# Sintaxis y Semántica de Lenguajes 2006.2

# TP – Sistema Bancario – Introducción

*por Ing. José María Sola*
200601013

## Temas

- Aplicación de **TADs** como herramienta para la **resolución de problemas**.
  - **Especificación** de TADs.
  - **Implementación** de TADs mediante la aplicación de diferentes herramientas provistas por el **lenguaje** y la **Biblioteca estándar** definidas por **ANSI C**.
- Aplicación de **Autómatas Finitos** para la **resolución de problemas**.

## Objetivo

El objetivo de este TP es dar una aplicación práctica a las herramientas **TADs** y **AFs**; utilizando a **ANSI C** como herramienta de implementación y a los **Lenguajes Formales** como marco teórico.

La aplicación práctica consta en el diseño de abstracciones que permitan, en forma simple, modelar conceptos básicos de un ***Sistema Bancario***. Las abstracciones son ***Fecha***, ***Transacción***, ***Cuenta*** y ***Banco***. Por cada una de ellas, se diseñará un tipo de dato abstracto (TAD). Diseñar un TAD implica el diseño de las especificaciones y de las implementaciones. La implementación se acompaña de un programa que verifique la concordancia con la especificación.

Esta sección presenta el TP en forma general, le siguen una sección con normativas presentación y una sección por cada TAD detallando sus requerimientos.

## Contexto del Problema

La empresa de desarrollo de software a la cual usted pertenece comienza a especializarse en el desarrollo de ***sistemas bancarios***. Ante la situación, la compañía decide invertir en el desarrollo de un *framework*[1] que maximice la productividad de los desarrolladores de aplicación, minimice los tiempos de desarrollo, facilite el agregado de futuras funcionalidades, y que minimice el tiempo entre la entrada de un nuevo desarrollador y el momento en cual pasa a ser productivo; es decir, un framework que contribuya a mejorar el proceso de desarrollo de software de la empresa.

La empresa decide armar un equipo, al cual usted pertenece, con la tarea de diseñar los TAD que formarán parte de este framework. Estos TAD serán probados de forma completa y se ejemplificará su uso en un programa de aplicación que simule su uso.

## Introducción

Los programadores que hagan uso de estas abstracciones construirán aplicaciones que permitan tareas como: abrir cuentas de diferente tipo, registrar transacciones, cancelar transacciones, acreditar y debitar montos, obtener saldos e informes, registrar transferencias entre cuentas de un mismo banco o de diferentes bancos, consultar estadísticas sobre clientes y sobre sus cuentas, persistir la información (a disco, por ejemplo) y luego recuperarla. Para que los programadores de las aplicaciones bancarias puedan construir los sistemas deben recibir lo necesario para trabajar con cada TAD:

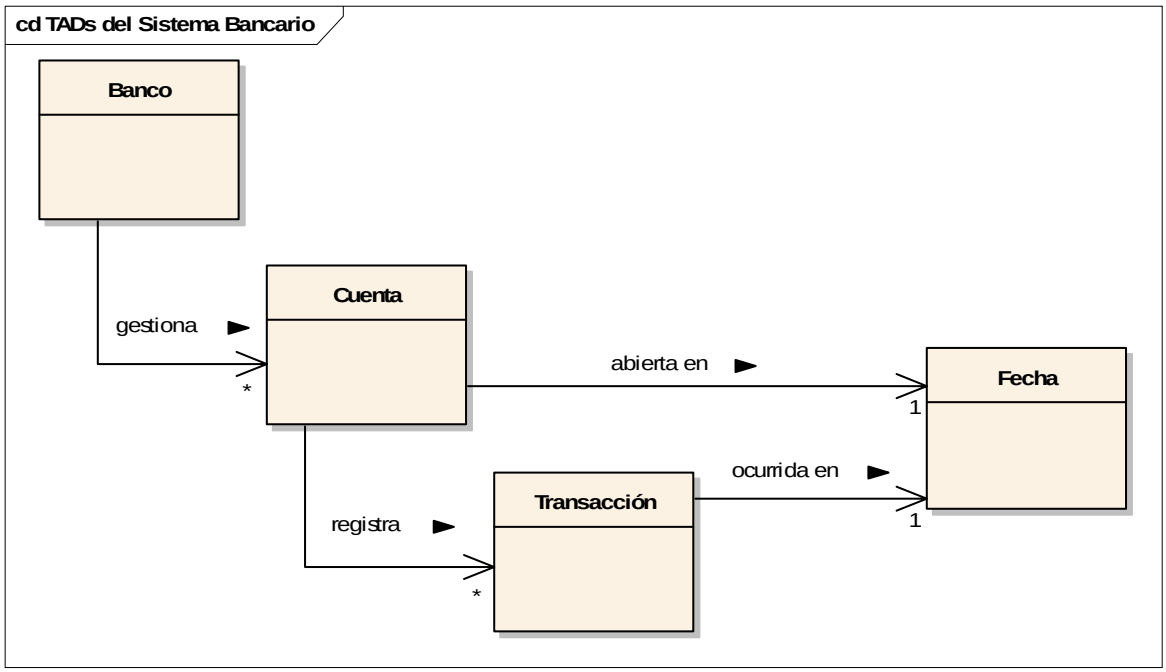
- especificación del TAD,
- documentación propia de la implementación del TAD,
- archivos encabezado y
- bibliotecas que encapsula la implementación de cada TAD.

