```
/*************************************************************************
***************
*********                PRÁCTICA 9 - Listas                    ********
*************************************************************************
*************/

function cons(elemento , lista) {          // con_agregadoAdelanteDeLaLista_
  /*
     PROPÓSITO:  Describir la lista que resulta de agregar el elemento de la lista dada,
     adelante de todos los demás.
     PARÁMETROS:
        * elemento: es de un tipo cualquiera Elemento.
        * lista: es de tipo Lista de Elementos.
     RESULTADO: [Elementos].
     PRECONDICIONES:
        * Ninguna.
  */

  return ( [elemento] ++ lista )
}

function snoc(lista , elemento) {          // laLista_con_AgregadoAtrás
  /*
     PROPÓSITO:  Describir la lista que resulta de agregar el elemento de la lista dada,
     al final de todos los demás.
     PARÁMETROS:
        * elemento: es de un tipo cualquiera Elemento.
        * lista: es de tipo Lista de Elementos.
     RESULTADO: [Elementos].
     PRECONDICIONES:
        * Ninguna.
  */
```

```
  return ( lista ++ [elemento] )
}

function secuenciaAritméticaDeNúmerosDe_A_(valorInicial, valorFinal) {
  /*
    PROPÓSITO:  Describir la lista que tiene todos los números desde el número
    "valorInicial" hasta el número "valorFinal"
    PARÁMETROS:
      * valorInicial: Número - descripción.
      * valorFinal: Número - descripción.
    RESULTADO: Una lista de tipo Lista de Números. Si valorFinal < valorInicial,
    la lista estará vacía.
    PRECONDICIONES:
      * Ninguna.
  */

  próximoNúmero := valorInicial
  listaHastaAhora := []
  repeat (valorFinal - valorInicial + 1) {
    listaHastaAhora := listaHastaAhora ++ [próximoNúmero]
    próximoNúmero := próximoNúmero + 1
  }
  return (listaHastaAhora)
}

function enTotal_IgualesA_(cantidad, elemento) {
  /*
    PROPÓSITO:  Describir la lista con cantidad de elementos iguales al elemento dado.
    PARÁMETROS:
      * cantidad: Número - descripción.
```

```
        * elemento: es de un tipo cualquiera "Elemento".
      RESULTADO: Una lista de tipo Lista de Elementos. Si "cantidad" <= 0, la lista estará
      vacía.
      PRECONDICIONES:
        * Ninguna
  */

  listaConstruídaHastaAhora := []    //Inicializar un acumulador SOLAMENTE tiene sentido en
una acumulación
  repeat (cantidad) {
    listaConstruídaHastaAhora := [elemento] ++ listaConstruídaHastaAhora
  }
  return (listaConstruídaHastaAhora)
}

function filaActual() {
  /*
    PROPÓSITO:  Describir la fila actual del tablero como una lista de celdas.
    RESULTADO: Una lista de tipo Lista de Celdas.
    PRECONDICIONES:
      * Ninguna.
    OBSERVACIONES: Precisamos la función celdaActual definida antes.
  */

  filaLeída := []
  IrAlBorde(Oeste)
  while (puedeMover(Este)) {
    filaLeída := filaLeída ++ [celdaActual()]   //Se agrega al final para que el orden sea el mismo
que en el que fueron recorridas
    Mover(Este)
  }
  return (filaLeída ++ [celdaActual()])
```

```
}

function tableroActual() {
  /*
    PROPÓSITO:  ....
    RESULTADO: Una lista de tipo Lista de Filas. (lista de listas de celdas)
    PRECONDICIONES:
      * Ninguna.
    OBSERVACIONES: Precisamos la función filaActual recién definida.
  */

  tableroLeído := []
  IrAlBorde(Norte)
  while (puedeMover(Sur)) {
    tableroLeído := tableroLeído ++ [filaActual()]
    Mover(Sur)
  }
  return (tableroLeído ++ [filaActual()])
}

/*
  primero(<expLista>)
    PROPÓSITO: describe el primer elemento de la lista dada
    PRECONDICIÓN: la lista dada no es vacía
    PARÁMETRO: la lista es de tipo Lista de “Elementos”
    RESULTADO: un valor de tipo “Elemento”
*/

/*
  resto(<expLista>)      //O SinElPrimero_
```

```
      PROPÓSITO: describe una lista con los elementos de la lista dada, excepto que sin el
primero de ellos
      PRECONDICIÓN: la lista dada no es vacía
      PARÁMETRO: la lista es de tipo Lista de “Elementos”
      RESULTADO: un valor de tipo Lista de “Elementos”
*/

/*
   esVacía(<expLista>)
      PROPÓSITO: indica si la lista es vacía
      PRECONDICIÓN: ninguna
      PARÁMETRO: la lista es de tipo Lista de “Elementos”
      RESULTADO: un valor de tipo Booleano
*/

function segundo(lista) {
   /*
      PROPÓSITO:  Describir el segundo elemento de la lista dada.
