Sintaxis y Semántica de Lenguajes 2006.1

TP – Sistema de Punto de Venta – Presentación

20060422

*Por **Ing. José María Sola***

*Con la colaboración de **Profesora Adriana Adamoli**, **Tomás Colombo**, **Profesora Marta Ferrari**, **Pablo Ignacio Krichevsky**, **Mariano de María** & **Facundo Merighi***

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 de Punto de Venta (POS, Point of Sale)*. Las abstracciones son *Producto*, *Ticket* y *Sesión de Venta* (o simplemente *Sesión*). 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.

Contexto del Problema

La empresa a la cual usted pertenece se especializa en el desarrollo de sistemas para organizaciones comerciales. Últimamente, la empresa notó, en el mercado, un crecimiento de la demanda de proyectos para clientes que poseen muchos puntos de venta (POS, Point of Sale), como por ejemplo supermercados y restaurantes de comidas rápidas. Ante esta demanda, la empresa decide invertir en el desarrollo de un *framework* que maximice la productividad de los desarrolladores de aplicación, minimice los tiempos de desarrollo, facilite el agregado de futuras funcionalidades, 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.

Su tarea es diseñar tres TADs que formarán parte de este framework. Estos tres TADs serán probados de forma completa y se ejemplificará su uso en un programa de aplicación que simule las ventas que se producen en un POS.

Introducción

Los programadores que hagan uso de estas abstracciones construirán aplicaciones que permitan tareas como: definir un producto, vender productos a un cliente registrando la venta en tickets, cancelar ventas, registrar tickets a una sesión de un cajero, consultar estadísticas sobre los tickets de una sesión, persistir la información (a disco, por ejemplo) y luego recuperarla entre otras.

Para eso, se proveerá al programador, de los tres tipos de datos abstractos; implementados en ANSI C y *encapsulados* en tres bibliotecas. El programador que *utilizará* los TADs recibirá la especificación, los tres archivos encabezados y las tres bibliotecas.

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.

Existen varios **roles**, similares pero diferenciables, para los especialistas involucrados en el diseño y utilización de los TADs:

- Especificador del TAD.
- Programador implementador del TAD.
- Especificador de las pruebas que valida las implementaciones del TAD.
- Programador de las pruebas que validan las implementaciones del TAD.
- Verificador de las corridas de los programas de prueba.
- Programador de las aplicaciones que utilizan el TAD.
- Usuario de las aplicaciones.

Breve Descripción de los TADs

Esta sección presenta el TP en forma general, le seguirá una sección con guías y normativas para la especificación e implementación y tres secciones más detallando los requerimientos de cada TDA.

La ***Sesión*** representa el periodo en el cual un cajero estuvo utilizando un POS; principalmente contiene todos los ***Tickets*** emitidos durante la *Sesión*. Cada *Ticket*, en esencia, es una venta de cierta cantidad de ***Productos***, un *Producto* es un ítem con un valor asociado.

El TAD *Producto* permite operaciones como: creación, determinación de precio e IVA, etc. *Ticket* permite operaciones como: creación, registro de venta de producto, vender productos genéricos, impresión de ticket, consultar sus totales, etc. *Sesión* permite registrar el cajero de la sesión, registrar tickets, consultar estadísticas de ventas, etc. Los tres TADs permiten almacenamiento en memoria externa (i.e. persistencia).

Nombres de los archivos involucrados en este TP

- Producto.c, Producto.h, Producto.lib, ProductoAplicacion.c y ProductoAplicacion.exe
- Ticket.c, Ticket.h, Ticket.lib, TicketAplicacion.c y TicketAplicacion.exe.
- Sesion.c, Sesion.h, Sesion.lib, SesionAplicacion.c y SesionAplicacion.exe.

Secuencia de Entregas

1. TAD Producto.
2. TAD Ticket.
3. TAD Sesión.
4. Programa de Aplicación. —

Sintaxis y Semántica de Lenguajes 2005.2

Normativas y Guías

20060422

Por Ing. ***José María Sola***

Con la colaboración de ***Pablo Ignacio Krichevsky***

Guía para la Especificación

Guía para la estructuración de la especificación de los valores

Descripción, n-upla, descripción de cada miembro de la n-upla, tipo de dato al que pertenece cada miembro de la n-upla, restricciones.

Guía para la estructuración de la especificación de las operaciones

La especificación de las operaciones tienen dos partes: Encabezado y Cuerpo.

Encabezado

- $f: A \times B \times C \rightarrow D \times E$
  Título de la operación en la forma de una función matemática.
- Clasificación de la operación.
- Breve descripción.