[1] *Framework*: Estructura software sobre la cual otros desarrollos software pueden organizarse y desarrollarse. Incluye programas, bibliotecas u otros software que ayudan a desarrollar y a interconectar los diferentes componentes de un software. Están diseñados para facilitar el desarrollo, permitiendo a diseñadores y programadores dejar de lado los detalles de más bajo nivel e invertir tiempo en la implementación de los requerimientos del negocio del sistema a desarrollar. Ejemplos: *Cocoa* de *Apple* computers para la *Macintosh*, *.NET Framework* de *Microsoft* y *Struts* de *Apache*.

Los valores de estos tipos de datos serán manipulados en memoria, pero podrán ser persistidos mediante flujos de texto y flujos binarios, para luego ser recuperados. En otras palabras, podrán ser conservados en memoria externa por un tiempo más prolongado.

## *Breve Descripción de los TAD*

Un ***banco*** gestiona las ***cuentas*** de sus clientes, las cuales registran diferentes ***transacciones*** que se producen en ***fechas*** determinadas.




El *TAD Banco* permite operaciones como apertura y cierre de cuentas, transferencias, consultas de saldos y estadísticas. *El TAD Cuenta* permite operaciones como debitar, acreditar y transferir, así como también consultas estadísticas. El *TAD Transacción* registra un movimiento bancario, tiene operaciones que permiten su creación y acceso a sus componentes. El *TAD Fecha* se utiliza al registrar transacciones, es posible comparar fechas y acceder a componente de ellas. Todos los TADs permiten persistencia (i.e. almacenamiento en memoria externa, secundaria).

## Nombres de los archivos involucrados en la resolución del TP

- Fecha.c, Fecha.h, Fecha.lib, FechaAplicacion.c y FechaAplicacion.exe
- Transaccion.c, Transaccion.h, Transaccion.lib, TransaccionAplicacion.c y TransaccionAplicacion.exe.
- Cuenta.c, Cuenta.h, Cuenta.lib, CuentaAplicacion.c y CuentaAplicacion.exe.
- Banco.c, Banco.h, Banco.lib, BancoAplicacion.c y BancoAplicacion.exe.

## Nombres de los archivos que deben presentar

Deben presentar los archivos fuente, no los binarios.

- Fecha.c, Fecha.h y FechaAplicacion.c.
- Transaccion.c, Transaccion.h y TransaccionAplicacion.c.
- Cuenta.c, Cuenta.h y CuentaAplicacion.c.
- Banco.c, Banco.h y BancoAplicacion.c.

## *Secuencia de Entregas*

Orden de Presentación de los diferentes TAD:
1. TAD **Fecha**;
2. TAD **Transacción**;
3. TAD **Cuenta**;
4. TAD **Banco**.

A su vez, los componentes de cada TAD se presentarán respetando la siguiente secuencia:
1. **Especificación** del conjunto de **valores**;
2. **Especificación** del conjunto de **operaciones**;
3. Archivo **encabezado**;
4. Aplicación de **prueba**; y por último
5. **Implementación**. —

# Sintaxis y Semántica de Lenguajes 2006.2

# TP – Sistema Bancario – Normativas para la Presentación

*por Ing. José María Sola*
20061013

Para que la Cátedra acepte la presentación del trabajo por parte del equipo, se deben cumplir los requisitos de tiempo y forma, si alguno no se cumple, la presentación se considera inválida y el TP debe se recupera en una de las fechas previstas para 1er o 2do recuperatorio.

## *Forma*

- El trabajo debe presentarse en hojas **A4 abrochadas en la esquina superior izquierda**.
- En el **encabezado de cada hoja** debe figurar el **título del trabajo**, el **título de entrega**, el **código de curso**, **número de equipo** y los **apellidos de los integrantes del equipo**.
- Las hojas deben estar enumeradas en el pie de las mismas con el formato "**Hoja n de m**".
- El **código fuente de cada componente del TP** debe comenzar con un **comentario encabezado**, con todos los datos del equipo de trabajo: **curso**; **legajo**, **apellido y nombre de cada integrante del equipo** y **fecha de última modificación**.
- La **fuente** (estilo de caracteres) a utilizar en la **impresión** de los **códigos fuente** y de las **capturas de las salidas** debe ser una fuente de ***ancho fijo*** (e.g. `Courier New`, `Lucida Console`).
- Cada **TAD** del TP se presenta en una **sección separada**; a su vez, **cada sección de cada TAD** tiene las siguientes **cinco grandes sub-secciones**:
  **Nombre del TAD**. Cada sección comienza con el nombre del TAD.

  **1. Comentarios.** En esta sección el equipo puede incluir comentarios sobre la resolución. Si se deciden **desviaciones con respecto al enunciado**, en esta sección deben **enumerarse y justificarse**.

  **2. Respuestas a puntos del enunciado**. En esta sección se debe responder a los puntos incluidos en el enunciado de cada TAD. Se debe incluir el número y texto del enunciado. Si un punto del enunciado se resuelve en la especificación o implementación, se debe responder con "*resuelto en la implementación*" o "*resuelto en la especificación*".

  **3. Especificación.** Especificación ***correcta***, ***completa***, ***concisa***, ***consistente***, ***sin ambigüedades***, ***sin detalles de implementación*** y ***verificable*** de los **valores** y de las **operaciones** del TAD.

  **4. Aplicación de Prueba**
  4.1. **Código Fuente**. Listado del código fuente de la aplicación de prueba, *TAD*Aplicacion.c.
  4.2. Salidas
  4.2.1. Captura impresa de la salida del **proceso de traducción** (BCC32).
  4.2.2. Captura impresa de las salidas de la **aplicación de prueba**.
  4.2.3. Impresión de archivos de prueba de entrada y de archivos de salida generados durante la prueba.