      PARÁMETROS:
         * lista: del tipo Lista de Elementos.
      RESULTADO: Un valor de tipo Elemento
      PRECONDICIONES:
         * La lista dada tiene al menos 2 elementos.
   */

   return (primero(resto(lista)))      //El segundo queda primero luego de sacar el primero
}

function sinLosDosPrimeros(lista) {
   /*
      PROPÓSITO:  Describir una lista con los elementos de la lista dada, excepto los
```

```
    primeros 2.
    PARÁMETROS:
      * lista: del tipo Lista de Elementos.
    RESULTADO: Un valor de tipo de Lista Elemento
    PRECONDICIONES:
      * La lista dada tiene al menos 2 elementos.
  */

  return (resto(resto(lista)))      //Al sacar el primero 2 veces seguidas, hay 2 elementos menos
}

function tercero(lista) {
  /*
    PROPÓSITO:  Describir el tercer elemento de la lista dada.
    PARÁMETROS:
      * lista: del tipo Lista de Elementos.
    RESULTADO: Un valor de tipo Elemento
    PRECONDICIONES:
      * La lista dada tiene al menos 3 elementos.
  */

  return (primero(sinLosDosPrimeros(lista)))      //Y al sacar 2, el que queda primero es el tercero
}

/*
  Listas:
    Son datos con estructura.
    Tienen muchas partes, pero no siempre la misma cantidad.
    Se pueden crear a través de funciones constructoras.
    Se puede obtener información de ellas a través de funciones de acceso.
```

```
    Se pueden hacer recorridos sobre los elementos de una lista.

    Son un tipo de datos muy poderoso y útil.
*/

/**************************************************************************
***************

**********          Pr. 9 - Ej. 4 direccionesAlBorde          ********

**************************************************************************
**************/

function direccionesAlBorde() {
  /*
    PROPÓSITO:  Describir una lista de direcciones hacía la cual el cabezal no se puede
    mover.
    RESULTADO: Una lista de tipo Lista de Direcciones.
    PRECONDICIONES:
      * Ninguna.
    OBSERVACIONES:
      * Si el cabezal se puede mover en todas las direcciones, el resultado es una lista
      vacía.
  */

  direccionesEncontradas := []
  direcciónActual := minDir()
  while (direcciónActual /= maxDir()) {
    listaHastaAhora := direccionesEncontradas ++
dirección_SiNoPuedeMover(direcciónActual)
    direcciónActual := siguiente(direcciónActual)
  }
  return (direccionesEncontradas ++ dirección_SiNoPuedeMover(direcciónActual))
}
```

```
function dirección_SiNoPuedeMover(direcciónAMover) {
  /*
    PROPÓSITO:  Describe una lista de direcciones de 1 elemento si el cabezal no puede
    moverse en la dirección dada, o una lista vacía en caso contrario.
    PARÁMETROS:
      * direcciónAMover: Dirección - La dirección a evaluar si el cabezal no se puede mover.
    RESULTADO: Una Lista de Direcciones.
    PRECONDICIONES:
      * Ninguna.
  */

  return (choose  [dirección] when (not puedeMover(dirección))
                  [] otherwise)
}

/*************************************************************************
***************
**********              Pr. 9 - Ej. 5 esSingular_                 ********
*************************************************************************
**************/

function esSingular_(listaDeElementos) {
  /*
    PROPÓSITO:  Indicar si la lista dada tiene 1 sólo elemento.
    PARÁMETROS:
      * listaDeElementos: Lista de Elementos - La lista a evaluar si tiene 1 sólo elemento.
    RESULTADO: Booleano.
    PRECONDICIONES:
      * Ninguna.
  */

  //return (resto(listaDeElementos) == [])  // Otra forma de resolverlo.
