Práctico 11 - Arreglo con Tope y Registros Variantes

Programación 1
InCo - Facultad de Ingeniería, Udelar

1. Dadas las siguientes declaraciones:

```
const
   CANT_PERS = ... ;{valor entero mayor estricto a 0}
   MAX_CADENA = ... ;{valor entero mayor estricto a 0}
type
   Cadena = record
                letras : array [1..MAX_CADENA] of char;
                largo : 0..MAX_CADENA;
            end;
   Persona = record
                nombre : Cadena;
                edad : 0..120;
                estado : (casado, soltero, divorciado);
                salario : Real;
                exenciones : 0..maxint;
             end;
var
   juanita : Persona;
   grupo : array [1..CANT_PERS] of Persona;
```

   Determine cuáles de las siguientes instrucciones son válidas:

   I) `grupo[1] := juanita`

   II) `grupo[1].nombre := 'juanita'`

   III) `read (grupo[1].estado)`

   IV) `with grupo do writeln (nombre)`

   V)
```
with grupo[100] do
begin
   read (edad)
end
```

   VI)
```
with juanita do
begin
   nombre := grupo[50].nombre;
   salario := grupo[1].salario
end
```

   Solución: I, V, VI

2. Una sociedad genealógica tiene un arreglo de registros con datos de personas. Los datos son, para cada persona, su nombre, la fecha de su nacimiento y los índices en el arreglo de los registros de los datos de su padre y de su madre. Cada nombre de persona aparece una única vez. Se definen los siguientes tipos de datos para representar esta realidad:

```
const
    MAXPERSONAS = ...; {valor entero mayor estricto a 0}
```

```
    MAXCAD      = ...; {valor entero mayor estricto a 0}

type
   Cadena = record
               letras : array [1..MAXCAD] of char;
               largo : 0..MAXCAD;
            end;
   Fecha = record
              dia : 1..31;
              mes : 1..12;
              anio : 0..maxint;
           end;
   Persona = record
                   nombre : Cadena;
                   fechNac : Fecha;
                   indMadre, indPadre : 0..MAXPERSONAS;
                end;
   Familia = record
                pers : array[1..MAXPERSONAS] of Persona;
                tope : 0..MAXPERSONAS;
             end;
```

Los campos `indMadre` e `indPadre` de una persona contienen el índice de los registros de la madre y del padre en el arreglo `pers` de la familia, o cero en caso de no disponer de la información correspondiente. En caso de ser distinto de cero, se asume que es un índice válido del arreglo.

(a) Escriba la función `cadenasIguales` que, dadas dos cadenas, determina si son iguales.

```
function cadenasIguales (cad1, cad2 : Cadena): Boolean;
```

(b) Escriba el procedimiento `desplegarCadena` que, dada una cadena, la despliega en la salida.

```
procedure desplegarCadena (cad: Cadena);
```

(c) Escriba el procedimiento `antepasados` que, dado el nombre de una persona en el parámetro `usted` y una familia en el parámetro `historia`, despliegue en la salida los nombres y fechas de nacimiento del padre y de la madre de la persona de nombre `usted` (si es que se dispone de la información correspondiente). En caso de que la persona de nombre usted no esté registrada, no se desplegará nada.

```
procedure antepasados (usted : Cadena; historia : Familia);
```