  **5. Implementación.** Biblioteca que implementa el TAD.
  5.1. **Listado** de código fuente del **archivo encabezado**, **parte pública**, *TAD*.h.
  5.2. **Listado** de código fuente de la **definición de la Biblioteca**, **parte privada**, *TAD*.c. Las funciones privadas deberán ser precedidas por su documentación, un comentario con la documentación de la función indicando, entre otras cosas, propósito de la función, semántica de los parámetros in, out e inout.
  5.3. **Salidas**. Captura impresa de la salida del **proceso de traducción** (BCC32 y TLIB).

- El TP será acompañado por:
  **A. Copia Digitalizada.** CD con copia de **solamente los 3 archivos de *código fuente* de cada TAD** (e.g.: *Fecha.c, Fecha.h, FechaAplicacion.c*) ***No se debe entregar ningún otro archivo***.
  **B. Formulario de Seguimiento de Equipo.**

## *Tiempo*

- En cada curso se diseñará un **cronograma de entregas parciales** del TP **previas a la entrega final**.
- Luego de la aprobación del TP se **evaluará individualmente de forma oral o escrita** (**"coloquio"**) **sobre los conceptos aplicados en el TP y sobre la resolución elegida**. Para presentarse al coloquio las **evaluaciones parciales deben estar aprobadas**.
- **Entrega final del TP**: Semana del lunes **20 de Noviembre**.
  **Coloquio** : Semana del lunes **27 de Noviembre**.
- **1er recuperatorio** : Semana del lunes **4 de Diciembre**.
  **Coloquio** : Semana del lunes **11 de Diciembre**.
- **2do recuperatorio** : Semana del **lunes 19 de Febrero**.
  **Coloquio** : Semana del lunes **26 de Febrero**.
- En cada curso se determinan las fechas y horas exactas.

## *Responsabilidades asociadas a la Presentación*

Todo trabajo que los equipos presenten a la Cátedra deberá ser realizado en su totalidad y exclusivamente por los integrantes del equipo. Sin bien existe total libertad para discutir y consultar con los docentes y/o compañeros las distintas soluciones que se podrán implementar, los códigos fuentes y la documentación deberán ser el fruto del trabajo del equipo, resultando en sanción la copia de códigos, documentación y/o cualquier otra parte del Trabajo Practico.

- La entrega por parte de un equipo de un TP que es una copia parcial o total de otro de TP de otro equipo, es considerada una falta gravísima, quedando desaprobados los integrantes de ambos equipos, debiendo recursar la materia y notificando a las autoridades de nuestro Departamento.
- La entrega de código fuente que no compile ANSI C o que compile con *warnings* no identificados con comentarios en el propio código fuente provocará que el TP deba ser recuperado, utilizando para ello una de las fechas previstas.—

# Sintaxis y Semántica de Lenguajes 2006.2

# TP – Sistema Bancario – Fecha

*por José María Sola & Adriana Adamoli*
20061016

## *Introducción*

El TAD Fecha otorga a los programadores la abstracción del concepto fecha, de uso frecuente en las aplicaciones bancarias desarrolladas por la empresa.

## *Valores*

Un valor del TAD Fecha representa el concepto de fecha de uso cotidiano. Se representa por un día del mes, un mes y un año; pero también es representable mediante un día del año y un año, ya que el mes está en función del día del año y del año. Un valor de este TAD *no* puede representar una fecha AC.

Implementar el conjunto de valores del TAD Fecha en algunas de las siguientes formas:

(**Punto 1.1**) con el tipo de dato int.
(**Punto 1.2**) con un struct con dos miembros char y un tercero int.
(**Punto 1.3**) con un struct con un miembro que represente el día del año y otro para el año.
(**Punto 1.4**) con un struct con tres miembros campos de bits. Leer **[K&R1988] "Chapter 6 – Structures – 6.9 Bit-fields"**.
(**Punto 1.5**) con un struct con dos miembros campos de bits. Leer **[K&R1988] "Chapter 6 – Structures – 6.9 Bit-fields"**.

En el curso se define para cada equipo que implementación debe construir.

### Representaciones literales de un valor del TAD Fecha

Un valor del TAD Fecha puede escribirse en formato corto y en formato largo. Ejemplos de formato corto: *01/01/1986*, *29/02/2000*, *05/08/2759* y *21/10/0078*. Ejemplos de formato largo: *1ero de Enero de 1986*, *29 de Febrero de 2000*, *5 de Agosto de 2759* y *21 de Octubre de 78*. Notar el uso del cero delante del día y del mes en el formato corto. El año en formato corto tiene como mínimo cuatro dígitos y se completa con ceros delante, en formato largo no. Notar la representación del primer día de cada mes para el formato largo. El conjunto de los formatos de fecha se define por extensión como: FormatoDeFecha = {FormatoDeFechaCorto, FormatoDeFechaLargo}

(**Punto 1.6**) Escribir la **ER** que representa el **LR** de las fechas en formato corto.
(**Punto 1.7**) Definir formalmente la **GIC** que genera el **LR** de las fechas en formato largo.
(**Punto 1.8**) Implementar el conjunto FormatoDeFecha con enum y typedef.

## *Operaciones*

### Operaciones de Creación

1. **Crear** : Día × Mes × Año → Fecha. Crear una fecha a partir de día, mes y año.
2. **CrearDesdeDíaDelAño** : Día × Año → Fecha. Crear una fecha dado un día del año y el año. Ejemplos: *1* y *2005* es *01/01/2005*, *365* y *2005* es *31/12/2005*, y *366* y *2000* es *31/12/2006*. Considerar años bisiesto.
3. **CrearDesdeCadena** : Cadena × FormatoDeFecha → Fecha. Crear una fecha a partir de una cadena y un indicador del formato de la cadena.