```

```
  return (not esVacía(listaDeElementos) && esVacía(resto(listaDeElementos)))
}

/**************************************************************************
***************
**********              Pr. 10 - Ej. 7 longitudDe_                ********
**************************************************************************
**************/

function longitudDe_(listaDeElementos) {
  /*
    PROPÓSITO:  Describir la longitud númerica que tiene la lista dada.
    PARÁMETROS:
      * listaDeElementos: Lista de Elementos - La lista de la cual se va a averiguar que
      cantidad de elementos tiene.
    RESULTADO: Número. Igual o mayor a 0. La cantidad es finita.
    PRECONDICIONES:
      * Ninguna.
     OBSERVACIONES:
      * Se usa foreach para recorrer toda la lista dada.
  */

  cantidadDeElementosContadas := 0
  foreach elemento in listaDeElementos {
    cantidadDeElementosContadas := cantidadDeElementosContadas + 1
  }
  return (cantidadDeElementosContadas)
}

/**************************************************************************
***************
**********              Pr. 10 - Ej. 8 a. sumatoriaDe_              ********
```

```
**********************************************************************
**************/

function sumatoriaDe_(listaDeNúmeros) {
  /*
    PROPÓSITO:  Describir la sumatoria de todos los números de la lista dada.
    PARÁMETROS:
      * listaDeNúmeros: Lista de Números - La lista de números a evaluar cuanto
      es la sumatoria de sus elementos numéricos.
    RESULTADO: Número.
    PRECONDICIONES:
      * Ninguna.
     OBSERVACIONES:
      * Se usa foreach para recorrer toda la lista dada.
  */

  sumatoriaHastaAhora := 0
  foreach número in listaDeNúmeros {
    sumatoriaHastaAhora := sumatoriaHastaAhora + número
  }
  return (sumatoriaHastaAhora)
}

/**********************************************************************
***************
**********          Pr. 10 - Ej. 8 b. productoriaDe_          ********
**********************************************************************
**************/

function productoriaDe_(listaDeNúmeros) {
  /*
    PROPÓSITO:  Describir el producto entre todos los números de la lista dada.
```

```
    PARÁMETROS:
      * listaDeNúmeros: Lista de Números - La lista de números a evaluar el
      producto entre todos sus elementos numéricos.
    RESULTADO: Número.
    PRECONDICIONES:
      * Ninguna.
    OBSERVACIONES:
      * Se usa foreach para recorrer toda la lista dada.
  */

  productoriaHastaAhora := unoSi_CeroSino(not esVacía(listaDeNúmeros))
  foreach número in listaDeNúmeros {
    productoriaHastaAhora := productoriaHastaAhora * número
  }
  return (productoriaHastaAhora)
}

/**************************************************************************
***************
**********          Pr. 10 - Ej. 10 reversoDe_          ********
**************************************************************************
**************/

function reversoDe_(listaAInvertir) {
  /*
    PROPÓSITO:  Describir una lista de elementos como la lista dada, pero con el orden
    de los elementos invertidos.
    PARÁMETROS:
      * listaAInvertir: [Elementos] - La lista de la cual se obtendrán los elementos
      para retornarlos en forma invertida en una nueva lista.
    RESULTADO: [Elementos]
    PRECONDICIONES:
```

```
        * Ninguna.
    */

    elementosInvertidosHastaAhora := []
    foreach elementoAInvertir in listaAInvertir {
        elementosInvertidosHastaAhora := cons(elementoAInvertir,
elementosInvertidosHastaAhora)
    }
    return (elementosInvertidosHastaAhora)
}

/**************************************************************************
***************
**********                Pr. 10 - Ej. 14 contiene_A_              ********
**************************************************************************
**************/

function contiene_A_(listaDeElementos, elementoAEncontrar) {
    /*
        PROPÓSITO:  Indicar si el elemento dado está en la lista dada.
        PARÁMETROS:
            * listaDeElementos: [Elemento] - La lista de elementos a evaluar si tiene el
            elemento que se está buscando.
            * elementoAEncontrar: Elemento - El elemento que se quiere evaluar si está en la
            lista dada.
        RESULTADO: Booleano.
        PRECONDICIONES:
            * Ninguna.
