Práctico 2.2

1)

1. Completá la implementación de listas dada en el teórico usando punteros.

```
type List of T = pointer to (Node of T)

type Node of T = tuple
                     elem : T
                     next : List of T
                 end tuple

fun empty ret l : List of T
      l ≔ NULL
end fun


proc addl(in e : T, in/out l : List of T)
      var p : pointer to (Node of T)
      aloc(p)
      p→elem ≔ e
      p→next ≔ l
      l ≔ p
end proc


proc destry(in/out l : List of T)
      if ¬isEmpty(l) →
            destroy(l→next)
            free(l)
      fi
end proc



fun isEmpty(in l : List of T) ret res : bool
      res ≔ l = empty
end fun

fun head(l : List of T) ret res : T
      res ≔ l→elem
end fun

proc tail(in/out l : List of T)
      var temp : List of T
      temp ≔ l
      l ≔ l→next
      free(temp)
end proc
```

```
proc concat(in/out l : List of T, in m : List of T)
      if empty(l) →
            l ≔ m
      else
            var temp : T
            temp ≔ head(l)
            tail(l)
            concat(l, m)
            addl(temp, l)
      fi
end proc

proc addr(in/out l : List of T, in e : T)
      var temp : List of T
      alloc(temp)
      temp→elem ≔ e
      temp→next ≔ empty
      concat(l, temp)
      destroy(temp)
end proc

fun length(l : List of T) ret n : nat
      if isEmpty(l) →
            n ≔ 0
      else
            n ≔ 1 + length(l→next)
      fi
end fun

fun length(l : List of T) ret n : nat
      n ≔ 0
      var s : List of T
      s ≔ l
      while ¬isEmpty(s) do
            s ≔ s→next
            n ≔ n+1
      od
end fun

fun index(l : List of T, n : nat) ret e : T
      if n = 0 →
            e ≔ head(l)
      else
            e ≔ index(l→next, n-1)
      fi
end fun

proc take(in/out l : List of T, in n : nat)
      if n = 0 →
```

```
            destroy(l)
      else if n > 0 ∧ ¬isEmpty(l) →
            var temp : T
            temp ≔ head(l)
            tail(l)
            take(l, n-1)
            addl(temp, l)
      fi fi
end proc

proc drop(in/out l : List of T, in n : nat)
      in n > 0 →
            tail(l)
            drop(l, n-1)
      fi
end proc

fun copy_list(l1 : List of T) ret l2 : List of T
      if isEmpty(l1) →
            l2 = empty
      else
            alloc(l2)
            l2→elem ≔ l1→elem
            l2→next ≔ copy_list(l1→next)
      fi
end fun
```

2)

2. Dada una constante natural N, implementá el TAD Lista de elementos de tipo T, usando un arreglo de tamaño N y un natural que indica cuántos elementos del arreglo son ocupados por elementos de la lista. ¿Esta implementación impone nuevas restricciones? ¿En qué función o procedimiento tenemos una nueva precondición?

```
type List of T = tuple
                     listToArray : array[1..N] of T
                     tam : nat
                 end tuple
```

Constructores:

```
fun empty ret l : List of T
      l.tam ≔ 0
end fun

proc addl(in e : T, in/out l : List of T)
      l.listToArray[N - l.tam] ≔ e
      l.tam ≔ l.tam + 1
end proc
```

{- La lista se construye desde la izquierda -}

Destructor:

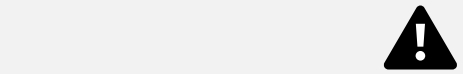
```
proc destroy(in/out l : List of T)
      l.tam ≔ 0
end proc
```

Operadores:

```
fun is_empty(l : List of T) ret b : bool
      b ≔ l.tam = 0

fun head(l : List of T) ret e : T
      e ≔ l.listToArray[N - l.tam + 1]

proc tail(in/out l : List of T)
      l.tam ≔ l.tam - 1

fun length(l : List of T) ret n : nat
      n ≔ l.tam

{- PRE: l.tam + l0.tam ≤ N -}
proc concat(in/out l : List of T, in l0 : List of T)
      for j ≔ N - l.tam + 1 to N do
            l.listToArray[j - l0.tam] ≔ l.listToArray[j]
      od
      for j ≔ N - l0.tam + 1 to N do
            l.listToArray[j] ≔ l0.lisToArray[j]
      od
end proc

proc addr(in/out l : List of T, in e : T)
      var a : list of T
      a.listToArray[N] ≔ e
      a.tam ≔ 1
      concat(l, a)
end proc

fun index(l : List of T, n : nat) ret e : T
      e ≔ l.listToArray[N - n]

proc take(in/out l : List of T, in n : nat)
      l.tam ≔ n `min` l.tam

proc drop(in/out l : List of T, in n : nat)
      for j ≔ N - l.tam to N - n do
            l.listToArray[j - n] ≔ l.listToArray[j]
      od
      l.tam ≔ l.tam - n
```

```
end proc

proc drop(in/out l : List of T, in n : nat)
      l.tam := l.tam - n

fun copy_list(l1 : List of T) ret l2 : List of T
      l2.listToArray := l1.listToArray
      l2.tam := l1.tam
```