(d) Escriba un programa principal que permita probar los subprogramas de las partes anteriores, declarando toda variable que sea necesaria. También debe definir cualquier otro subprograma auxiliar que necesite para carga y/o exhibición de datos.

```
program ej2;
const
  MAXPERSONAS = 20;
  MAXCAD      = 10;

type
  Cadena =  record
              letras : array [1..MAXCAD] of char;
              largo : 0..MAXCAD;
            end;
  Fecha = record
            dia : 1..31;
            mes : 1..12;
            anio : 0..maxint;
          end;
  Persona = record
              nombre : Cadena;
              fechNac : Fecha;
              indMadre, indPadre : 0..MAXPERSONAS;
```

```
            end;
  Familia = record
              pers : array[1..MAXPERSONAS] of Persona;
              tope : 0..MAXPERSONAS;
            end;

var
  f: Familia; i, j, indh, indp: integer; n: Cadena;

function cadenasIguales (cad1, cad2 : Cadena): Boolean;
var i: integer;
begin
  if cad1.largo = cad2.largo then
  begin
    i := 1;
    while (i <= cad1.largo) and (cad1.letras[i] = cad2.letras[i]) do
      i := i + 1;
    cadenasIguales := i > cad1.largo
  end
  else
    cadenasIguales := false
end;

procedure desplegarCadena (cad: Cadena);
var i: integer;
begin
    for i := 1 to cad.largo do
      write(cad.letras[i])
end;

procedure antepasados (usted : Cadena; historia : Familia);
var i: integer;
begin
  i := 1;
  while (i <= historia.tope) and
          (not cadenasIguales(usted, historia.pers[i].nombre)) do
    i := i + 1;
  if i <= historia.tope then
    with historia.pers[i] do
    begin
      if indPadre <> 0 then
        begin
          writeln('Información del padre:');
          desplegarCadena(historia.pers[indPadre].nombre);
          with historia.pers[indPadre].fechNac do
            writeln(' ', dia:1, '/', mes:1, '/', anio:1)
        end;
      if indMadre <> 0 then
        begin
          writeln('Información de la madre:');
          desplegarCadena(historia.pers[indMadre].nombre);
          with historia.pers[indMadre].fechNac do
            writeln(' ', dia:1, '/', mes:1, '/', anio:1)
        end;
    end
end;

procedure cargarNombre (var nombre: Cadena);
var i: integer; c: char;
```

```
begin
  with nombre do
  begin
    read(c);
    i := 0;
    while (c in ['a'..'z']) or (c in ['A'..'Z'])  do
    begin
      i := i + 1;
      letras[i] := c;
      read(c)
    end;
    largo := i
  end;
end;

procedure cargarPersona (var p: Persona);
begin
  cargarNombre(p.nombre);
  with p.fechNac do
    readln(dia, mes, anio);
  p.indMadre := 0;
  p.indPadre := 0
end;

begin
  write('Indique cantidad de personas a leer: ');
  readln(j);
  for i := 1 to j do
  begin
    write('Ingrese los datos de la persona ',i:1,
      ' de la forma "nombre dia mes anio": ');
    cargarPersona(f.pers[i])
  end;
  f.tope := j;
  write('Indique cantidad de relaciones madre-hijo/a: ');
  readln(j);
  for i := 1 to j do
  begin
    write('Ingrese los indices de la madre y del hijo/a: ');
    readln(indp, indh);
    f.pers[indh].indMadre := indp
  end;
  write('Indique cantidad de relaciones padre-hijo/a: ');
  readln(j);
  for i := 1 to j do
  begin
    write('Ingrese los indices de la padre y del hijo/a: ');
    readln(indp, indh);
    f.pers[indh].indPadre := indp
  end;
  write('Ingrese el nombre de una persona para obtener sus padres: ');
  cargarNombre(n);
  antepasados(n, f)
end.
```

3. Dado el tipo `Cadena` definido para almacenar hasta `MAX` caracteres:

```
const
    MAX = ...; {valor mayor estricto a 0}
```

```
type
   Cadena = record
               letras : array [1..MAX] of char;
               largo : 0..MAX;
            end;
```

(a) Escriba un procedimiento llamado `cargarCadena` que tenga como parámetro una cadena de caracteres y la cargue con caracteres leidos de la entrada estándar. Al ingresar los caracteres, se utilizará un punto (.) para marcar el fín de la cadena (el cual no forma parte de la misma, solo será tipeado para marcar su finalización). En caso de que se ingresen más de `MAX` caracteres, solamente se cargarán los primeros `MAX`, descartando los restantes.

```
procedure cargarCadena (var cad: Cadena);
var i: integer; c: char;
begin
  with cad do
  begin
    read(c);
    i := 0;
    while (i < MAX) and (c <> '.') do
    begin
      i := i + 1;
      letras[i] := c;
      read(c)
    end;
    largo := i;
    readln;
  end;
end;
```

(b) Escriba una función llamada `contarOcurrencias` que tenga como parámetro una cadena de caracteres llamada `frase` y una variable de tipo carácter llamada `letra`. La función devuelve el número de apariciones del carácter `letra` en la cadena `frase`.

```
function contarOcurrencias (frase: Cadena; letra: char): integer;
var i,ac: integer;
begin
  ac := 0;
  with frase do
    for i := 1 to largo do
      if letras[i] = letra then
        ac := ac + 1;
  contarOcurrencias := ac
end;
```

