1

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

```
implement List of T where

type Node of T =    tuple
                        elem : T

                        next : pointer to (Node of T)
                    end tuple

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

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)
    alloc(p)

    p->elem := e
    p->next := l

    l := p
end proc

proc destroy(in/out l : List of T)
    var p : pointer to (Node of T)

    while (l != null) do
        p := l

        while (p->next != null) do
            p := p->next
        od

        free(p)
    od

end proc

fun is_empty(l : List of T) ret b : bool
    b := l = null
end fun

fun head(l : List of T) ret e : T
    e := l->elem
end fun

proc tail(in/out l : List of T)
```

```
    var p : pointer to (Node of T)
    p := l
    l := l->next

    free(p)
end proc

proc addr (in/out l : List of T,in e : T)
    var p, q : pointer to (Node of T)
    alloc(p)

    p->elem := e
    p->next := null
    q := l

    if (not is_empty(q)) then
        while (q->next != null) do
            q := q->next
        od

        q->next := p
    else
        q := p
    fi
end proc

fun length(l : List of T) ret n : nat
    var p : pointer to (Node of T)

    p := l
    n := 0

    while (not is_empty(p)) do
        p := p->next
        n := n + 1
    od
end fun

proc concat(in/out l : List of T,in l0 : List of T)
    var p : pointer to (Node of T)

    p := l

    if (not is_empty(l)) then
        while (p->next != null) do
            p := p->next
        od

        p->next := l0
    else
        l := l0
    fi
end proc

fun index(l : List of T,n : nat) ret e : T
```

```
    var p : pointer to (Node of T)

    p := l

    while (n > 0) do
        p := p->next
        n := n - 1
    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

    while (n > 0) do
        p := p->next
        n := n - 1
    od

    destroy(p)
end proc

proc drop(in/out l : List of T,in n : nat)
    var p, q : pointer to (Node of T)

    p := l

    while (n > 1) do
        l := l->next
        n := n - 1
    od

    q := l
    l := l->next
    q->next := null

    destroy(p)
end proc

fun copy_list(l1 : List of T) ret l2 : List of T
    var p, q : pointer to (Node of T)

    p := l
    q := null
    l2 := q

    while (not is_empty(p)) do
        alloc(q)
        q->elem := p->elem
        q->next := null
        q := q->next
        p := p->next
```

```
    od
end fun
```

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?

```
implement List of T where

type List of T =    tuple
                        list : array[1..N] of pointer to T
                        length : nat
                    end tuple

fun empty() ret l : List of T
    l.list := null
    l.length := 0
end fun

// length(l) < N
proc addl(in e : T, in/out l : List of T)
    var p : pointer to T
    var i : nat

    alloc(p)

    *p := e
    i := l.length

    while (i > 0) do
        l.list[i] := l.list[i + 1]
        i := i - 1
    od

    l.length := l.length + 1
    l.list[1] := p
end proc

proc destroy(in/out l : List of T)
    while (l.length > 0) do
        free(l.list[l.length])
        l.length := l.length - 1
    od

end proc

fun is_empty(l : List of T) ret b : bool
    b := l.length = 0
end fun

fun head(l : List of T) ret e : T
```

```
    e := l.list[1]
end fun

proc tail(in/out l : List of T)
    free(l.list[1])
    l.length := l.length - 1

    for i := 1 to l.length do
        l.list[i] := l.list[i + 1]
    od

end proc

// length(l) < N
proc addr (in/out l : List of T,in e : T)
    var p : pointer to T
    alloc(p)

    *p := e
    l.length := l.length + 1
    l.list[l.length] := p
end proc

fun length(l : List of T) ret n : nat
    n := l.length
end fun

// length(l) + length(0) <= N
proc concat(in/out l : List of T,in l0 : List of T)
    for i := 1 to l0.length do
        l.list[i + l.length] := l0.list[i]
    od

    l.length := l.length + l0.length

end proc

fun index(l : List of T,n : nat) ret e : T
    e := l.list[n]
end fun

proc take(in/out l : List of T, in n : nat)
    for i := n + 1 to l.length do
        free(l.list[i])
    od

    l.length := n

end proc

proc drop(in/out l : List of T,in n : nat)
    for i := 1 to n do
        free(l.list[i])
    od
```

```
    for i := n to l.length do
        l.list[i - n] := l.list[i]
        free(l.list[i])
    od

    l.length := l.length - n

end proc

fun copy_list(l1 : List of T) ret l2 : List of T
    var p : pointer to T

    l2.length := l1.length

    for i := 1 to l1.length do
        alloc(p)
        l2.list[i] := p
        *p := l1.list[i]
    od
end fun
```