3)



```
proc add_att(in/out l : list of T, in e : T, n : nat)
      var l1 : list of T
      copy_list(l, l1)
      take(l, n)
      drop(l1, n)
      addl(l1, e)
      concat(l, l1)
      destroy(l1)
end proc
```

4) 🛠️

4. (a) Especificá un TAD *tablero* para mantener el tanteador en contiendas deportivas entre dos equipos (equipo A y equipo B). Deberá tener un constructor para el comienzo del partido (tanteador inicial), un constructor para registrar un nuevo tanto del equipo A y uno para registrar un nuevo tanto del equipo B. El tablero sólo registra el estado actual del tanteador, por lo tanto el orden en que se fueron anotando los tantos es irrelevante.

   Además se requiere operaciones para comprobar si el tanteador está en cero, si el equipo A ha anotado algún tanto, si el equipo B ha anotado algún tanto, una que devuelva verdadero si y sólo si el equipo A va ganando, otra que devuelva verdadero si y sólo si el equipo B va ganando, y una que devuelva verdadero si y sólo si se registra un empate.

   Finalmente habrá una operación que permita anotarle un número n de tantos a un equipo y otra que permita "castigarlo" restándole un número n de tantos. En este último caso, si se le restan más tantos de los acumulados equivaldrá a no haber anotado ninguno desde el comienzo del partido.

   (b) Implementá el TAD Tablero utilizando una tupla con dos contadores: uno que indique los tantos del equipo A, y otro que indique los tantos del equipo B.

   (c) Implementá el TAD Tablero utilizando una tupla con dos naturales: uno que indique los tantos del equipo A, y otro que indique los tantos del equipo B. ¿Hay alguna diferencia con la implementación del inciso anterior? ¿Alguna operación puede resolverse más eficientemente?

```
spec Tablero where

constructores

fun TableroInicial() ret res : Tablero
{- El tablero vacío -}

fun más1A(t : Tablero) ret u : Tablero
{- Agregar 1 punto al equipo A -}

fun más1B(t : Tablero) ret u : Tablero
{- Agregar 1 punto al equipo B -}




fun AGana(t : Tablero) ret res : Bool
{- Dice si A va ganando -}



fun agregar_n_a_A(n : nat, t : Tablero) ret res : Tablero
{- Agregar n puntos al equipo A -}

fun restar_n_a_A(n : nat, t : Tablero) ret res : Tablero
{- Quita n puntos al equipo A, si A tiene menos de n puntos, lo deja sin puntos -}
```

5)

5. Especificá el TAD Conjunto finito de elementos de tipo T. Como constructores considerá el conjunto vacío y el que agrega un elemento a un conjunto. Como operaciones: una que chequee si un elemento e *pertenece* a un conjunto c, una que chequee si un conjunto *es vacío*, la operación de *unir* un conjunto a otro , *insersectar* un conjunto con otro y obtener la *diferencia*. Estas últimas tres operaciones deberían especificarse como procedimientos que toman dos conjuntos y modifican el primero de ellos.

```
spec set of T
```

Constructores:

```
fun emptySet ret s : set of T
{- Crea un conjunto vacío -}

proc addToSet(in/out s : set of T, in a : T)
```

{- Agrega elemento a un conjunto, si este ya está en el conjunto, no hace nada (se asume que `T` tiene definida la función `(=))` -}

Destructor:

```
proc destroy(in/out s : set of T)
```
{- Destruye un conjunto, liberando memoria -}