(c) Escriba una función llamada `existeVocal` que tenga como parámetro una cadena de caracteres y determine si en la cadena hay o no alguna letra vocal. La función devuelve `true` en caso afirmativo y `false` en caso negativo.

```
function existeVocal (frase: Cadena): boolean;
var i: integer;
begin
  i := 1;
  with frase do
  begin
    while (i <= largo) and not (letras[i] in ['a','e','i','o','u']) do
      i := i + 1;
    existeVocal := i <= largo
  end
end;
```

4. Se desea implementar un procedimiento que calcule las raíces de la ecuación cuadrática $ax^2 + bx + c = 0$ donde a, b, y c son coeficientes reales. El procedimiento debe determinar si la ecuación tiene dos raíces reales y distintas, una raíz real doble o dos raíces complejas conjugadas. Para devolver el resultado, se definirá un tipo de datos `TipoRaices` mediante una estructura de registro con variante, de modo tal que contemple los tres casos posibles. El cabezal del procedimiento es el siguiente:

   `procedure raices (a,b,c : real; var r : TipoRaices);`

   (a) Defina el tipo `TipoRaices` utilizando la estructura de registro con variante. Debe definir también cualquier tipo de datos auxiliar que pueda necesitar.

```
type TipoRaices = record
    case tipo: (reales, realDoble, complejasConjugadas) of
      reales: (
        r1, r2: real
      );
      realDoble: (
        r: real
      );
      complejasConjugadas: (
        rr, ri: real
      )
  end;
```

   (b) Implemente el procedimiento `raices`, de acuerdo al comportamiento descrito.

```
procedure raices (a,b,c : real; var r : TipoRaices);
var disc, x1, x2: real;
begin
  disc := sqr(b) - 4*a*c;
  x1 := -b/(2*a);
  with r do
    if disc < 0 then
    begin
      tipo := complejasConjugadas;
      rr := x1;
      ri := sqrt(-disc)/(2*a)
    end
    else if disc = 0 then
    begin
      tipo := realDoble;
      r := x1
    end
    else
    begin
      tipo := reales;
      x2 := sqrt(disc)/(2*a);
      r1 := x1 + x2;
      r2 := x1 - x2
    end
end;
```

5. Se considera el tipo de datos `Nerr` que es la unión de los números naturales (N) con el conjunto `Err`. El conjunto `Err` se define como `{diverr, reserr, argerr}`, donde `diverr` es el error de la división por cero, `reserr` es el error de la resta con resultado negativo y `argerr` es el error de cualquier operación donde alguno de sus argumentos no es natural.

   Se definen las siguientes operaciones sobre elementos del tipo `Nerr`:

   `division: Nerr x Nerr -> Nerr`

   - Si a pertenece a N y b pertenece a N$-\{0\}$ $\Rightarrow$ `division(a,b) = a DIV b`;

- Si a pertenece a N y $b = 0 \Rightarrow$ `division(a,b) = diverr`;
- Si a pertenece a Err o b pertenece a $Err \Rightarrow$ `division(a,b) = argerr`;

`resta: Nerr x Nerr -> Nerr`

- Si a pertenece a N, b pertenece a N y $a \geq b \Rightarrow$ `resta(a,b) = a - b`;
- Si a pertenece a N, b pertenece a N y $a < b \Rightarrow$ `resta(a,b) = reserr`;
- Si a pertenece a Err o b pertenece a $Err \Rightarrow$ `resta(a,b) = argerr`;

`suma: Nerr x Nerr -> Nerr`

- Si a pertenece a N y b pertenece a $N \Rightarrow$ `suma(a,b) = a + b`;
- Si a pertenece a Err o b pertenece a $Err \Rightarrow$ `suma(a,b) = argerr`;

`producto: Nerr x Nerr -> Nerr`

- Si a pertenece a N y b pertenece a N $\Rightarrow$ `producto(a,b) = a * b`;
- Si a pertenece a Err o b pertenece a $Err \Rightarrow$ `producto(a,b) = argerr`;