    */

    elementosQueFaltanVer := listaDeElementos
    while (not esVacía(elementosQueFaltanVer) &&
```

```
        primero(elementosQueFaltanVer) /= elementoAEncontrar) {
            elementosQueFaltanVer := resto(elementosQueFaltanVer)
    }
    return (not esVacía(elementosQueFaltanVer))
}

/**************************************************************************
***************
**********                Pr. 10 - Ej. 15 sinDuplicados_          ********
**************************************************************************
**************/

function sinDuplicados_(listaDeElementos) {
    /*
        PROPÓSITO:  Describir una lista en base a la lista dada pero sin que tenga ninguno de
        sus elementos repetidos luego de su primera aparición.
        PARÁMETROS:
            * listaDeElementos: [Elemento] - La lista de elementos a evaluar y quitar sus
            elementos duplicados.
        RESULTADO: [Elemento]
        PRECONDICIONES:
            * Ninguna.
    */

    sinDuplicadosEncontrados := []
    foreach elemento in listaDeElementos {
        sinDuplicadosEncontrados := sinDuplicadosEncontrados ++
                        singular_Si_(elemento, not contiene_A_(sinDuplicadosEncontrados,
elemento))
    }
    return (sinDuplicadosEncontrados)
}
```

```
/*************************************************************************
***************

**********          Pr. 10 - Ej. 17 algunoDe_Entre_Y_          ********

*************************************************************************
**************/


function algunoDe_Entre_Y_(listaDeNúmeros, desde, hasta) {
  /*
    PROPÓSITO:  Indicar si la lista dada contiene algún número que se encuentra entre
    el rango de "desde" y "hasta", pero sin incluirlos.
    PARÁMETROS:
      * listaDeNúmeros: [Elemento] - La lista de números a evaluar si alguno de sus
      elementos está en el rango buscado.
      * desde: Número - El extremo menor del intervalo a evaluar.
      * hasta: Número - El extremo mayor del intervalo a evaluar.
    RESULTADO: Booleano.
    PRECONDICIONES:
      * "desde" debe ser menor a "hasta"
  */

  númerosQueFaltanVer := listaDeNúmeros
  while (not esVacía(númerosQueFaltanVer) && (not
esValor_Entre_Y_(primero(númerosQueFaltanVer)), desde, hasta)) {
      númerosQueFaltanVer := resto(númerosQueFaltanVer)
  }
  return (not esVacía(númerosQueFaltanVer))


/*  OTRA SOLUCIÓN SIN SUBTAREA - VALENTINA - CONSULTAR


  listaDeRango:= [(número1 + 1) .. (número2 - 1)] // +1 Y -1 porque es estrictamente mayor o
menor.
```

```
        while (not esVacía(listaDeRango) && not contiene_A_(lista, primero(listaDeRango))) {
                listaDeRango:= sinElPrimero(listaDeRango)
        }
        return (not esVacía(listaDeRango))
*/
}

function esValor_Entre_Y_(númeroAEvaluar, mínimoDelRango, máximoDelRango) {
  /*
    PROPÓSITO:  Indicar si el "númeroAEvaluar" es mayor al "mínimoDelRango" o si es menor
    al "máximoDelRango"
    PARÁMETROS:
      * númeroAEvaluar: Número - El número que se desea evaluar si está dentro del rango
      de valores dados.
      * mínimoDelRango: Número - El extremo menor del intervalo a evaluar.
      * máximoDelRango: Número - El extremo mayor del intervalo a evaluar..
    RESULTADO: Booleano.
    PRECONDICIONES:
      * "mínimoDelRango" es menor a "máximoDelRango"
  */

  return (númeroAEvaluar > mínimoDelRango || númeroAEvaluar < máximoDelRango)
}

/**************************************************************************
***************
**********          Pr. 10 - Ej. 18 lista_estáIncluidaEn_          ********
**************************************************************************
**************/

function lista_estáIncluidaEn_(primeraLista, segundaLista) {
  /*
```

```
    PROPÓSITO:  Indicar si todos los elementos de la primera lista dada están incluídos
    en la segunda segunda lista dada.
    PARÁMETROS:
      * primeraLista: [Elemento] - La lista que se desea saber si está contenida en la
      segunda lista dada.
      * segundaLista: [Elemento] - La lista a evaluar si tiene todos los elementos de
      la primera lista dada.
    RESULTADO: Booleano.
    PRECONDICIONES:
      * Las listas dadas deben ser del mismo tipo de elementos.
      * Cada una de las listas dadas no pueden tener elementos duplicados.
