AYED II
Lautaro Gastón Peralta

Práctico 2.2

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

```
operations
    fun is_empty(l : List of T) ret b : bool
    {- Devuelve True si l es vacía. -}

    fun head(l : List of T) ret e : T
    {- Devuelve el primer elemento de la lista l -}
    {- PRE: not is_empty(l) -}

    proc tail(in/out l : List of T)
    {- Elimina el primer elemento de la lista l -}
    {- PRE: not is_empty(l) -}

    proc addr (in/out l : List of T,in e : T)
    {- agrega el elemento e al final de la lista l. -}

    fun length(l : List of T) ret n : nat
    {- Devuelve la cantidad de elementos de la lista l -}

    proc concat(in/out l : List of T,in l0 : List of T)
    {- Agrega al final de l todos los elementos de l0
      en el mismo orden.-}

    fun index(l : List of T,n : nat) ret e : T
    {- Devuelve el n-ésimo elemento de la lista l -}
    {- PRE: length(l) > n -}

    proc take(in/out l : List of T,in n : nat)
    {- Deja en l sólo los primeros n
      elementos, eliminando el resto -}

    proc drop(in/out l : List of T,in n : nat)
    {- Elimina los primeros n elementos de l -}

    fun copy_list(l1 : List of T) ret l2 : List of T
    {- Copia todos los elementos de l1 en la nueva lista l2 -}
```

```
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
        l:=l->next
        free(p)
    od
    l:=null
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 : pointer to (Node of T)
        var q : pointer to (Node of T)
        alloc(q)
        q->elem:=e
        q->next:=null
        if (not is_empty(l)) then
                p:=l
                while (p->next != null) do
                        p:=p->next
                od
                p->next:=q
        else
                l:=q
        fi
end proc

fun length (l : List of T) ret n : nat
        var p : pointer to (Node of T)
        n:=0
        p:=l
        while (p!=null) do
                n:=n+1
                p:=p->next
        od
end fun

proc concat (in/out l : List of T, in l0 : List of T)
        var p : pointer to (Node of T)
        var q : pointer to (Node of T)
        p:=l
        q:=l0
        if (not is_empty(l)) then
```

```
                while (p->next != null) do
                        p:=p->next
                od
                p->next:=q
        else
                p:=q
        fi
end proc

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

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

          p := l

          {- voy a "avanzar" el puntero p hasta el elemento n -}
          for i:=1 to n-1 do
                   if p!=null  then
                           p := p->next
                   fi
          od

        if p!=null then
                q := p
                p := p->next
                q->next := null
        fi

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

end proc

proc drop (in/out l : List of T,in n : nat)
  var i : nat
  i := 0
  while i < n and (l != null) do
```

```
    var p : pointer to (Node of T)
    p := l
    l := p->next
    free(p)
    i := i + 1
  od
end proc

fun copy_list (l1 : List of T) ret l2 : List of T
        if (not is_empty(l1)) then
                var p := pointer to (Node of T)
                p:=l1
                while (p!=null) do
                        l2->elem:=p->elem
                        l2->next:=p->next
                        p:=p->next
                od
        else
                l2:=null
        fi
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
                    elems : array[1..N] of T
                    size : nat
                  end tuple

fun empty() ret l : List of T
        l.size:=0
end fun

proc addl (in e : T, in/out l : List of T)
        for i:=l.size downto 1 do
                l.elems[i+1]:= l.elems[i]
        od
        l.elems[1]:=e
        l.size:= l.size+1
end proc

proc destroy (in/out l : List of T)
        l->size:=0 —-> ò usar skip (consultar).
end proc
```

```
fun is_empty (l : List of T) ret b : bool
        b := l.size = 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)
  if (l.size = 1) then
    l.size = 0
  else
    for i = 1 to l.size-1 do
      l.elem[i] = l.elem[i + 1]
    od
  fi
  l.size := l.size-1
end proc

proc addr (in/out l : List of T, in e : T)
        l.elems[l.size+1]:=e
        l.size:=l.size+1
end proc

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

proc concat (in/out l : List of T, in l0 : List of T)
  // Verificar si hay espacio suficiente en l para concatenar l0
  if (l.size + l0.size > N) then
    error("No hay suficiente espacio para concatenar las listas")
    return
  fi

  // Copiar los elementos de l0 a l
  for i := 1 to l0.size do
    l.elems[l.size + i] := l0.elems[i]
  od

  // Actualizar el tamaño de l
  l.size := l.size + l0.size
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.size := n ‘min’ l.size
end proc

proc drop (in/out l : List of T,in n : nat)
   for i:= n down to 1 do
      l.elems[i] := l.elems [i+1]
      l.size := l.size-1
   od
end proc


fun copy_list (l1 : List of T) ret l2 : List of T
        if (not is_empty(l1)) then
                for i:=1 to l1.size do
                        l2.elems[i]:=l1.elems[i]
                od
                l2.size:=l1.size
        else
                l2.size:=l1.size
        fi
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.

   AYUDA: Puede ayudarte usar las operaciones copy, take y drop.