(**Punto 1.9**) **Especificar** las **operaciones de creación** de **dos maneras**: **con precondiciones** y **sin precondiciones**.
(**Punto 1.10**) **Implementar** las **operaciones de creación con precondiciones**.

### Operaciones de Proyección (Getters)

4. **GetDía** : Fecha → Día
5. **GetMes** : Fecha → Mes
6. **GetAño**: Fecha → Año
7. **GetDíaDelAño** : Fecha → DíaDelAño

## Otras Operaciones

8. **Comparar**. Análoga a strcmp.

9. **GetFechaMedia**. Ejemplo: *01/01/1986* y *03/01/1986* produce *02/01/1986*.

10. **GetComoCadena** : Fecha× FormatoDeFecha → Cadena. Operación complementaria a **CrearDesdeCadena**.

11. **IsBisiesto**. Ejemplos: para *21/12/1900* retorna falso, para *01/09/2000* retorna verdadero.
(**Punto 1.11**) Desarrollar función privada static int IsBisiesto(int unAnio).

## Requerimientos para la implementación

En la implementación utilizar las siguientes declaraciones de variables externas y de funciones privadas:

```
static char *losNombresDeLosMeses[]={
    "","Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio", "Julio",
    "Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre"
};

static int losDiasDeLosMeses[2][13]={
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

static int GetDiaDelAnio(int unDia, int unMes, int unAnio);
static void GetMesDia(int unAnio, int unDiaDelAnio, int *elMes, int *elDia);
```

(**Punto 1.12**) Para aplicar las anteriores declaraciones leer de **[K&R1988]**:
**"Chapter 5 - Pointers and Arrays – 5.8 Initialization of Pointer Arrays"** y
**"Chapter 5 - Pointers and Arrays – 5.7 Multi-dimensional Arrays"**.

## Operaciones de Persistencia

**Binario**

12i. **Read** : Flujo → Fecha × Flujo 12o. **Write** : Fecha × Flujo → Flujo
(**Punto 1.12**) Diseñar una secuencia de datos que represente eficientemente una Fecha en un flujo binario. Explicar ese diseño.

**Texto**

13i. **Scan** : Flujo → Fecha × Flujo 13o. **Print** : Fecha × Flujo → Flujo
La secuencia de datos en un flujo de texto que representa una fecha es similar a la de las cadenas con formato corto, con el agregado de que la fecha finaliza con *numeral* (#): **dd/mm/aaaa#**

## Prototipos de algunas de las funciones que implementan las operaciones

```
 1 . Fecha Fecha_Crear(int unDia, int unMes, int unAnio);
 3 . Fecha Fecha_CrearDesdeCadena(const char unaCadena[], FormatoDeFecha unFormato);
 7 . int Fecha_GetDiaDelAnio(Fecha unaFecha);
 8 . int Fecha_Comparar(Fecha unaFecha, Fecha otraFecha);
10 . void Fecha_GetComoCadena(
         Fecha unaFecha,
         char unaCadena[],
         FormatoDeFecha unFormato
     );
11 .  int Fecha_IsBisiesto(Fecha unaFecha);
12i. Fecha Fecha_Read(FILE *in);
13o. void Fecha_Print(FILE *out, Fecha unaFecha);
```

(**Punto 1.13**) Diseñar las sentencias que permitan leer todas las fechas disponibles en un flujo, ya sea con Read o Scan, y determinen si el fin de la lectura fue por error o no ¿Cómo escribir la condición del ciclo para que cumpla con el *idioma* presentado en la sección **1.5.1 de [K&R1988]** para la lectura? Ayuda: operador coma (,). ¿Qué condición se debe evaluar a la salida del ciclo?—

# Sintaxis y Semántica de Lenguajes 2006.2

# TP – Sistema Bancario – Transacción

*por José María Sola & Adriana Adamoli*
20061029

## *Introducción*

El TAD Transacción permite a los programadores de las aplicaciones desarrolladas por la empresa la abstracción del concepto transacción bancaria, es decir, movimiento en una cuenta.

## *Valores*

Un valor del TAD Transacción tiene un ***código***, ocurre en una ***fecha***, se realiza por un ***medio***, tiene un ***concepto*** y es por un ***monto***.

El ***código*** es una cadena de 10 dígitos hexadecimales; la ***fecha*** se representa con un valor del ***TAD Fecha***; los posibles ***medios*** de una transacción son *EnLinea*, *Ventanilla* o *Cajero*; el ***concepto*** es una cadena no vacía y de longitud menor o igual *a 40*; el ***monto*** es un número racional muy grande que puede ser representado en notación decimal punto fijo con dos dígitos en la parte fraccionaria. Los créditos se representan con montos positivos y los débitos con montos negativos.

(**Punto 2.1**) En la especificación, **describir formalmente** el lenguaje de los **códigos** y de los **conceptos**.
(**Punto 2.2**) En la especificación, **describir matemáticamente** el conjunto al cual pertenece **monto**.
(**Punto 2.3**) **Implementar** el conjunto **Medio** con enum y typedef.

En la implementación, un valor del TAD Transacción está soportado por las siguientes declaraciones:

```
#define LONGITUD_MAXIMA_CONCEPTO 40
#define LONGITUD_CODIGO          10

typedef struct {
    char     elCodigo[LONGITUD_CODIGO];
    Fecha    laFecha;
    Medio    elMedio;
    char     elConcepto[LONGITUD_MAXIMA_CONCEPTO+1];
    long int elMonto;
} Transaccion;
```

(**Punto 2.4**) Buscar la forma de implementar el monto (que es valor racional, pertenece al conjunto **Q**) en un objeto del tipo long int (que es un valor entero, pertenece al conjunto **Z**).

