Carnet ________________ **Nombres** __________________________________

CI2126 Computación II, Dic 2014/Mar 2015

Examen I (25%)

1.- (6 puntos) Indique con **V**(erdadero) o **F**(also) a las expresiones siguientes:

a.- `typedef double xyz` declara una variable "xyz" del tipo `double` — **F**
b.- Los punteros ocupan espacio en memoria sólo cuando son no nulos. — **F**
c.- La expresión `(*puntero).atributo` y `puntero->atributo` dan resultados iguales — **V**
d.- Si **p** es puntero a **char**, `p = malloc(32)` crea una cadena de caracteres — **V**
e.- La *lista* es una estructura del tipo FIFO (*First In First Out*). — **F**
f.- La *pila* es una estructura del tipo LIFO (*Last In First Out*). — **V**
g.- Una *cola* se puede simular con una *pila*, si "desempilar" invierte la *pila* antes. — **V**
h.- Si **p** es puntero a `float` no nulo, `p[0] = 3.5;` es una instrucción legal — **V**
i.- Si declaramos `FILE* arch; arch = fopen("a.txt", "w");` Si ∃ "a.txt", se produce error. — **F**
j.- Declarando **int* p**, entonces `p = calloc(1,sizeof(int))` crea un arreglo de 10 **ints** — **F**
k.- `float call ( float (*f) ( float x), x) { return f(x); }` es correcto — **V**
l.- `double *f (double x); f = sqrt; double x = f(4.0);` es correcto — **F**

4.- (3 pts) Dado el siguiente programa escrito en lenguaje C, indique qué hace el mismo y cuáles valores produciría cuando al ejecutarlo el usuario introduce como entrada "**2**":

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

float doble (float x) { return 2*x; }
float cuad  (float x) { return x*x; }
float ident (float x) { return   x; }
float uno   (float x) { return 1.0; }

int main()
{
       typedef float (*Fn_R_R) (float);
       static Fn_R_R fn[] = {doble, cuad, ident, uno};
       float x;
       int i;

       fprintf (stdout, "\n Introduzca un numero real: ");
       fscanf  (stdin,  "%f", &x);

       for (i=0; i<3; i++)
       {
              float r = fn[i] ( fn[i+1] ( x + i ) );
              fprintf(stdout, "\n %4.2f provee los resultados r: %4.2f \n", x, r);
       }
       exit (0);
}
```

```
 Introduzca un numero real: 2

r = fn[0]( fn[1]( 2.0+0) )  ➞  doble(cuad (2.0))  ➞  2.00 provee los resultados r: 8.00
r = fn[1]( fn[2] (2.0+1) )  ➞   cuad(ident(3.0))  ➞  2.00 provee los resultados r: 9.00
r = fn[2]( fn[3] (2.0+2) )  ➞    ident(uno(4.0))  ➞  2.00 provee los resultados r: 1.00
```

2.- **(5 puntos)** Dados los TADs **Cola** y **Pila**, desarrollar una operación primitiva, **Cola_agregar_en_posicion**, que recibe como parámetros: la cola ***q***, un elemento ***e*** a agregar en la cola y un entero ***pos*** (la posición en la cual se quiere agregar el elemento ***e*** en la cola ***q***). Esta primitiva *encola* el elemento en una cola en la posición **pos** (por ejemplo si **pos** es 3, se encolaría como el tercer elemento desde el primero). En caso de que la *cola* ***q*** tenga menos de **pos** elementos, se debe agregar como último en la cola. La misma secuencia de nodos puede ser tratada como Cola y Pila. Las definiciones de **Base_s** y **Nodo_s** se dan como referencia, pero no deben ser manipuladas. ***Sólo puede usar las funciones primitivas de Pila y Cola descritas abajo, sin navegar explícitamente por los apuntadores.*** No todas las primitivas son necesarias para una implementación eficiente.