(a) Defina el tipo de datos `Err`.

```
Err = (diverr, reserr, argerr);
```

(b) Defina el tipo de datos `Nerr`. Defina también cualquier tipo de datos auxiliar que pueda necesitar.

```
Nat = 0 .. maxint;
Tipo = (natural,error);

(* tipo de naturales con error *)
Nerr = record
          case es : Tipo of
              natural : (num : Nat);
              error   : (cod : Err);
       end;
```

(c) Escriba los siguientes procedimientos que implementan, respectivamente, las operaciones division, resta, suma y producto del tipo Nerr.

```
procedure division (a, b: Nerr; var resu: Nerr);
procedure resta (a, b: Nerr; var resu: Nerr);
procedure suma (a, b: Nerr; var resu: Nerr);
procedure producto (a, b :Nerr; var resu: Nerr);
```

```
(* subprogramas auxiliares *)
function hayArgError(n1,n2 : Nerr) : boolean;
begin
   hayArgError := (n1.es = error) or (n2.es = error)
end;

procedure asignarError (terr : Err; var n : Nerr);
begin
   n.es  := error;
   n.cod := terr
end;

procedure asignarNatural (num : Nat; var n : Nerr);
begin
   n.es  := natural;
   n.num := num
```

```
end;

(* division *)
procedure division (n1,n2 : Nerr; var n3 : Nerr);
begin
     if hayArgError(n1,n2) then
      asignarError(argerr,n3)
     else if n2.num = 0 then (*sabemos que n2 es un natural*)
             asignarError(diverr, n3)
    else
       asignarNatural(n1.num div n2.num, n3)
end;

procedure resta (n1,n2 : Nerr; var n3 : Nerr);
begin
     if hayArgError(n1, n2) then
      asignarError(argerr, n3)
     else if n1.num < n2.num then (*sabemos que n1 y n2 son los dos naturales*)
             asignarError(reserr, n3)
    else
       asignarNatural(n1.num - n2.num, n3)
end;

procedure suma (n1,n2 : Nerr; var n3 : Nerr);
begin
     if hayArgError(n1, n2) then
      asignarError(argerr, n3)
     else
      asignarNatural(n1.num + n2.num, n3)
end;

procedure producto (n1,n2 : Nerr; var n3 : Nerr);
begin
     if hayArgError(n1, n2) then
      asignarError(argerr, n3)
     else
      asignarNatural(n1.num * n2.num, n3)
end;
```

6. Se desea implementar el producto de matrices de elementos de tipo `Nerr` (del ejercicio anterior). Las matrices a operar podrán tener diferentes dimensiones. Se sabe que si una matriz tiene dimensión `mxn`, entonces `m` y `n` son enteros positivos que nunca superarán constantes conocidas `Y` y `X` respectivamente.

   El producto entre matrices de tipo `Nerr` se define de manera análoga al producto de matrices de números naturales con la suma y el producto para el tipo `Nerr` dado en el ejercicio anterior.

   Sea la matriz `m1` de dimensión `m x n` y la matriz `m2` de dimensión `p x q`. Si `n = p`, entonces el producto `m1 x m2` tendrá dimensión `m x q`, en caso contrario diremos que el producto falla.

   (a) Defina el tipo `MNerr`, que representa las matrices de dimensiones `m x n` de tipo `Nerr`, para cualquier `m` entre 1 e Y, y para cualquier `n` entre 1 y X. Puede asumir que X e Y son constantes con valores mayores o iguales que 1. Además `MNerr` debe tener un valor de error `merr` para el caso en que el producto de matrices falle. Defina también cualquier tipo enumerado auxiliar que pueda necesitar.

```
(* Tipos de las matrices *)
rangoM = 1 .. X;
rangoN = 1 .. Y;
tipoMat = record
              matri : array [rangoM,rangoN] of Nerr;
              topefila : rangoM;
              topecol : rangoN;
```

```
          end;

MErr = (arguerr, dimensionerr);
Tip = (matriz, errr);
MNErr = record
          case es :Tip of
                matriz: (mat : tipoMat);
                errr:   (cod : MErr);
          end;
```