  */

  elementosIncluídosVistos := []
  foreach elemento in primeraLista {
    elementosIncluídosVistos := elementosIncluídosVistos ++
                    singular_Si_(elemento, contiene_A_(segundaLista, elemento))
  }
  return (primeraLista == elementosIncluídosVistos)
}

/*    OTRA SOLUCIÓN PERO CON WHILE - BÚSQUEDA

  elementosQueFaltanVer := primeraLista
  while (not esVacía(elementosQueFaltanVer) && contiene_A_(segundaLista,
primero(elementosQueFaltanVer))){
    elementosQueFaltanVer := resto(elementosQueFaltanVer)
  }
  return (esVacía(elementosQueFaltanVer))
*/
```

```
/*****************************************************************************
***************
*********          Pr. 10 - Ej. 19 estáOrdenada_          ********
*****************************************************************************
**************/

function estáOrdenada_(listaDeNúmeros) {
  /*
    PROPÓSITO:  Indicar si la lista dada tiene sus elementos numéricos ordenados de menor
    a mayor.
    PARÁMETROS:
      * listaDeNúmeros: [Número] - La lista a evaluar si tiene sus elementos ordenados
      de menor a mayor.
    RESULTADO: Booleano.
    PRECONDICIONES:
      * Ninguna. Es total.
  */

  elementosPorVer := listaDeNúmeros
  while (not esVacía(elementosPorVer) &&          // Evalúa que la lista dada no sea vacía,
para no hacer boom ni tener precon
      not esSingular_(elementosPorVer) &&         // Evalúa mientras la lista dada no tenga 1
sólo elemento, xq tengo q preguntar por el 1ero y el 2do elemento
      (primero(elementosPorVer) <= segundo(elementosPorVer))) {       // Evalúa si el 1er
elemento es menor o igual que el 2do para saber si está ordenada.
        elementosPorVer := resto(elementosPorVer)          // Si se cumplen las condiciones,
está ordenada, saco 1 elemento de la lista de la variable para
  }                                                 // evaluar la próxima iteración.
  return (esSingular_(elementosPorVer))       // Pregunta si la lista que tengo en la variable
tiene un sólo elemento. Si es así, toda la lista está ordenada,
}

/*****************************************************************************
***************
```

```
**********              Pr. 10 - Ej. 20 posiciónDe_enLaQueAparece_          ********
*************************************************************************
**************/

function posiciónDe_enLaQueAparece_(lista, elemento) {
  /*
    PROPÓSITO:  Describir en que posición de la lista dada se encuentra la 1era aparición
    del elemento dado.
    PARÁMETROS:
      * lista: [Elemento] - descripción.
      * elemento: Elemento - del mismo tipo que los elementos de "lista" - descripción.
    RESULTADO: Número.
    PRECONDICIONES:
      * La lista dada no es vacía.
      * Existe al menos 1 aparición del elemento dado en la lista dada.
  */

  elementosPorRecorrer := lista
  posiciónDelElemento := 0
  while (primero(elementosPorRecorrer) /= elemento) {
    posiciónDelElemento := posiciónDelElemento + 1
    elementosPorRecorrer := resto(elementosPorRecorrer)
  }
  return (posiciónDelElemento)
}

/*************************************************************************
***************
**********              Pr. 10 - Ej. 21 sinLaPrimeraAparicióndeDe_en_          ********
*************************************************************************
**************/
```

```
function sinLaPrimeraAparicionDe_en_(elemento, lista) {
  /*
    PROPÓSITO:  Describir una lista en base a la lista dada, pero sin la primera aparición
    del elemento dado, si es que éste aparece en dicha lista.
    PARÁMETROS:
      * elemento: Elemento, del mismo tipo que los elementos de la lista dada - El
      elemento a buscar y eliminar su primera aparición, si hubiere.
      * lista: [Elemento] - La lista a buscar el elemento dado.
    RESULTADO: [Elemento]
    PRECONDICIONES:
      * Ninguna. Función Total
  */

  return (choose  lista_SinElElemento_(lista, elemento) when (contiene_A_(lista, elemento))
                                    lista otherwise)
}

function lista_SinElElemento_(lista, elemento) {
  /*
    PROPÓSITO:  Describir una lista en base a la lista dada, pero sin la primera aparición
    del elemento dado.
    PARÁMETROS:
      * lista: [Elemento] - La lista donde se va a buscar el elemento dado.