```
   /* Definición apuntador del TAD Nodo */
typedef struct Nodo_s* Nodo;

   /* Definición apuntador del TAD Pila */
typedef struct Base_s* Pila;

   /* Definición apuntador del TAD Cola */
typedef struct Base_s* Cola

 /* Dice si el apuntador p generico es nulo */
bool esVacia (void* p);      /* _pre: ∃ p */
```

```
 /* Selector para obtener un elem de un nodo */
Elem getElem (Nodo n)         /* _pre: ∃ n */

 /* Mutador para guardar un elem de un nodo */
Elem setElem (Nodo n, Elem e) /* _pre: ∃ n */

 /* Selector para obtener un elem de un nodo */
Nodo consNodo (Elem e)         /* _pre: true */

 /* Selector para obtener un elem de un nodo */
Nodo destNodo (Elem* e)        /* _pre: true */
```

```
  /* Constructor devuelve una pila vacía */
Pila consPila ();          /* _pre: true */

  /* Libera memoria ocupada por la pila */
void destPila (Pila* p);    /* _pre: ∃ p */

      /* Devuelve el tope de la pila */
Elem topePila (Pila p);     /* _pre: ∃ p */

      /* Empila e y devuelve la Pila */
Pila empilar (Pila p, Elem e);
  /* _pre: ∃ p */

  /* Desempila el tope actual de la Pila */
Pila desempilar(Pila p, Elem* e);
  /* _pre: ∃ p */
```

```
  /* Constructor devuelve una pila vacía */
Cola consCola ();           /* _pre: true */

   /* Libera memoria ocupada por la cola */
void destCola (Cola* q);      /* _pre: ∃ q */

     /* Devuelve el primero de la cola */
Elem primeroCola (Cola q);    /* _pre: ∃ q */

      /* Devuelve el ultimo de la cola */
Elem ultimoCola (Cola q);     /* _pre: ∃ q */

     /* Encola e como último de la cola */
Cola encolar (Cola q, Elem e); /* _pre: ∃ q */

     /* Desencola el primero de la cola */
Cola desencolar(Cola q, Elem* e);
  /* _pre: ∃ q */
```

a) Solución con **Pila** auxiliar **p**

```
Cola Cola_agregar_en_pos(Cola q, Elem e, int pos)
{
    Pila p;   /* Pila auxiliar */
    Elem x;
    int k=0;

    _pre( q != NULL && pos>=0 );

    p = consPila();
    while (k<pos-1 && !esVacia(q))  /* sacar los pos-1 anteriores en pila p */
    {
        desencolar(q, &x);
        empilar(p, x);
        k++;
    }
    empilar(q, e);       /* meter el elemento de primero en la cola q */
    while (!esVacia(p))  /* Devolver los pos-1 anteriores en pila p   */
    {
        desempilar(p, &x);
        empilar(q, x);
    }
    destPila(&p);
    return q;
}
```

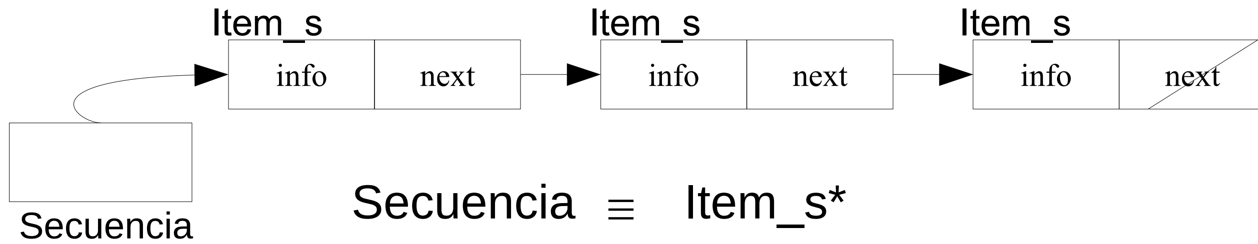
b) Solución con **Cola** auxiliar **r**;

```
Cola Cola_agregar_en_pos_otra(Cola q, Elem e, int pos)
{
    Cola r;   /* Cola auxiliar */
    Elem x;
    int k=0;