Cuerpo

- $f: M \times K \times K \rightarrow S \times M$
  Refinación de los conjuntos.
- $f(m_1, k_1, k_2) = (s, m_2)$
  Identificación de los datos y resultados. Notar que son elementos de conjuntos.
- Precondiciones y Poscondiciones.
- Dominio e Imagen.
- Semántica descripta en lenguaje matemático con la ayuda de axiomas o de lenguaje natural.
- Ejemplo(s).

Normativas para las Implementaciones

Esta sección presenta normas para la codificación en ANSI C que tienen por objeto facilitar el entendimiento y modificación de programas. Su gran mayoría son aplicables a distintos lenguajes de programación y no se restringen solo a C.

Indentación (Sangría)

De **[K&R1988] "Chapter 1 – A Tutorial Introduction – 1.2 Variables and Arithmetic Expressions"**
<< El cuerpo de un while puede ser unas o más sentencias incluidas en llaves, como en el convertidor de temperatura, o una sola declaración sin las llaves, como en:

```
while (i < j)
    i = 2 * i;
```

En cualquier caso, siempre indentaremos las sentencias controladas por el while con un tabulado (que hemos mostrado como cuatro espacios) así podemos ver de un vistazo qué sentencias están dentro del ciclo. La indentación acentúa la estructura lógica del programa Aunque a los compiladores de C no les interesa la apariencia de un programa, las indentaciones y espaciados correctos son críticos para hacer los programas fáciles de leer. Recomendamos escribir solo una sentencia por línea, y usar espacios en blanco alrededor de operadores para clarificar la agrupación. La posición de las lleves es menos importante, aunque hay personas que mantienen creencias apasionadas sobre la ubicación de las mismas. Hemos elegido uno de varios estilos populares. Escoja un estilo que le quede bien, y luego úselo de manera consistente. >>

```
/* Correcto */
unsigned long int Factorial(unsigned int n){
→      if( n <= 1 )
→      →      return 1;
→      return n * Factorial( n – 1 );
}

/* Incorrecto */
unsigned long int Factorial(unsigned int n)
{
if(n<=1)
return 1; return n * Factorial(n-1);}
```

Estilos de Indentación
A continuación se muestran diferentes estilos, la elección es puramente subjetiva, pero su aplicación debe ser consistente.

Estilo K&R – También conocido como "The One True Brace Style"
Secciones principales de Java usan este estilo, que era el recomendado anteriormente. Es un estilo recomendable, es el que usa la Cátedra.

```
while( SeaVerdad() ) {
    HacerUnaCosa();
    HacerOtraCosa();
}
HacerUnaUltimaCosaMas();
```

Estilo BSD/Allman
Microsoft Visual Studio 2005 impone este estilo por defecto. Nuevas secciones de Java usan este estilo. Es un estilo recomendable.

```
while( SeaVerdad() )
{
    HacerUnaCosa();
    HacerOtraCosa();
}
HacerUnaUltimaCosaMas();
```

Estilo Whitesmiths

```
while( SeaVerdad() )
    {
    HacerUnaCosa();
    HacerOtraCosa();
    }
HacerUnaUltimaCosaMas();
```

Estilo GNU

```
while( SeaVerdad() )
  {
    HacerUnaCosa();
    HacerOtraCosa();
  }
HacerUnaUltimaCosaMas();
```

Estilo Pico

```
while( SeaVerdad()
{   HacerUnaCosa();
    HacerOtraCosa(); }
HacerUnaUltimaCosaMas();
```

Estilo Banner

```
while( SeaVerdad() ) {
    HacerUnaCosa();
    HacerOtraCosa();
    }
HacerUnaUltimaCosaMas();
```

Constantes Punto Flotante

De **[K&R1988] "Chapter 1 – A Tutorial Introduction – 1.2 Variables and Arithmetic Expressions"**
<< … escribir constantes de punto flotante con puntos decimales explícitos inclusive cuando son valores enteros enfatiza su naturaleza de punto flotante a los lectores humanos. **>>**

Variables Innecesarias

De **[K&R1988] "Chapter 6 – A Tutorial Introduction – 1.3 The For Statement"**
<<… en cualquier contexto donde se permite usar el valor de algún tipo, se pude usar una expresión más complicada de ese tipo. **>>**
Esto evita el uso de variables innecesarias

```
/* Suponiendo una variable int a */
/* Correcto */
printf("Cociente: %d\nResto: %d\n", a / 2, a % 2);

/* Suponiendo una variable int a */
/* Incorrecto */
int c = a / 2;
int r = a % 2;
printf("Cociente: %d\nResto: %d\n", c, r);
```

No Usar Números Mágicos

De **[K&R1988] "Chapter 6 – A Tutorial Introduction – 1.4 Symbolic Constants"**
<< Es mala práctica enterrar en un programa "números mágicos" como 300 y 20; contienen poca información para alguien que pueda tener que leer el programa más adelante, y son difíciles de cambiar de una manera sistemática [durante el mantenimiento].