3

Implementá el procedimiento `add_at` que toma una lista de tipo T, un natural n, un elemento e de tipo T, y agrega el elemento e en la posición n, quedando todos los elementos siguientes a continuación.
Esta operación tiene como precondición que n sea menor al largo de la lista.

```
implement List of T where

proc add_at(in/out l : List of T, in n : nat, e : T)
    var p, q : pointer to (Node of T)
    alloc(p)

    p->elem := e

    if (n > 1) then
        q := l

        for i := 1 to n - 2 do
            q := q->next
        od

        p->next := q->next
        q->next := p
    else
        p->next := l
        l := p
    fi

end proc
```

4

a

Especificá un TAD tablero para mantener el tanteador en contiendas deportivas entre dos equipos (equipo A y equipo B).
Debería 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.

```
spec Tablero where

constructors
    fun gameStart() ret l : Tablero
    {- tanteador inicial. -}

    proc pointA (in/out l : Tablero)
    {- nuevo tanto del equipo A. -}

    proc pointB (in/out l : Tablero)
    {- nuevo tanto del equipo B. -}

destroy
    proc destroy (in/out l : Tablero)
    {- Libera memoria en caso que sea necesario. -}

operations
    fun noPoints(l : Tablero) ret b: bool
    {- comprueba si el tanteador está en cero. -}

    fun teamAhasPoints(l : Tablero) ret b: bool
    {- comprueba si el equipo A ha anotado algún tanto. -}

    fun teamBhasPoints(l : Tablero) ret b: bool
    {- comprueba si el equipo B ha anotado algún tanto. -}

    fun teamAIsWinning(l : Tablero) ret b: bool
    {- comprueba si el equipo A va ganando. -}
```

```
    fun teamBIsWinning(l : Tablero) ret b: bool
    {- comprueba si el equipo B va ganando. -}

    fun isDraw(l : Tablero) ret b: bool
    {- comprueba si hay empate. -}

    proc addNPointToTeam(in team: char, n: nat, in/out l : Tablero)
    {- añade n puntos a team. -}
    {- PRE: team = 'A' o team = 'B'. -}

    proc penalizeTeam(in team: str, n: nat, in/out l : Tablero)
    {- quita n puntos a team. -}
```

b

```
implement Tablero where

type List of T =    tuple
                        A : Contador of nat
                        B : Contador of nat
                    end tuple

fun gameStart() ret l : Tablero
    l.A := init()
    l.B := init()
end fun

proc pointA (in/out l : Tablero)
    incr(l.A)
end proc

proc pointB (in/out l : Tablero)
    incr(l.B)
end proc

fun noPoints(l : Tablero) ret b: bool
    b := is_init(l.A) && is_init(l.B)
end fun

fun teamAhasPoints(l : Tablero) ret b: bool
    b := not is_init(l.A)
end fun

fun teamBhasPoints(l : Tablero) ret b: bool
    b := not is_init(l.B)
end fun

fun teamAIsWinning(l : Tablero) ret b: bool
    b := l.A > l.B
end fun

fun teamBIsWinning(l : Tablero) ret b: bool
    b := l.B > l.A
end fun
```

```
fun isDraw(l : Tablero) ret b: bool
    b := l.B = l.A
end fun

proc addNPointToTeam(in team: char, n: nat, in/out l : Tablero)
    if (team = 'A') then
        for i := 1 to n do
            pointA(l)
        od
    else
        for i := 1 to n do
            pointB(l)
        od
    fi

end proc

proc penalizeTeam(in team: str, n: nat, in/out l : Tablero)
    var i: nat
    i := 0

    if (team = 'A') then
        while (i < n && not is_init(l.A)) do
            dec(l.A)
        od
    else
        while (i < n && is_init(l.B)) do
            dec(l.B)
        od
    fi
end proc
```

C

```
implement Tablero where

type List of T =    tuple
                        A : nat
                        B : nat
                    end tuple

fun gameStart() ret l : Tablero
    l.A := 0
    l.B := 0
end fun

proc pointA (in/out l : Tablero)
    l.A := l.A + 1
end proc

proc pointB (in/out l : Tablero)
    l.B := l.B + 1
end proc
```

```
fun noPoints(l : Tablero) ret b: bool
    b := (l.A + l.B) = 0
end fun

fun teamAhasPoints(l : Tablero) ret b: bool
    b := l.A > 0
end fun

fun teamBhasPoints(l : Tablero) ret b: bool
    b := l.B > 0
end fun

fun teamAIsWinning(l : Tablero) ret b: bool
    b := l.A > l.B
end fun

fun teamBIsWinning(l : Tablero) ret b: bool
    b := l.B > l.A
end fun

fun isDraw(l : Tablero) ret b: bool
    b := l.B = l.A
end fun

proc addNPointToTeam(in team: char, n: nat, in/out l : Tablero)
    if (team = 'A') then
        l.A := l.A + n
    else
        l.B := l.B + n
    fi

end proc

proc penalizeTeam(in team: str, n: nat, in/out l : Tablero)
    if (team = 'A') then
        if (l.A > n) then
            l.A := l.A - n
        else
            l.A := 0
        fi
    else
        if (l.B > n) then
            l.B := l.B - n
        else
            l.B := 0
        fi
    fi
end proc
```