    _pre( q != NULL && pos>=0 );

    r = consCola();
    while (k<pos-1 && !esVacia(q)) /* sacar los pos-1 anteriores en cola r */
    {
        desencolar(q, &x);
        encolar(r, x);
        k++;
    }
    encolar(r, e);       /* meter el elemento de ultimo en la cola r       */
    while (!esVacia(q))  /* encolar el resto de elementos de q en cola r   */
    {
        desencolar(q, &x);
        encolar(r, x);
    }
    while (!esVacia(r))  /* es necesario traspasar elementos de r a q     */
    {
        desencolar(r, &x);
        encolar(q, x);
    }
    destCola(&r);
    return q;
}
```

3.- **(5 pts)** Usando la estructura siguiente de secuencia como simple apuntador a un Item_s.




```
   /* Definición adelantada del TDA Elem_s*/
typedef struct Item_s* Secuencia;

typedef struct Item_s {
    int       info;
    Secuencia next;
} Item_s;
```

a) Debe haser los prototipos, pre y post condiciones de las siguientes primitivas:

consSecuencia: void → **Secuencia**;	Crea una **Secuencia** vacía.
consArraySecuencia: **int** [ ] x **int** → Secuencia;	Crea un objeto **Secuencia** con los elementos de un arreglo de **n** enteros en el mismo orden.
destSecuencia: ***Secuencia** → void;	Destruye una **Secuencia** vacía.
consItem: **int** → **Secuencia**;	Crea una **Secuencia** que apunta al **Item_s** con info **i**.
esVacia: **Secuencia** → **bool**	Devuelve **true** si ***s** *es* **nulo**.
cardinalidad: **Secuencia** → **int**	Devuelve cuantos elementos tiene la secuencia.
cabeza: **Secuencia** → **int**	Devuelve *s->info*; *precondición:* s no vacía.
resto: **Secuencia** → **Secuencia**	Devuelve *s->next*; *precondición:* s no vacía.

b) Implemente una nueva primitiva, *esOrdenada*, que reciba una **Secuencia** y determine de forma ***recursiva*** si los elementos de la misma están ordenados ascendentemente. No puede destruir ni alterar el contenido de la secuencia recibida.

Ayuda: Una secuencia ***vacía*** siempre está ordenada, al igual que una secuencia con ***un*** sólo elemento.

c) Implemente una nueva primitiva, *intercalarSecuencias*, que reciba dos secuencias (que SE REQUIERE estén ordenadas ascendentemente) y devuelva una tercera secuencia que contenga los elementos de ambas secuencias intercalados, añadiendo al final aquellos sobrantes de la secuencia más larga, si es el caso. No puede destruir ni alterar el contenido de las secuencias recibidas como parámetro.

d) Haga un trozo de código que cree dos secuencias a partir de los arreglos de abajo y las intercale.

```
int A[] = { -1, 1, 4, 7, 8 };            /* DIM = 5 */
int B[] = { -1, 2, 3, 3, 5, 5, 8, 12 }; /* DIM = 8 */
```

a)

```c
/* Crea una Secuencia vacía. */
Secuencia consSecuencia( );
      /* pre: true */

/* Crea un objeto Secuencia con los elementos de
   un arreglo de n enteros en el mismo orden. */
Secuencia consArraySecuencia(int A[], int n);
      /* pre: n >= 0 */

/* Destruye una Secuencia vacía. */
void destSecuencia(Secuencia* s);
      /* pre: s != NULL */

/* Crea una Secuencia que apunta al Item_s con info i. */
Secuencia consItem(int i);
      /* pre: true */

/* Devuelve true si  s es  nulo. */
bool esVacia(Secuencia s);
      /* pre: s != NULL */

/* Devuelve cuantos elementos tiene la secuencia. */
int cardinalidad(Secuencia s);
      /* pre: s != NULL */