## *Operaciones*

1. **Crear** : Código × Fecha × Medio × Concepto × Monto → Transacción.
(**Punto 2.5**) **Especificar** la operación de creación de **dos maneras**: **con precondiciones** y **sin precondiciones**.
(**Punto 2.6**) **Implementar** la operación de creación **con precondiciones**.

2. **Consolidar** : Transacción × Transacción × Código → Transacción.
Dadas dos transacciones crea una nueva que las consolida. La fecha es la de la mayor, el monto es la suma y el medio es *EnLinea*.

## Operaciones de Proyección y Mutación (Getters y Setters)

(**Punto 2.7**) **Especificar** e **implementar con precondiciones**.

| | |
|---|---|
| 3g. **GetCodigo** | 3s. **SetCodigo** |
| 4g. **GetFecha** | 4s. **SetFecha** |
| 5g. **GetMedio** | 5s. **SetMedio** |
| 6g. **GetConcepto** | 6s. **SetConcepto** |
| 7g. **GetMonto** | 7s. **SetMonto** |

(**Punto 2.8**) Durante la **implementación**, tener en cuenta que para el usuario del TAD, el **código**(3) y el **concepto**(6) son cadenas que siguen el modelo de **cadenas de ANSI C**, más allá de cual sea la **representación interna**.
(**Punto 2.9**) Durante la **implementación**, tener en cuenta que para el usuario del TAD, el **monto** (7) es un valor `float`, más allá de cual sea la **representación interna**.

## Operaciones de Persistencia

**Binario**
8i **Read** 8o. **Write**
La secuencia de datos que forma una transacción en un flujo binario es la siguiente:

| Código | Fecha | Medio | Longitud del concepto | Concepto | Monto |
|---|---|---|---|---|---|

De esta forma las transacciones se persisten en flujos binarios como registros de longitud variable. Dada la declaración `Transaccion unaTransaccion;` entonces las siguientes expresiones definen la longitud en bytes de cada campo.

- Código: `sizeof unaTransaccion.elCodigo`
- Fecha: definido por la implementación de la operación **Write** del TAD Fecha.
- Medio: `sizeof unaTransaccion.elMedio`
- Longitud del concepto: `sizeof(sizet_t)`
- Concepto: la cantidad de bytes establecida en el valor del campo anterior, definida por el valor de la expresión `strlen(unaTransaccion.elConcepto)`
- Monto: `sizeof unaTransaccion.elMonto`

**Texto**
9i. **Scan** 9o. **Print**
La secuencia de datos en un flujo de texto que representa una transacción es la siguiente:
**`{código, fecha, medio, concepto, monto}\n`**
Notar el espacio luego de cada coma.

(**Punto 2.10**) En la especificación, **describir formalmente** el **lenguaje** de las **transacciones** en **flujos de texto**.
(**Punto 2.11**) En la implementación, invocar a **fscanf** y **fprintf**.

## Prototipos de algunas de las funciones que implementan las operaciones

(**Punto 2.12**) ¿Por qué el calificador **const** es necesario en **algunos de los parámetros y en otros no**.
(**Punto 2.13**) Calificar a los parámetros con **const cada vez que sea necesario**.
(**Punto 2.14**) ¿Por qué el **símbolo *** se utiliza en **algunos de los parámetros y en otros no**?
(**Punto 2.15**) Utilizar el **símbolo * cada vez que sea conveniente**.

```
 1 . Transaccion Transaccion_Crear(
         const char  unCodigo[],
         Fecha       unaFecha,
         Medio       unMedio,
         const char  unConcepto[],
         float       unMonto
     );
 3g. void Transaccion_GetCodigo(
         const Transaccion *unaTransaccion,
         char elCodigo[]
     );
 6s. void Transaccion_SetConcepto(
         Transaccion *unaTransaccion,
         const char elConcepto[]
     );
 8i. Transaccion Transaccion_Read (FILE *in);
 9o. void        Transaccion_Print(FILE *out, const Transaccion *unaTransaccion);
```

# Sintaxis y Semántica de Lenguajes 2006.2

# TP – Sistema Bancario – Cuenta

*por José María Sola & Adriana Adamoli*
20061106

## Introducción

El TAD Cuenta permite a los programadores de las aplicaciones desarrolladas por la empresa la abstracción del concepto cuenta bancaria de un cliente. Una cuenta registra sus transacciones en el orden en que ocurren.

## Valores

Un valor del TAD Cuenta tiene un ***código de cuenta***, es de un ***cliente***, fue ***abierta*** en una fecha y registra ***transacciones***. La cuenta nula se presenta por la 0-upla: ( ).

El ***código de cuenta*** es una cadena formada por un *identificador* ANSI C seguido de una *constante entera octal* ANSI C. El ***cliente*** se representa por una cadena no nula, la ***apertura*** se representa con un valor del ***TAD Fecha***, las ***transacciones*** es una secuencia de valores del ***TAD Transacción*** en el orden natural que fueron registradas.

(**Punto 3.1**) Definir formalmente la **GIC** que genera el **LR** de los **códigos de cuenta**.
(**Punto 3.2**) Definir formalmente el **AFD** mínimo que reconoce el **LR** de los **códigos de cuenta**.