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 ConjuntoFinito of T where

constructors
    fun empty() ret c : ConjuntoFinito of T
    {- conjunto vacío . -}

    proc add(in/out c : ConjuntoFinito of T)
    {- agrega un elemento a un conjunto. -}

destroy
    proc destroy (in/out c : ConjuntoFinito of T)
    {- Libera memoria en caso que sea necesario. -}

operations
    fun isIn(e: in, c : ConjuntoFinito of T) ret b: bool
    {- chequea si un elemento e pertenece a un conjunto c. -}

    fun isEmpty(c : ConjuntoFinito of T) ret b: bool
    {- chequea si un conjunto es vacío. -}

    proc union(c1: ConjuntoFinito of T, c2: ConjuntoFinito of T)
    {- une un conjunto a otro. -}

    proc intersection(c1: ConjuntoFinito of T, c2: ConjuntoFinito of T)
    {- insersectar un conjunto con otro. -}

    proc diff(c1: ConjuntoFinito of T, c2: ConjuntoFinito of T)
    {- obtiene la diferencia de dos conjuntos. -}
```

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.

```
implement ConjuntoFinito of T where

type ConjuntoFinito = List of T
```

```
fun emptyC() ret c : ConjuntoFinito of T
    c := empty()
end fun

proc add(in e: T, in/out c : ConjuntoFinito of T)
    addl(e, c)
end proc

fun isIn(e: in, c : ConjuntoFinito of T) ret b: bool
    var i: nat

    i := length(c)
    b := false

    while (i > 0 && not b) do
        b := index(c, i) = e
    od
end fun

fun isEmpty(c : ConjuntoFinito of T) ret b: bool
    b := is_empty(c)
end fun

proc union(c1: ConjuntoFinito of T, c2: ConjuntoFinito of T)
    concat(c1, c2)
end proc

proc intersection(c1: ConjuntoFinito of T, c2: ConjuntoFinito of T)
    var cTemp : ConjuntoFinito of T
    var n := nat

    cTemp := emptyC()

    if (length(c1) > length(c2)) then
        n := length(c2)
    else
        n := length(c1)
    fi

    for i := 1 to n) do
        if (c1[i] = c2[i]) then
            add(c1[i], cTemp)
        fi
    od

    destroy(c1)
    union(c1, cTemp)
end proc

proc diff(c1: ConjuntoFinito of T, c2: ConjuntoFinito of T)
    var cTemp : ConjuntoFinito of T
    var n := nat

    cTemp := emptyC()
```

```
    if (length(c1) < length(c2)) then
        n := length(c2)
    else
        n := length(c1)
    fi

    for i := 1 to n) do
        if (not c1[i] = c2[i]) then
            add(c1[i], cTemp)
        fi
    od

    destroy(c1)
    union(c1, cTemp)
end proc
```

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.

```
implement ConjuntoFinito of T where

type ConjuntoFinito = List of T

fun emptyC() ret c : ConjuntoFinito of T
    c := empty()
end fun

proc add(in e: T, in/out c : ConjuntoFinito of T)
    var i, k: nat
    var x : T

    i := length(c)
    k := length(c) + 1
    isRepeated := false

    while (i > 0 && not isRepeated) do
        x := index(c, i)
        isRepeated := x = e

        if (x > e) then
```

```
            k := i
        fi
    od

    if (i = 0 && not isRepeated) then
        add_at(c, k, e)
    fi
end proc

fun isIn(e: in, c : ConjuntoFinito of T) ret b: bool
    var i: nat

    i := length(c)
    b := false

    while (i > 0 && not b) do
        b := index(c, i) = e
    od
end fun

fun isEmpty(c : ConjuntoFinito of T) ret b: bool
    b := is_empty(c)
end fun

proc union(c1: ConjuntoFinito of T, c2: ConjuntoFinito of T)
    for i := longN downto length(c2) do
        add(c2[i], c1)
    od
end proc

proc intersection(c1: ConjuntoFinito of T, c2: ConjuntoFinito of T)
    var cTemp : ConjuntoFinito of T
    var n := nat

    cTemp := emptyC()

    if (length(c1) > length(c2)) then
        n := length(c2)
    else
        n := length(c1)
    fi

    for i := n to 1) do
        if (c1[i] = c2[i]) then
            add(c1[i], cTemp)
        fi
    od

    destroy(c1)
    union(c1, cTemp)
end proc

proc diff(c1: ConjuntoFinito of T, c2: ConjuntoFinito of T)
    var cTemp : ConjuntoFinito of T
    var n1, n2 := nat
```

```
    cTemp := emptyC()

    if (length(c1) < length(c2)) then
        n1 := length(c2)
        n2 := length(c1)
    else
        n1 := length(c1)
        n2 := length(c2)
    fi

    for i := n1 downto 1 do
        if (i > n2 || not c1[i] = c2[i]) then
            add(c1[i], cTemp)
        fi
    od

    destroy(c1)
    union(c1, cTemp)
end proc
```