/* Devuelve s->info;  */
int cabeza(Secuencia s)  ;
      /* pre: s != NULL */

/* Devuelve s->next;   */
Secuencia resto(Secuencia)
      /* pre: s != NULL */
```

b.1) Solución usando recursión simple (también es de cola)

```c
bool esOrdenada(Secuencia s)
{
    if (esVacia(s) || esVacia(resto(s)))
       return true;

    if (cabeza(s) <= cabeza(resto(s)))
        return esOrdenada(resto(s));

    return false;
}
```

b.2) Solución usando recursión de cola con auxiliar

```
bool esOrdenadaAux(int valor, Secuencia s)
{
    if (esVacia(s))
        return true;
    if (valor <= cabeza(s))
        return esOrdenadaAux(cabeza(s), resto(s));
    return false;
}

bool esOrdenada(Secuencia s)
{
    if (esVacia(s))
        return true;
    return esOrdenadaAux(cabeza(s),resto(s));
}
```

c)

```
Secuencia intercalar( Secuencia A, Secuencia B)
{
    _pre( esOrdenada(A) && esOrdenada(B) );
    Secuencia x = A;
    Secuencia y = B;
    Secuencia s;
    int  n = cardinalidad(x) + cardinalidad(y);
    int* Z = malloc(n*sizeof(int));
    int  k = 0;

    while (!esVacia(x) && !esVacia(y))
    {
        if (cabeza(x < cabeza(y)))
        {
            Z[k] = cabeza(x);
            x = resto(x);
        } else {
            Z[k] = cabeza(y);
            y = resto(y);
        }
        k++;
    }

    while (!esVacia(x))
    {
        Z[k++] = cabeza(x);
        x = resto(x);
    }

    while (!esVacia(y))
    {
        Z[k++] = cabeza(y);
        y = resto(y);
    }

    s = consArraySecuencia(Z, n);
    return s;
}
```

5.- **(6 puntos)** Dado la siguiente especificación funcional sintáctica del TAD **Matriz NxM** (N filas y M columnas), diseñe la especificación formalmente en C (lo que sería el "Matriz.h")

(a) Las declaraciones necesarias para crear la matriz como estructura DINÁMICA.

```
typedef  ...  MatrizTAD_s;   /* La estructura del TAD Matriz NxM */

typedef ...   Matriz;        /* El apuntador a la Matriz         */
```

(b) Los prototipos necesarias de las operaciones de la matriz y sus precondiciones

`consMatriz :  ncolumnas x nfilas ---> Matriz`	Genera una matriz llena de ceros
`destMatriz :  Matriz ---> void`	Elimina matriz y libera recursos al sistema
`consIdentidad :  nfilas ---> Matriz`	Devuelve una matriz identidad cuadrada
`getNumFilas :  Matriz ---> Entero`	Indica la cantidad de filas de una matriz
`getNumColumnas :  Matriz ---> Entero`	Indica la cantidad de columnas de una matriz
`esCuadrada :  Matriz ---> Bool`	Indica si una matriz es cuadrada
`setValor :  Matriz x fila x columna x Real ---> Real`	Coloca un valor en la posición [fila,columna] indicada dentro de la matriz y devuelve el valor.
`getValor :  Matriz x fila x columna --->  Real`	Devuelve el valor en esa posición
`sumaMatriz :  Matriz x Matriz ---> Matriz`	Suma matrices de iguales dimensiones y devuelve una nueva matriz con el resultado
`multMatriz :  Matriz x Matriz ---> Matriz`	Multiplica matrices y devuelve una nueva matriz con el resultado. El *NumColumnas* de la primera matriz debe ser igual al *NumFilas* de la segunda
`multEscalar :  Matriz x Real ---> Matriz`	Multiplica por un escalar a toda la matriz

(c) Las declaraciones necesarias para crear la estructura de la matriz

Haga la implementación de *sumaMatriz y multMatriz*

a)

```c
typedef struct MatrizTAD_s*   Matriz;  /* El apuntador a la Matriz         */

typedef struct MatrizTAD_s {
    int numFilas;
    int numCols;
    float* celda; // tambien float[DIM][DIM], pero ineficiente
} MatrizTAD_s;   /* La estructura del TAD Matriz NxM */
```

b)