(b) Implemente el procedimiento `mprod`, el cual recibe dos matrices `m1` y `m2`, retornando en `resu` el producto `m1 x m2` o `merr` en caso de que dicho producto falle.

```
procedure mprod (m1, m2: MNerr; VAR resu: MNerr);
```

```
(* procedimiento para multiplicar matrices *)
procedure mprod (m1: MNErr; m2: MNerr; var m3: MNerr);
var filprod,colprod,iter:integer;
    sum, produ : Nerr;
begin
   if (m1.es = errr) or (m2.es = errr)
   then begin
      m3.es := errr;
      m3.cod := arguerr
        end
   else (* sabemos que es matriz *)
   if (m1.mat.topecol <> m2.mat.topefila)
   then begin
           m3.es := errr;
           m3.cod := dimensionerr
        end
   else begin (* compatibles: m1.mat.topecol = m2.mat.topefila,
                    el prod es m1.mat.topefila x m2.mat.topecol  *)
      m3.es := matriz;
      m3.mat.topefila := m1.mat.topefila;
      m3.mat.topecol := m2.mat.topecol;
      for filprod:= 1 to m1.mat.topefila do
        for colprod:= 1 to m2.mat.topecol do
        begin
          sum.es := natural;
          sum.num := 0;
          produ.es := natural;
          for iter:= 1 to m2.mat.topefila do
          begin
            producto(m1.mat.matri[filprod,iter],m2.mat.matri[iter,colprod],produ);
            suma(sum,produ,sum)
          end;
          m3.mat.matri[filprod,colprod] := sum
        end
    end
end;
```

7. En una isla del Caribe una banda de piratas se gana la vida asaltando barcos mercantes. Anualmente en la isla se realiza un evento multitudinario llamado "la entrega de los premios Calavera". Para este año los piratas están pensando en entregar el premio *Calavera de oro* al pirata que haya conseguido más dinero asaltando barcos para la banda. Como usualmente los piratas discuten a quién le corresponden los premios en trifulcas interminables y sangrientas, este año la comisión directiva de la banda ha decidido informatizar el registro de logros de los piratas en los distintos asaltos, con la esperanza de terminar así con algunas de las discusiones sobre los créditos que le corresponden a cada pirata.

Los logros de los piratas durante el año se representan mediante las siguientes declaraciones:

```
const
    MAXPIRATAS   = ...; {valor entero mayor estricto a 0}
    MAXASALTOS   = ...; {valor entero mayor estricto a 0}
    MAXDIGITOSCI = ...; {valor entero mayor estricto a 0}
    MAXCADENA    = ...; {valor entero mayor estricto a 0}

type
    TipoCadena = record
        letras: array [1..MAXCADENA] of char;
        tope:   0 .. MAXCADENA
    end;

    TipoCI = array [1..MAXDIGITOSCI] of '0'..'9';

    TipoFecha  = record
        dia: 1..31;
        mes: 1..12;
        anio: 0..maxint;
    end;

    TipoAsalto = record
        nombre_barco: TipoCadena;
        fecha: TipoFecha;
        botin: integer;
    end;

    ConjuntoAsaltos = record
        asaltos: array [1..MAXASALTOS] of TipoAsalto;
        tope: 0..MAXASALTOS
    end;

    TipoCausaMuerte = (asesinato, enfermedad, accidente);

    TipoPirata = record
        nombre: TipoCadena;
        ci: TipoCI;
        case estaVivo: boolean of
            true:  (asaltos: ConjuntoAsaltos);
            false: (causaMuerte: TipoCausaMuerte; fechaMuerte: TipoFecha)
    end;

    Banda = record
        pirata: array [1..MAXPIRATAS] of TipoPirata;
        tope: 0..MAXPIRATAS
    end;
```

(a) Implemente la función `dineroObtenidoPorPirata` que calcula la suma de dinero obtenida por el pirata de cedula `CI`, en el año `anio` por la banda `b`. En caso de que el pirata se encuentre muerto o no se encuentre en la banda debe retornar **0**.

```
function dineroObtenidoPorPirata(pirata: TipoCI; anio: integer; b:Banda) : integer;
```