```
proc add_at (in/out l : List of T, in e : T, in n : nat)
   var l1 : List of T
   copylist(l,l1)
   take(l,n)
   drop(l1,n)
   addl(l1,e)
   concat(l, l1)
   destroy(l1)
end proc
```

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?

a)

spec Tablero **of** T

CONSTRUCTORES:

fun initpartido () **ret** t : Tablero **of** T
{-Crea un nuevo partido-}

proc golA (**in**/**out** t : Tablero **of** T)
{-registra un tanto del equipo A-}

proc golB (**in**/**out** t : Tablero **of** T)
{-registra un tanto del equipo B-}

OPERACIONES

fun is_empty_tablero (t : Tablero **of** T) **ret** b : **bool**
{-comprueba si el tablero esta en cero-}

fun hizo_gol_A (t : Tablero **of** T) **ret** b : **bool**
{-comprueba si A hizo un gol-}

fun hizo_gol_B (t : Tablero **of** T) **ret** b : **bool**
{-comprueba si B hizo un gol-}

fun ganando_A (t : Tablero **of** T) **ret** b : **bool**
{-comprueba si A va ganando-}

fun ganando_B (t : Tablero **of** T) **ret** b : **bool**
{-comprueba si B va ganando-}

fun empate (t : Tablero **of** T) **ret** b : **bool**
{-comprueba si estan empatando-}

```
proc suma_n (in/out t : Tablero of T, in n : nat)
{-anota un numero n de tantos a un equipo-}

proc resta_n (in/out t : Tablero of T, in n : nat)
{-resta un numero n de tantos a un equipo-}
```

b)

```
Implement Tablero of T where

type Tablero of T = tuple
                      golA: Counter
                      golB: Counter
                    end tuple
```

CONSTRUCTORES:

```
fun initpartido () ret t : Tablero of T
  init(t.golA)
  init(t.golB)
end fun

proc golA (in/out t : Tablero of T)
  inc(t.golA)
end proc

proc golB (in/out t : Tablero of T)
  inc(t.golB)
end proc
```

OPERACIONES

```
fun is_empty_tablero (t : Tablero of T) ret b : bool
  b := is_init(t.golA) ^ is_init(t.golB)
end fun

fun hizo_gol_A (t : Tablero of T) ret b : bool
  b := not is_init(t.golA)
end fun

 fun hizo_gol_B (t : Tablero of T) ret b : bool
  b := not is_init(t.golB)
end fun

fun empate (t : Tablero of T) ret b : bool
  while t.golB != 0 ^ t.golA != 0 do
    decr (t.golB)
    decr (t.golA)
   od
 b := is_init (t.golB) ^ is_init (t.golA)
```

```
 end fun

fun ganando_A (t : Tablero of T) ret b : bool
   if (not empate (t)) then
     while t.golB != 0 ^ t.golA != 0 do
       decr (t.golB)
       decr (t.golA)
   od
      b := is_init (t.golA)
   else
      b:= false
   fi
end fun

fun ganando_B (t : Tablero of T) ret b : bool
 if (not empate (t)) then
   while t.golB != 0 ^ t.golA != 0 do
   decr (t.golB)
   decr (t.golA)
   od
      b := is_init (t.golB)
 else
       b:= false
fi
 end fun

proc suma_nA (in/out t : Tablero of T, in n : nat)
   for i = 1 to n do
      golA(t)
   od
end proc

proc suma_nB (in/out t : Tablero of T, in n : nat)
   for i = 1 to n do
      golB(t)
   od
end proc

proc resta_nA (in/out t : Tablero of T, in n : nat)
   for i = 1 to n do
      decr(golA)
   od
end proc

proc resta_nB (in/out t : Tablero of T, in n : nat)
   for i = 1 to n do
      decr(golB)
   od
```

end proc

c)

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** c : set **of** T
{-Crea un conjunto vacio-}

proc addtoSet (**in**/**out** c : set **of** T, **in** e : T)
{-Agrega un elemento a un conjunto-}

{-OPERADORES: -}
fun exist (c : set **of** T, e : T) **ret** b : **bool**
{-Chequea si un elemento pertenece a un conjunto c-}

fun is_emptySet (c : set **of** T) **ret** b : **bool**
{-Chequea si un conjunto es vacio-}

proc unionSet (c : set **of** T, c1 : set **of** T)
{-Une un conjunto set a otro conjunto set-}

proc intersecSet (c : set **of** T, c1 : set **of** T)
{-Interseca un conjunto set a otro conjunto set-}

proc diferenciaSet (c : set **of** T, c1 : set **of** T)
{-Obtiene la diferencia entre el set c y el set c1-}

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.

a)
implement set **of** T **where**
type set **of** T = List **of** T

```
{-CONSTRUCTORES: -}
fun emptySet () ret c : set of T
   c := empty()
end fun

proc addtoSet (in/out c : set of T, in e : T )
    if not exist(c, e) then
          addl(e, c)
    fi
end proc

{-OPERADORES: -}
fun exist (c : set of T, e : T) ret b : bool
   b := false
   for i:=0 to length (c) do
      b:= b v (index (c, i) = e)
   od
end fun

fun is_emptySet (c : set of T) ret b : bool
   b := is_empty(c)
end fun

proc unionSet ( c : set of T, c1 : set of T)
   for i:=0 to length(c1) do
         if (exist(c,index(c1,i))) then
                addl(c,index(c1,i))
         fi
    od
end proc

proc intersecSet ( c : set of T, c1 : set of T)
 var i : nat
 i := 0
 while i < length (c) do
   if (not exist (c, index (c1,i))) then
      remove_index (c, i) → implementar
   else
      i:=i+1
   fi
 od
end proc

proc diferenciaSet ( c : set of T, c1 : set of T)
var i : nat
i := 0
while i < length (c) do
   if (exist (c, index (c1,i)) = true) then
```

```
      remove_index (c, i)
   else
      i := i +1
   fi
od
end proc
proc remove_index (in/out c : List of t, in n : nat)
   var c1 : List of T
   c1 := copy_list (c)
   take (c,n-1)
   drop (c1, n)
   concat (c,c1)
   destroy (c1)
end proc
```

FALTA HACER EL 6B