```c
/* Devuelve una matriz identidad cuadrada */
Matriz consMatriz (int nfilas, int ncolumnas);
/* pre: nfilas > 0 && ncolumnas >  0*/

/* Devuelve una matriz identidad cuadrada */
Matriz consIdentidad (int nfilas) {
/* pre: nfilas > 0 */
    return consMatriz(nfilas, nfilas);
}

/* Devuelve la cantidad de filas de una matriz */
int getNumFilas(Matriz A);
/* pre: A != NULL */

/* Devuelve la cantidad de Columnas de una matriz */
int getNumColumnas(Matriz A);
/* pre: A != NULL */

/* Indica si una matriz es cuadrada */
bool esCuadrada(Matriz A)
/* pre: A != NULL */

/* Coloca un valor en la posición [fila,columna] y devuelve el valor. */
float setValor(Matriz A, int fila, int columna, float valor);
/* pre: A != NULL && 0 < fila < getNumFilas(A) && 0 < columna < getNumColumnas(A) */

/* Devuelve el valor en esa posición */
float getValor(Matriz A, int fila, int columna);
/* pre: A != NULL && 0 < fila < getNumFilas(A) && 0 < columna < getNumColumnas(A) */

/* Suma matrices de iguales dimensiones y
   devuelve una nueva matriz con el resultado */
Matriz sumaMatriz(Matriz A, Matriz B);
/* pre: A != NULL && B != NULL
   pre: getNumFilas(A) == getNumFilas(B) && getNumColumnas(A) == getNumColumnas(B) */

/* Multiplica matrices y devuelve una nueva matriz con el resultado. El NumColumnas
   de la primera matriz debe ser igual al NumFilas de la segunda  */
Matriz multMatriz(Matriz A, Matriz B);
/* pre: A != NULL && B != NULL
   pre: 0 < fila < getNumFilas(A) && 0 < columna < getNumColumnas(A)
   pre: getNumColumnas(A) == getNumFilas(B) */

/* Multiplica por un escalar a toda la matriz */
Matriz multEscalar(Matriz A, float s);
/* pre: A != NULL */
```

c) Haga la implementación de sumaMatriz y multMatriz

```
/* Suma matrices de iguales dimensiones y devuelve
      una nueva matriz con el resultado */
Matriz sumaMatriz(Matriz A, Matriz B)
{
    int i, j ,k;
    float suma;

    _pre(A != NULL && B != NULL);
    _pre(getNumFilas(A)    == getNumFilas(B)   &&
         getNumColumnas(A) == getNumColumnas(B));

    Matriz R = consMatriz (getNumFilas(A), getNumColumnas(A));

    for (i=0; i<getNumFilas(A); i++)
    {
        suma = 0.0;
        for (j=0; j<getNumColumnas(A); j++)
        {
            suma += getValor(A, i, j) + getValor(B, i, j);
        }
        setValor(R, i, j, suma);
    }
    return R;
}

/* Multiplica matrices y devuelve una nueva matriz con l resultado.
   El NumColumnas de la matriz A debe ser igual al NumFilas de B  */
Matriz multMatriz(Matriz A, Matriz B)
{
    int i, j ,k;
    float suma;
    int NFA, NCA, NFB, NCB;

    _pre(A != NULL && B != NULL);

    NFA = getNumFilas(A);
    NCA = getNumColumnas(A);
    NFB = getNumFilas(B);
    NCB = getNumColumnas(B);

    _pre(NCA == NFB);

    Matriz R = consMatriz (NFA, NCB);

    for (i=0; i<NFA; i++)
    {
        for (j=0; j<NCB; j++)
        {
            suma = 0.0;
            for (k=0; k<NCA; k++)
            {
                suma += getValor(A, i, k) * getValor(B, k, j);
            }
            setValor(R, i, j, suma);
        }
    }
    return R;
}
```