      * elemento: Elemento, del mismo tipo que los elementos de la lista dada - El
      elemento a buscar y filtrar su primera aparición en la lista retornada.
    RESULTADO: [Elemento]
    PRECONDICIONES:
      * La lista dada no es vacía.
      * La lista dada tiene al menos 1 aparición del elemento dado entre sus elementos.
  */
```

```
  listaConElElementoABuscar := lista
  listaSinElElementoBuscado := []
  while (primero(listaConElElementoABuscar) /= elemento) {
    listaSinElElementoBuscado := snoc(listaSinElElementoBuscado,
primero(listaConElElementoABuscar))
    listaConElElementoABuscar := resto(listaConElElementoABuscar)
  }
  return (listaSinElElementoBuscado ++ resto(listaConElElementoABuscar))
}

/**************************************************************************
***************
**********          Pr. 10 - Ej. 22 a. mínimoElementoDe_          ********
**************************************************************************
**************/

function mínimoElementoDe_(listaDeNúmeros) {
  /*
    PROPÓSITO:  Describir el número menor que tiene la lista dada.
    PARÁMETROS:
      * listaDeNúmeros: [Número] - La lista a buscar cual es el número menor.
    RESULTADO: Número.
    PRECONDICIONES:
      * La lista dada no es vacía.
  */

  mínimoHastaAhora := primero(listaDeNúmeros)
  foreach número in resto(listaDeNúmeros) {
    mínimoHastaAhora := mínimoEntre_Y_(mínimoHastaAhora, número)
  }
  return (mínimoHastaAhora)
}
```

```
/**************************************************************************
***************
**********          Pr. 10 - Ej. 22 b. sinElMínimoElemento_          ********
**************************************************************************
**************/

function sinElMínimoElemento_(listaDeNúmeros) {
  /*
    PROPÓSITO:  Describir una lista basada en la lista dada pero sin la primera aparición
    del número menor de ella.
    PARÁMETROS:
      * listaDeNúmeros: [Número] - La lista a evaluar pero sin la primera aparición del
      número menor.
    RESULTADO: [Número]
    PRECONDICIONES:
      * La lista dada no es vacía.
  */

  return (sinLaPrimeraAparicióndeEn_(mínimoElementoDe_(listaDeNúmeros),
listaDeNúmeros))
}

/**************************************************************************
***************
**********          Pr. 10 - Ej. 22 c. lista_ordenada          ********
**************************************************************************
**************/

function lista_ordenada(listaDeNúmeros) {
  /*
    PROPÓSITO:  Describir, a partir de la lista dada, una lista con los mismos elementos
    pero ordenados de menor a mayor.
```

```
    PARÁMETROS:
      * listaDeNúmeros: [Número] - La lista base para retornar una lista con los mismos
      elementos pero ordenados de menor a mayor.
    RESULTADO: [Número]
    PRECONDICIONES:
      * La lista dada no es vacía.
  */

  elementosOrdenados := [mínimoElementoDe_(listaDeNúmeros)]
  elementosPorOrdenar := sinElMínimoElemento_(listaDeNúmeros)
  while (not esVacía(elementosPorOrdenar)) {
    elementosOrdenados := snoc(elementosOrdenados,
mínimoElementoDe_(elementosPorOrdenar))
    elementosPorOrdenar := sinElMínimoElemento_(elementosPorOrdenar)
  }
  return (elementosOrdenados)
}

/**************************************************************************
***************
**********          Pr. 10 - Ej. 23 a. máximoElementoDe_          ********
**************************************************************************
**************/

function máximoElementoDe_(listaDeNúmeros) {      //Hacer con foreach!!! ARREGLAR
  /*
    PROPÓSITO:  Describir, de la lista dada,  el número mayor que en ella se encuentra.
    PARÁMETROS:
      * listaDeNúmeros: [Número] - La lista a buscar cual es el número mayor.
    RESULTADO: Número.
    PRECONDICIONES:
      * La lista dada no es vacía.
```

```
  */

  máximoVisto := primero(listaDeNúmeros)
  númerosPorComparar := resto(listaDeNúmeros)
  while (not esVacía(númerosPorComparar)) {
    máximoVisto := máximoEntre_Y_(máximoVisto, primero(númerosPorComparar))
    númerosPorComparar := resto(listaDeNúmeros)
  }
  return (máximoVisto)
}

/**************************************************************************
***************
**********                Pr. 10 - Ej. 23 b. sinElMáximoElemento_          ********
**************************************************************************
**************/

function sinElMáximoElemento_(listaDeNúmeros) {
  /*
    PROPÓSITO:  Describir, a partir de una lista dada, una lista de elementos pero sin la
    primera aparición del mayor número.