En la implementación, un valor del TAD Cuenta está soportado por la siguiente declaración:

```
typedef struct {
    const char  *elCodigoDeCuenta;
    const char  *elCliente;
    Fecha        laApertura;
    size_t       laCantidadDeTransacciones;
    Transaccion *lasTransacciones;
} Cuenta;
```

El miembro `lasTransacciones` es un puntero al comienzo de un arreglo de elementos del tipo `Transaccion`. La memoria necesaria para el arreglo varía en función de los elementos que se van agregando y removiendo, se deberá utilizar las funciones `calloc` ó `malloc` y `realloc`. Sea `Cuenta *unaCuenta;` una variable correctamente inicializada, la siguiente expresión accede al último elemento del arreglo:

```
unaCuenta->lasTransacciones[ unaCuenta->laCantidadDeTransacciones - 1 ]
```

(**Punto 3.3**) Presentar mediante un gráfico la estructura que implementa este tipo de dato (si el equipo los conoce, puede valerse de los *diagramas de clase* y *diagramas de objetos* de *UML 2.0* http://www.omg.org).

## Operaciones

(**Punto 3.4**) **Especificar** e **implementar** **sin precondiciones** las siguientes operaciones:
1a **Crear**, 4s **SetCliente**, 5c **Acreditar**, 5d **Debitar** y 5f **RemoveTransacciónPorÍndice**.

1a. **Crear** : Cadena × Cadena × Fecha → Cuenta.
(**Punto 3.5**) **Construir** la siguiente **función privada** que indica si la cadena dato es un **código de cuenta**:

```
static int IsCodigoDeCuenta(const char *string);
```

Esta función es un predicado que define el lenguaje (conjunto) de los códigos de cuenta (i.e. dada una cadena retorna verdadero si es una palabra que pertenece al lenguaje de los códigos de cuenta). La función implementa un autómata finito reconocedor. Ver punto 3.2. Leer de **[MUCH2000]**:
**"Capítulo 11 – Aplicación I: validación de cadenas"**
**"Capítulo 12 – Aplicación II: Analizadores Léxicos"**
(**Punto 3.6**) Implementar está operación con el concepto de "*semántica de referencia*": la función no retorna una cuenta, sino, una referencia (puntero) a una cuenta. Invocar a `malloc` para reservar espacio para la estructura cuenta, para el código de cuenta y para el cliente. Contrastar con el concepto de "*semántica de valor*" utilizado en los anteriores dos TAD. **Responder**: ¿cuándo y qué funciones deben invocar a `free`?
(**Punto 3.7**) Para aplicar este diseño leer de **[K&R1988]**:
**"Chapter 6 - Structures – 6.5 Self-referential Structures"**,
**"Chapter 6 - Structures – 6.6 Table Look-Up"** y
**"Chapter 6 - Structures – 6.7 Typedef"**.

## Operaciones de Proyección y Mutación (Getters y Setters)

2g. **GetCodigoDeCuenta**
3g. **GetFecha**
4g. **GetCliente** 4s. **SetCliente.**

## Operaciones sobre las Transacciones

5a. **GetCantidadDeTransacciones** : Cuenta → $\mathbf{N_0}$

5b. **AddTransacción** : Cuenta × Transacción → Cuenta

5c. **Acreditar** : Cuenta × Código × Fecha × Medio × Concepto × Crédito → Cuenta
**Acreditar** : Cuenta × Cadena × Fecha × Medio × Cadena × **Q** → Cuenta

5d. **Debitar** : Cuenta × Código × Fecha × Medio × Concepto × Débito → Cuenta
**Debitar** : Cuenta × Cadena × Fecha × Medio × Cadena × **Q** → Cuenta

5e. **RemoveTransacciónPorÍndice** : Cuenta × **Z** → Cuenta

5f. **GetTransacciónPorÍndice**

5g. **GetTransacciónPorCódigo**

Las operaciones 5c y 5d registran transacciones en la cuenta de una forma particular: crédito y débito, reciben un número mayor a cero mientras que la operación 5b lo hace de una más forma general. Las operaciones 5c y 5d se deben especificar sin precondiciones, por eso el código es pasado como una cadena, por ejemplo.

## Operaciones de Consulta de Montos

6a. **GetSaldo**
6b. **GetDébitosPorUnConcepto**
6c. **GetCréditosPorUnConcepto**
6d. **GetTotalDeDébitos**
6e. **GetTotalDeCréditos**

## Operaciones de Persistencia

**Binario**
7i **Read** 7o. **Write**
La secuencia de datos que forma una cuenta en un flujo binario es la siguiente:

| Longitud del código de cuenta | Código de cuenta | Longitud del cliente | Cliente | Fecha | Cantidad de Transacciones | Transacciones |
|---|---|---|---|---|---|---|

Dado este diseño, las transacciones se escriben en flujos binarios como registros de longitud variable, la variabilidad no está dada solo en el largo de algunos campos, sino también en la cantidad de campos que almacenan valores del TAD Transacción. Sea `Cuenta *unaCuenta;` una variable correctamente inicializada, entonces las siguientes expresiones definen la longitud en bytes de cada campo.

- Longitud del código de cuenta: `sizeof(size_t)`
- Código de cuenta: la cantidad de bytes establecida en el valor del campo anterior, definida por el valor de la expresión `strlen(unaCuenta->elCodigoDeCuenta)`
- Longitud del cliente: `sizeof(size_t)`
- Cliente: la cantidad de bytes establecida en el valor del campo anterior, definida por el valor de la expresión `strlen(unaCuenta->elCliente)`
- Fecha: definido por la implementación de la operación **Write** del TAD Fecha.
- Cantidad de transacciones: `sizeof unaCuenta->laCantidadDeTransacciones`
- Transacciones: definido por la implementación de la operación **Write** del TAD Transacción, multiplicado por `unaCuenta->laCantidadDeTransacciones`

**Texto**
8i. **Scan** 8o. **Print**
La secuencia de datos en un flujo de texto que representa una cuenta es la siguiente:
**{**\t*códigoDeCuenta*, *cliente*, *fecha*, *cantidadDeTransacciones*\n
\t**{**\n
\t\t*transacción*\n
\t\t*transacción*\n…
\t**}**\n
**}**\n
Notar el espacio luego de cada coma, los niveles de tabulación (indentado), llaves y nuevas líneas.
(**Punto 3.8**) En la implementación, invocar a **fscanf** y **fprintf**.