Funciones y procedimientos:

```
fun (∈)(in a : T, s : set of T) ret res : bool
```
{- Chequea si un elemento se encuentra en un conjunto (se asume que `T` tiene definida la función `(=))` -}

```
fun isEmptySet(in s : set of T) ret res : bool
```
{- Chequea si un conjunto es vacío -}

```
proc unionSet(in/out s : set of T, in t : set of T)
```
{- Agrega todos los elementos de `t` a `s` (se asume que `T` tiene definida la función `(=))` -}

```
proc intersecSet(in/out s : set of T, in t : set of T)
```
{- Elimina todos los elementos de `s` que no se encuentren en `t` (se asume que `T` tiene definida la función `(=))` -}

```
proc diferenciaSet(in/out s : set of T, in t : set of T)
```
{- Elimina todos los elementos de `s` que se encuentren en `t` (se asume que `T` tiene definida la función `(=))` -}

6)

6. Implementá el TAD Conjunto finito de elementos de tipo T utilizando:

   (a) una lista de elementos de tipo T, donde el constructor para agregar elementos al conjunto se implementa directamente con el constructor **addl** de las listas.

   (b) una lista de elementos de tipo T, donde se asegure siempre que la lista está ordenada crecientemente y no tiene elementos repetidos. Debes tener cuidado especialmente con el constructor de agregar elemento y las operaciones de unión, intersección y diferencia. A la propiedad de mantener siempre la lista ordenada y sin repeticiones le llamamos *invariante de representación*. Ayuda: Para implementar el constructor de agregar elemento puede serte muy útil la operación *add_at* implementada en el punto 3.

6a)

Implementación:

```
type set of T = List of T
```

Constructores:

```
fun emptySet ret s : set of T
      s ≔ emptyList

proc addToSet(in/out s : set of T, in a : T)
      addl(s, a)
```

Destructor:

```
proc destroy(in/out s : set of T)
      destroy_list(s)
```

Operaciones:

```
fun (∈)(in a : T, s : set of T) ret res : bool
      res ≔ false
      for j ≔ 0 to length(s) - 1 do
            res ≔ res ∨ index(s, j) = a
      od
end fun

fun isEmptySet(in s : set of T) ret res : bool
      res ≔ isEmptyList(s)
end fun

proc unionSet(in/out s : set of T, in t : set of T)
      concat(a, t)
end proc

proc removeIndex((in/out s : List of T, in n : nat)
      var temp : List of T
      temp ≔ copy_list(s)
      take(s, n)
      drop(temp, n + 1)
      concat(s, temp)
      destroy(temp)
end proc

proc intersecSet(in/out s : set of T, in t : set of T)
      var j : nat
      j ≔ 0
      while j < length(s) do
            if index(s, j) ∈ t →
                  j ≔ j + 1
            else
                  removeIndex(s, j)
            fi
      od
end proc

proc diferenciaSet(in/out s : set of T, in t : set of T)
```

```
      var j : nat
      j ≔ 0
      while j < length(s) do
            if index(s, j) ∈ t →
                  removeIndex(s, j)
            else
                  j ≔ j + 1
            fi
      od
end proc
```

6b)

Implementación:

```
type set of T = List of T
```

Constructores:

```
fun emptySet ret s : set of T
      s ≔ emptyList

proc addToSet(in/out s : set of T, in a : T)
      var j, length : nat
      length ≔ length(s)
      j ≔ 0
      while j < length ∧ index(s, j) < a do
            j ≔ j+1
      od
      if j = length ∨ index(s, j) ≠ a →
            add_att(s, j+1)
      fi
end proc
```

Funciones y procedimientos:

```
fun (∈)(in a : T, s : set of T) ret res : bool
      res ≔ false
      for j ≔ 0 to length(a) - 1 do
            res ≔ res ∨ index(s, j) = a
      od

fun isEmptySet(in s : set of T) ret res : bool
      res ≔ isEmptyList(s)
end fun

proc unionSet(in/out s : set of T, in t : set of T)
      for j ≔ 0 to length(t) do
            addToSet(s, index(t, j))
```

```
      od
end proc

proc intersecSet(in/out s : set of T, in t : set of T)
      var j : nat
      j ≔ 0
      while j < length(s) do
            if index(s, j) ∈ t →
                  j ≔ j + 1
            else
                  removeIndex(s, j)
            fi
      od
end proc

proc diferenciaSet(in/out s : set of T, in t : set of T)
      var j : nat
      j ≔ 0
      while j < length(s) do
            if index(s, j) ∈ t →
                  removeIndex(s, j)
            else
                  j ≔ j + 1
            fi
      od
end proc
```