    PARÁMETROS:
      * listaDeNúmeros: [Número] - La lista a evaluar cual es el mayor número para
      eliminarlo de la lista retornada
    RESULTADO: [Número]
    PRECONDICIONES:
      * La lista dada no es vacía.
  */

  return (sinLaPrimeraAparicióndeDe_en_(máximoElementoDe_(listaDeNúmeros),
listaDeNúmeros))
}
```

```
/*************************************************************************
***************
**********        Pr. 9 - Números Invertidos - Implementar                ********
*************************************************************************
**************/

function númerosDe_Invertidos(númeroADarVuelta) {
  /*
    PROPÓSITO:  Describir el número dado en forma invertida.
    PARÁMETROS:
      * númeroADarVuelta: Número - El número a invertir.
    RESULTADO: Número.
    PRECONDICIONES:
      * "númeroAInvertir" debe tener al menos 2 dígitos.
  */

  númeroAInvertir := númeroADarVuelta
  númeroAlRevés := 0
  while (númeroAInvertir > 0) {
    númeroAlRevés := númeroAlRevés * 10 + (númeroAInvertir mod 10)
    númeroAInvertir := númeroAInvertir div 10
  }
  return( númeroAlRevés )
}

/*************************************************************************
***************
**********        Pr. 9 - singular_Si_     Fuera de la práctica pero  útil     ********
*************************************************************************
**************/

function singular_Si_(elemento, condición) {
```

```
  /*
     PROPÓSITO:  Describir una lista que contiene el único elemento dado si se cumple la
     condición dada. Caso contrario describe una lista vacía.
     PARÁMETROS:
        * elemento: Cualquier Tipo.
        * condición: Booleano - La condición a evaluar si el elemento se va agregar a la
        lista.
     RESULTADO: [Elemento]
     PRECONDICIONES:
        * Ninguna.
  */

  return (choose [elemento] when (condición)
                 [] otherwise)
}

/*************************************************************************
***************
**********          Último Elemento de la Lista          ********
*************************************************************************
**************/

function últimoDeLa_(lista){
//Describe el último elemento que tiene la lista
//La lista dada no puede ser vacía

  últimoVisto := primero(lista)
  foreach elemento in resto(lista){
     últimoVisto := primero([elemento])
  }
  return (últimoVisto)
}
```

```
/*************************************************************************
***************
**********                PROCESAMIENTO DE LISTAS                ********
*************************************************************************
**************/

// ESQUEMA DE RECORRIDO DE FILTRO - Los elementos de la lista con los que me quiero quedar
function elemento_FiltradoDe_(elementoAFiltrar, listaAFiltrar) {

   elementosFiltradosAlMomento := []
   foreach elemento in listaAFiltrar {
      elementosFiltradosAlMomento := elementosFiltradosAlMomento ++
                        singular_Si_(elemento, seCumpleCondiciónDeFiltroPara_(elemento))
   }
   return (elementosFiltradosAlMomento)
}

function mínimoEntre_Y_(valor1, valor2) {
   /*
      PROPÓSITO:  Describe el valor mínimo entre los valores dados.
      PARÁMETROS:
         * valor1: Número - El primer valor a comparar si es el menor.
         * valor2: Número - El segundo valor a comparar si es el menor.
      RESULTADO: Número
      PRECONDICIONES:
         * Ninguno.
   */

   return (choose valor1 when (valor1 < valor2)
            valor2 otherwise)
```

```
}
function máximoEntre_Y_(valor1, valor2) {
  /*
    PROPÓSITO:  Describe el valor máximo entre los valores dados.
    PARÁMETROS:
      * valor1: Número/Color/Dirección/Booleano - El primer valor a comparar si es el mayor.
      * valor2: Número/Color/Dirección/Booleano - El segundo valor a comparar si es el
mayor.
    RESULTADO: Número
    PRECONDICIONES:
      * Los valores dados deben ser del mismo tipo.
  */

  return (choose valor1 when (valor1 > valor2)
            valor2 otherwise)
}
```