Una forma a tratar el tema de los números mágicos es darles nombres significativos. Una línea *#define* define a un nombre simbólico o constante simbólica como una cadena particular de caracteres:

`#define` <*nombre*> <*texto de reemplazo*>

Después de eso, cualquier ocurrencia del nombre (pero no entre comillas ni como parte de otro nombre) será substituida por el texto de reemplazo correspondiente. El nombre tiene la misma forma que un nombre de variable: una secuencia de letras y de dígitos que comienza con una letra. El texto de reemplazo puede ser cualquier secuencia de caracteres; no se limita a los números.**>>**

No Usar Sentencias `goto`

De **[K&R1988] "Chapter 3 – Control Flow – 3.8 Goto and labels"**
<< Formalmente, la sentencia `goto` nunca es necesaria, y en la práctica es casi siempre fácil escribir código sin ella…**>>**
<<…Sin embargo, hay algunas situaciones donde los `goto`s pueden encontrar su lugar. El más común es abandonar el proceso dentro de alguna de la estructura profundamente anidada, por ejemplo salir de dos o ciclos inmediatamente. La sentencia `break` no puede ser usada directamente ya que solo sale del ciclo más interno…**>>**
<< …El código que involucre un `goto` puede siempre ser escrito sin él, aunque quizás al precio de algunas pruebas repetidas o una variable extra…**>>**
<< … Con algunas excepciones como las citadas aquí, un código que utilice sentencias `goto` es en general más difícil de comprender y de mantener que un código sin `goto`s. Aunque no somos dogmáticos sobre el tema, sí parece que las sentencias goto se deben utilizar en raras oportunidades, o quizás nunca…**>>**

De **[K&R1988] "Chapter 5 – Pointers and Arrays"**
<<…Los punteros, junto con la sentencia goto, han sido duramente castigados y son conocidos como una manera maravillosa de crear programas imposibles de entender. Pero esto es verdad solo cuando son utilizados negligentemente… **>>**

Variables Externas

Las variables externas son llamadas así porque son definidas fuera de cualquier función. Son también conocidas como variables globales por que son globalmente accesibles.

De **[K&R1988] "Chapter 5 – Pointers and Arrays"**
<< Basarse demasiado en variables externas es peligroso puesto que conduce a programas con conexiones de datos que no son del todo obvias –las variables se pueden cambiar en maneras inesperadas e incluso inadvertidas– y el programa se torna difícil de modificar. La segunda versión del programa "línea más larga" es inferior a la primera, en parte por estas razones, y en parte porque destruye la generalidad de dos útiles funciones al escribir dentro de ellas los nombres de las variables que manipulan. **>>**

Diseño del Proyecto – Inclusión de Archivos

Conozca la forma correcta de diseñar la distribución de archivos que forman parte de un proyecto. Utilice los archivos encabezados para incluir declaraciones de cualquier tipo salvo definiciones de funciones. Las definiciones de las funciones se escriben en un archivo fuente aparte, mientras que en el encabezado se ubican los prototipos las funciones públicas. No utilice directivas del tipo:

```
#include "mal.c"
```

Diseño de Funciones

De **[K&R1988] "Chapter 6 – A Tutorial Introduction – 1.7 Functions"**
<< Una función proporciona una manera conveniente de encapsular un cómputo, la cual puede entonces ser utilizada sin preocuparse de su implementación. Con funciones correctamente diseñadas, es posible ignorar *cómo* se hace un trabajo; saber *qué* es lo que se hace es suficiente. C hace el tema de funciones fácil, conveniente y eficiente; usted verá a menudo una función corta definida y llamada una única vez, solo porque clarifica una cierta sección del código.**>>**

Las funciones deben abstraer un proceso, acción, actividad o cálculo. Lo importante para quien invoca a la función es "que" hace pero no "como" lo hace. Para diseñar correctamente una función considerar los siguientes parámetros de diseño:

Ocultamiento de Información
Diseñe las funciones de tal manera que no expongan detalles de implementación. No debe ser conocido por quien invoca la función "como" es que la función realiza la tarea.

Alta Cohesión
Significa que una función debe realizar solo una tarea.
Supóngase que se esta escribiendo un programa que resuelve integrales,
la función `MostrarResultado` deberá solo imprimir un resultado, no calcularlo. En forma similar, la función encargada de calcular este resultado, `Integrar`, no deberá imprimirlo. Esto no descarta que exista una función que utilice a ambas funciones para resolver la integral y mostrar su resultado. Tampoco es correcto que dos funciones hagan una tarea a medias.

