunto **of** T **where type** Conjunto **of** T = Lista **of** T

```
proc addc(in/out c: Conjunto, e: T)
  addl(c, e)
end proc
```

b) (INCOMPLETE)

implement Conjunto **of** T **where type** Conjunto **of** T = Lista **of** T

```
{- Conjunto vacio -}
fun empty_set() ret r : Conjunto
  r:= empty()
end fun
{- Agregar elemento -}
proc addc(in/out c: Conjunto, e: T )
  var i: Nat
  i:= 0
  if ¬ isEmpty(c)then
    while (index(c, i) < e) do
      i = i+1
    od
    add_at(c, i, e)
  else
    addl(c, e)
  fi
end proc
```

```
{- isInSet -}
fun isInSet(e: T, c: Conjunto) ret r : bool
  var c1: Conjunto of T
  var head: T
  c1 := copy_list(c)
  head:= head(c1)
  tail(c1)
  while (¬ is_empty(c1) ∧ head != e) do
    head:= head(c1)
    tail(c1)
  od
  if head = e then
    r:= true
  else
    r:= false
  fi
end fun
{- isEmpty -}
fun isEmpty(c: Conjunto) ret r : bool
  r:= is_empty(c)
end fun
{- Union -}
proc union(in/out c1: Conjunto, in c2: Conjunto)
  var c3: Conjunto of T
  var head: T
  c3:= copy_list(c2)
  while (¬ is_empty_set(c3)) do
    head:= head(c3)
    tail(c3)
    addc(c1, head)
  od
end proc
{- Intersect -}
proc intersection(in/out c1: Conjunto, in c2: Conjunto)
  var c3, c4: Conjunto of T
  var head1, head2: T
  c3:= copy_list(c2)
  while (¬ is_empty_set(c3)) do
    head:= head(c3)
    tail(c3)
    if isInSet(c1, head) then
      addc(c4, head)
    fi
  od
  c1:= c4
end proc
```

```
{- Difference -}
proc difference(in/out c1: Conjunto, in c2: Conjunto)
   var c3, c4: Conjunto of T
   var head1, head2: T
   c3:= copy_list(c1)
   while (¬ is_empty_set(c3)) do
      head:= head(c3)
      tail(c3)
      if ¬ isInSet(c2, head) then
         addc(c4, head)
      fi
   od
   c1:= c4
end proc
```