## Prototipos de algunas de las funciones que implementan las operaciones

(**Punto 3.9**) ¿Qué implica que la función de creación retorne un ***puntero* a una Cuenta**?
(**Punto 3.10**) ¿Debe especificarse la operación **Fecha_Destruir**? ¿Debe ser **pública** o **privada**?

```
1a. Cuenta *Cuenta_Crear(
        const char *unCodigoDeCuenta,
        const char *unCliente,
        Fecha       unaFechaDeApertura
    );
1b. void Cuenta_Destruir(Cuenta *unaCuenta);
2g. char *Cuenta_GetCodigoDeCuenta(
        const Cuenta *unaCuenta,
        char         *unCodigoDeCuenta
    );
4s. char *Cuenta_SetCliente(
        Cuenta     *unaCuenta,
        const char *unCliente
    );
5b. Cuenta *Cuenta_AddTransaccion(
        Cuenta            *unaCuenta,
        const Transaccion *unaTransaccion,
    );
5c. Cuenta *Cuenta_Acreditar(
        Cuenta     *unaCuenta,
        const char *unCodigo,
        Fecha       unaFecha,
        Medio       unMedio,
        const char *unConcepto,
        float       unMonto
    );
6b. double Cuenta_GetCreditosPorUnConcepto(
        const Cuenta *unaCuenta,
        const char   *unConcepto
    );
7i. Cuenta *Cuenta_Read (FILE *in);
8o. Cuenta *Cuenta_Print(FILE *out, const Cuenta *unaCuenta);
```

—

# Sintaxis y Semántica de Lenguajes 2006.2

# TP – Sistema Bancario – Banco

*por José María Sola & Adriana Adamoli*
20061112

## *Introducción*

El TAD Banco permite a los programadores de las aplicaciones desarrolladas por la empresa la abstracción del concepto de banco. Un banco gestiona las cuentas de sus clientes y las transacciones que en ellas se vuelcan.

## *Valores*

Un valor del Banco Cuenta tiene un ***nombre*** y gestiona ***cuentas***. El ***nombre*** es una cadena no nula, las ***cuentas*** es una secuencia de valores del ***TAD Cuenta*** con el ordenamiento dado por el orden en que cada cuenta fue abierta en el banco. En la implementación, un valor del TAD Banco está soportado por la siguiente declaración:

```
typedef struct {
    const char  *elNombre;
    size_t       laCantidadDeCuentas;
    Cuenta     **lasCuentas;
} Banco;
```

La implementación de la secuencia `lasCuentas` sigue el mismo modelo que ANSI C utiliza para los argumentos de línea de comando. La función main tiene un parámetro que indica la cantidad de argumentos y otro que es un puntero al comienzo de un arreglo de punteros a cadenas: `int main(int argc, char **argv)`.
(**Punto 4.1**) **Leer [K&R1988] "Chapter 5 – Pointers and Arrays – 5.10 Command Line Arguments"**. La secuencia de cuentas se implementa con un arreglo de punteros a `Cuentas`. El miembro `laCantidadDeCuentas` indica la longitud del arreglo, y `lasCuentas` es un puntero al primer elemento del arreglo. Los elementos del arreglo son del tipo de dato *puntero a Cuenta*, por lo que `lasCuentas` debe ser del tipo de dato *puntero a puntero a Cuenta*. Al implementar las operaciones que manipulan estos conjuntos, deberán utilizar `calloc` ó `malloc` y `realloc`. Sea `Banco *unBanco;` una variable correctamente inicializada, la siguiente expresión accede al último elemento del arreglo:

```
unBanco->lasCuentas[ unBanco->laCantidadDeCuentas - 1 ]
```

(**Punto 4.2**) Presentar mediante un gráfico la estructura que implementa este tipo de dato (si el equipo los conoce, puede valerse de los *diagramas de clase* y *diagramas de objetos* de *UML 2.0* http://www.omg.org).

## *Operaciones*

(**Punto 4.3**) **Especificar** e **implementar** **sin precondiciones** siempre que sea posible.

1. **Crear** : Cadena → Banco. Crea un banco sin cuentas, retorna la 0-upla si el nombre es incorrecto.

2. **GetNombre**.

3. **Transferir** : Banco × CódigoDeCuenta × Banco × CódigoDeCuenta × Fecha × MontoDeTransferencia → Banco × Banco × Lógico