Se sugiere implementar primero las siguientes funciones auxiliares:

```
function ciIguales (ci1, ci2: TipoCI): boolean;
(* Retorna true si ci1 y ci2 son iguales y false en caso contrario*)

function contarDinero (ca: ConjuntoAsaltos; anio:integer): integer;
(* Retorna la suma del dinero obtenido en los asaltos del conjunto ca realizados
```

```
durante el año anio. *)
```

```
function ciIguales (ci1, ci2: TipoCI): boolean;
var i: integer;
begin
  i := 1;
  while (i <= MAXDIGITOSCI) and (ci1[i] = ci2[i]) do
    i := i + 1;
  CIiguales := i > MAXDIGITOSCI
end;

function contarDinero (ca: ConjuntoAsaltos; anio:integer): integer;
var i, dinero: integer;
begin
  dinero := 0;
  for i := 1 to ca.tope do
    if ca.asaltos[i].fecha.anio = anio then
      dinero := dinero + ca.asaltos[i].botin;
  contarDinero := dinero
end;

function dineroObtenidoPorPirata(pirata: TipoCI; anio: integer; b:Banda) : integer;
var i: integer;
begin
  i := 1;
  while (i <= b.tope) and not ciIguales(b.pirata[i].ci, pirata) do
    i := i + 1;
  if (i > b.tope) or not b.pirata[i].estaVivo then
    dineroObtenidoPorPirata := 0
  else
    dineroObtenidoPorPirata := contarDinero(b.pirata[i].asaltos, anio)
end;
```

(b) Implemente un procedimiento el cual, dada una banda de piratas `piratas` y un año `anio`, retorna en el parámetro `piratasMerecedores` las cédulas de los piratas vivos merecedores del premio *Calavera de Oro*. Tener en cuenta que varios piratas pueden coincidir en la cantidad de dinero obtenido para la banda.

```
procedure hallarGanadores (piratas:Banda; anio:integer; var piratasMerecedores: ConjuntoCIs);
```

El tipo ConjuntoCIs se declara como sigue:

```
type
    ConjuntoCIs = record
        cedulas: array [1..MAXPIRATAS] of TipoCI;
        tope: 0..MAXPIRATAS
    end;
```

```
procedure hallarGanadores (piratas:Banda; anio:integer;
  var piratasMerecedores: ConjuntoCIs);
var i, j, max, dinero: integer;
begin
  { Encontrar primer pirata vivo y utilizar como pirata merecedor provisional }
  j := 1;
  while (j <= piratas.tope) and not piratas.pirata[j].estaVivo do
    j := j + 1;
  if j <= piratas.tope then
  begin
    piratasMerecedores.tope := 1;
    piratasMerecedores.cedulas[1] := piratas.pirata[j].ci;
```

```
    max := contarDinero(piratas.pirata[j].asaltos, anio)
  end
  else
    piratasMerecedores.tope := 0;

  { Comparar el botin del resto de piratas contra el botin de los piratas
    merecedores provisionales }
  for i := j+1 to piratas.tope do
    if piratas.pirata[i].estaVivo then
    begin
      dinero := contarDinero(piratas.pirata[i].asaltos, anio);
      if dinero > max then
      begin
        piratasMerecedores.tope := 1;
        piratasMerecedores.cedulas[1] := piratas.pirata[i].ci;
        max := dinero
      end
      else if dinero = max then
      begin
        piratasMerecedores.tope := piratasMerecedores.tope + 1;
        piratasMerecedores.cedulas[piratasMerecedores.tope] := piratas.pirata[i].ci
      end
    end
end;
```

8. Se desea trabajar con una aritmética de naturales de hasta 100 dígitos. Los enteros de *Pascal* no soportan dicha aritmética, por lo que se piensa utilizar la siguiente representación de naturales basada en arreglos con tope, de tal manera que las unidades queden en el primer lugar del arreglo, las decenas en el segundo, y así hasta que el dígito más significativo quede en el tope:

```
const
    MaxDig = ...; {valor entero mayor estricto a 0};
type
    Digito = 0..9;
    Natural = record
        digitos : array[1..MaxDig] of Digito;
        tope : 0..MaxDig;
    end;
```