***Transfiere*** un ***monto*** de una ***cuenta origen*** a otra ***cuenta destino***. Como resultado, la ***cuenta origen registra un débito*** y ***la destino un crédito*** por el monto transferido. El medio de las transacciones es ***EnLinea***; el ***concepto*** de la ***transacción*** en la ***origen*** es ***"Transferencia hacia: "*** *concatenado* con el ***código de la cuenta destino***; y ***el concepto*** de la ***transacción*** en la ***destino es "Transferencia desde: "*** concatenado con el ***código de la cuenta origen***. Las cuentas pueden pertenecer a un mismo banco. El ***valor lógico*** indica ***si la transferencia se pudo realizar*** ya que la cuenta destino o la de origen pueden ***no existir*** en los bancos origen y destino, el origen puede tener ***saldo insuficiente***, el ***monto de la transferencia*** es ***negativo***, etc.
(**Punto 4.4**) El monto de la transferencia (denotado anteriormente por el conjunto *Transferencia*) puede ser calculado a partir de una ***expresión matemática simple***. La expresión puede estar disponible en ***dos medios diferentes***: ***cadena*** o ***flujo de texto***, por lo que esta operación se va a ***especificar e implementar en dos versiones***: una que recibe una ***cadena con la expresión*** y otra que recibe un ***flujo de texto que permite la lectura de una expresión***.
3a. **TransferirDesdeCadena** : Banco × Cadena × Banco × Cadena × Fecha × **Cadena** → Banco × Banco × Lógico
3b. **TransferirDesdeFlujo** : Banco × Cadena × Banco × Cadena × Fecha × **Flujo** → Banco × Banco × Lógico

(**Punto 4.5**) Para implementar 3a y 3b se debe implementar un ***evaluador de expresiones matemáticas simples***, desarrollado con un ***autómata reconocedor-accionador***. Estas expresiones matemáticas tienen como operandos los números enteros en base decimal no negativos y como operadores los cuatro básicos (**+**, **-**, ***** y **/**). La precedencia es la normal, es decir, ***** y **/** tienen mayor precedencia que **+** y **-**, los paréntesis no forman parte del lenguaje, no es necesario que el evaluador rechace las expresiones con divisiones por cero.

(**Punto 4.6**) Definir formalmente la **GIC** que genera el **LR** de las **expresiones matemáticas simples**.
(**Punto 4.7**) Definir formalmente el **AFD** que reconoce el **LR** de las **expresiones matemáticas simples** y **dibujar su dígrafo**.

(**Punto 4.8**) Las implementaciones de ***TransferirDesdeCadena*** y ***TransferirDesdeFlujo*** deben invocar a:

```
static int EvaluarExpresionEnCadena(const char *in, float *elResultado);
static int EvaluarExpresionEnFlujo (FILE       *in, float *elResultado);
```

Las funciones operan como un autómata, analizan la expresión **carácter a carácter**. La primera función toma los caracteres **uno a uno desde una cadena**, mientras que la segunda hace lo mismo pero **desde un flujo de texto**. Es recomendable que estas funciones llamen, a su vez, a otras funciones que implementen el **comportamiento que es común** a las evaluaciones desde ambos orígenes. La evaluación de una expresión **desde un flujo de texto finaliza cuando**: 1) se encuentra un carácter del alfabeto pero en una **posición incorrecta** (e.g. `"12+/2"`), 2) se encuentra un **carácter espúreo** (e.g. `"51+A"` ó `"51+3#"` ó `"51+3\n"`), ó 3) **no es posible leer del flujo** (i.e. `(c=fgetc(in))==EOF`). Ambas funciones retornan 0 cuando la expresión es inválida. Ejemplos:

```
float valor;
assert(EvaluarExpresionEnCadena("10/2+3", &valor));
assert(valor==8);
```

Leer de **[MUCH2000]**: **"Capítulo 11 – Aplicación I: validación de cadenas"** y **"Capítulo 12 – Aplicación II: Analizadores Léxicos"**

## Operaciones sobre las Cuentas

4a. **GetCantidadDeCuentas**
4b. **AddCuenta**. Abre una cuenta en el banco: agrega la cuenta dato al banco.
4c. **RemoveCuentaPorCódigo**. Dado un código de cuenta cierra la cuenta asociada: remueve la cuenta del banco.
4d. **GetCuentaPorÍndice.**
4e. **GetCuentaPorCódigo**.

## Operaciones de Consulta

5a. **GetCuentaConMayorSaldo**. Si no es única, retorna la 0-upla.
5b. **GetTotalDeSaldos**. Sumatoria de los saldos de cada cuenta del banco.
5c. **GetPromedioDeSaldos**.

## Operaciones de Persistencia

**Binario** 6i **Read** 6o. **Write** Un banco se represente en un flujo binario con secuencias de la forma:

| Longitud del nombre del banco | Nombre del banco | Cantidad de Cuentas | Cuentas |
|---|---|---|---|

**Texto** 7i. **Scan** 7o. **Print** Un banco se represente en un flujo de texto con secuencias de la forma:

```
{\tnombreDelBanco, cantidadDeCuentas\n
\t{\n
\t\tcuenta\n
\t\tcuenta\n…
\t}\n
}\n
```

Notar el espacio luego de cada coma, los niveles de tabulación (indentado), llaves y nuevas líneas.
(**Punto 4.7**) En la implementación, invocar a **fscanf** y **fprintf**.

## Prototipos de algunas de las funciones que implementan las operaciones

```
 1a. Banco *Banco_Crear(const char *unNombre);
 1b. Banco *Banco_Destruir(Banco *unBanco);
 2g. char  *Banco_GetNombre(const Banco *unBanco);
 3a. int    Banco_TransferirDesdeCadena(
         Banco      *unBancoOrigen,  const char *unaCuentaOrigen,
         Banco      *unBancoDestino, const char *unaCuentaDestino,
         Fecha       unaFecha,
         const char *unMonto
     );
 3b. int    Banco_TransferirDesdeFlujo(
         Banco      *unBancoOrigen,  const char *unaCuentaOrigen,
         Banco      *unBancoDestino, const char *unaCuentaDestino,
         Fecha       unaFecha,
         FILE       *in
     );
 6i. Banco *Banco_Read (FILE *in);
 7o. Banco *Banco_Print(FILE *out, const Banco *unBanco);—
```