   (a) Implemente la suma de Naturales representados en términos de la estructura anterior. Utilice el siguiente cabezal:

```
procedure sumaNaturales (a, b : Natural; var c : Natural);
```

```
procedure sumaNaturales (a, b : Natural; var c : Natural);
var i, topeMenor, lleva, suma: integer; mayor: Natural;
begin
  if a.tope < b.tope then
  begin
    topeMenor := a.tope;
    mayor := b
  end
  else
  begin
    topeMenor := b.tope;
    mayor := a
  end;
  lleva := 0;
  for i := 1 to topeMenor do
```

```
  begin
    suma := a.digitos[i] + b.digitos[i] + lleva;
    lleva := suma div 10;
    c.digitos[i] := suma mod 10
  end;
  for i := topeMenor + 1 to mayor.tope do
  begin
    suma := mayor.digitos[i] + lleva;
    lleva := suma div 10;
    c.digitos[i] := suma mod 10
  end;
  if lleva > 0 then
  begin
    c.tope := mayor.tope + 1;
    c.digitos[c.tope] := lleva;
  end
  else
    c.tope := mayor.tope
end;
```

(b) Analice cuál sería la dificultad si la representación fuera al revés, es decir que el dígito menos significativo quede en el tope y el más significativo en el primer lugar del arreglo.

9. Deseamos hallar la descomposición de un número natural ($n <= N$) en factores primos.

Ej: $8 = 2 * 2 * 2$, $36 = 2 * 2 * 3 * 3$, $37 = 37$, $20 = 2 * 2 * 5$, $105 = 3 * 5 * 7$, $600 = 2 * 2 * 2 * 3 * 5 * 5$

Vamos a suponer que dicha descomposición no involucra más de `M` números primos, donde `M` se supone una constante previamente definida en el valor adecuado. Deseamos almacenar la descomposición en un arreglo con tope:

```
type Descomp = record
                 factores : array [1..M] of Factor;
                 tope : 0..M;
               end;
```

el cual contiene los factores ordenados en forma ascendente y sus respectivos exponentes.

```
type Factor = record
              primo : 1..N;
              case multiple : boolean of
                true: (exponente : 2..N);
                false : ();
              end;
```

Los factores se representan mediante un registro con variante (record-case). Si el exponente de un factor es 1, entonces se almacena simplemente el número primo. Si por el contrario, el exponente del factor es mayor que 1, entonces se almacena el número primo y el valor del exponente.

(a) Implemente un procedimiento que reciba como entrada un número cualquiera mayor que 1 y retorne su descomposición en factores primos.

Utilizar el siguiente cabezal:

```
Procedure factorizacion (num : Integer; var listaFac : Descomp);
```

(b) Escriba un programa principal que permita probar el subprograma de la parte anterior, declarando toda variable que sea necesaria. También debe definir cualquier otro subprograma auxiliar que necesite para carga y/o exhibición de datos en la salida estándar.

```
program Pr11Ej9;
const M = 50;
      N = 50;

type Factor = record
                primo : 1..N;
                case multiple : boolean of
                  true: (exponente : 2..N);
                  false : ();
              end;

Descomp = record
            factores : array [1..M] of Factor;
            tope : 0..M;
          end;
var
   num : Integer;
   listaFac : Descomp;

procedure mostrarDescomposicion (listaFac : Descomp);
var
  i : Integer;
begin
  write(listaFac.factores[1].primo:1);
  if listaFac.factores[1].multiple then
    write('^', listaFac.factores[1].exponente:1);

  for i := 2 to listaFac.tope do
  begin
    write('*', listaFac.factores[i].primo:1);
    if listaFac.factores[i].multiple then
      write('^', listaFac.factores[i].exponente:1);
  end;
end;

procedure factorizacion (num : Integer; var listaFac : Descomp);
var
  p: 1..N;
begin
  listaFac.tope := 0;
  p := 2;
  while num <> 1 do
  begin
    if num mod p = 0 then
    begin
      listaFac.tope := listaFac.tope + 1;
      with listaFac.factores[listaFac.tope] do
      begin
        primo := p;
        num := num div p;
        if num mod p = 0 then
        begin
          multiple := true;
          exponente := 2;
          num := num div p;
          while num mod p = 0 do
          begin
            exponente := exponente + 1;
            num := num div p
          end
        end
```

```
        else
          multiple := false
      end
    end;
    p := p + 1
  end
end;

begin
  write('Ingrese un número mayor a 1: ');
  readln(num);

  factorizacion (num, listaFac);

  write(num:1, ' = ');
  mostrarDescomposicion(listaFac);
end.
```