Bajo Acomplamiento
Implica que una función tiene poca dependencia con el resto del sistema, para poder utilizar una función debe ser suficiente con conocer su prototipo (su parte pública).
Un ejemplo de alto acoplamiento es una función que retorna un valor modificando una variable global; si el identificador de la variable global cambia, se debe cambiar la función.

Las funciones deben contener menos de 20 sentencias o declaraciones. El uso de los espacios horizontales y verticales debe ser tal que permita leerlo en un medio de 80 columnas.

Archivos Encabezado

El contenido de los archivos encabezado debe estar "rodeado" por las siguientes directivas al preprocesador:

```
#ifndef INCLUIR_TAD
#define INCLUIR_TAD

/* Contenido del archivo encabezado. */

#endif
```

Esto permite la compilación condicional, evitando incluir más de una vez el contenido del archivo encabezado. Reemplazar *`TAD`* por un nombre del TAD que se está incluyendo.

Convención de Nomenclatura para los Identificadores

Primero se presentan los diferentes estilos de "capitalización" y luego se explican las reglas que establecen el estilo a aplicar según la entidad que se identifica.

Estilos de "Capitalización"

PascalCase
Todas las palabras juntas, sin espacios. Todas las palabras comienzan con mayúsculas y siguen en minúsculas. Ejemplos:

```
EstaEsUnaFraseEnPascalCase
Pila
AgregarElemento
```

camelCase
Todas las palabras juntas, sin espacios. Todas las palabras comienzan con mayúsculas y siguen en minúsculas, excepto la primera que está en minúsculas. Ejemplos:

```
estaEsUnaFraseEnCamelCase
unaPila
laLongitud
```

UPPERCASE
Todas las palabras juntas, sin espacios. Todas las palabras en mayúsculas. En general, se separan las palabras con undescores. Ejemplos:

```
ESTAESUNAFRASEENUPPERCASE
ESTA_ES_OTRA
MAXIMO
```

Underscores (Guiones Bajos)

- No usar con `camelCase` ni con `PascalCase`,
- Sí usar con `UPPER_CASE`.
- No usar delante de identificadores.

Identificadores para diferentes elementos
La regla general es: usar *nombres significativos* para *identificadores significativos* y *nombres no significativos* para *identificadores no significativos*.
Los nombres significativos son en general largos y los no significativos cortos.
La misma regla se puede aplicar para identificadores de acceso global e identificadores de acceso local.
Así, una función que permite calcular un balance –tiene acceso global y es un concepto significativo– debe tener un identificador como `GetBalance`; mientras que una variable que se utiliza para iterar un arreglo –tiene acceso local y no es un concepto significativo– debe tener un identificador tan simple como `i`.

No usar Notación Húngara
La notación Húngara es una notación, que entre otras cosas, promueve la práctica de prefijar los identificadores para incluir *metadata* (datos sobre los datos) al identificador, como por ejemplo el tipo de dato de una variable.
Charles Simonyi desarrolló esta notación en Microsoft –llamada así por lo extraño de los identificadores resultantes y por la nacionalidad del autor– pero luego, la propia Microsoft desestimó su uso.
Una copia del paper original por *Charles Simonyi* se encuentra en http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs600/html/HungaNotat.asp
<< No use notación Húngara. Los nombres buenos describen semántica, no tipo de dato. >>
Este tipo de notación genera dependencia con el lenguaje y complica el mantenimiento de los programas. La funcionalidad de la Notación Húngara hoy la proveen los IDEs (Integrated Development Enviroments) modernos, que con solo posicionar el cursor sobre el identificador informa los atributos anunciados en su declaración. Por otro lado, las funciones (procedimientos, métodos) deben ser diseñadas de una forma tal que requieran pocas líneas de código, por lo que la declaración de las variables y parámetros se encuentran en un contexto acotado que facilita la ubicación de las declaraciones.
[Design Guidelines for Class - Naming Guidelines - Parameter Naming Guidelines] http://msdn.microsoft.com/library/en-us/cpgenref/html/cpconFieldUsageGuidelines.asp

Nombres de tipo (typedef)
En PascalCase, sustantivo.
`SistemaPlanetario`
`Planeta`
`NumeroAstronomico`.

Funciones y macros con parámetros
En PascalCase, deben comenzar con un verbo en infinitivo y en general son seguidas por un sustantivo o frase.

```
OrdenarArreglo()
SepararUsuarioDeDominio()
Planeta_SetNombre()
NumeroAstronomico_EsOverflow()
```

Funciones y macros con parámetros públicas de cada TAD
Los identificadores de las funciones públicas que implementan las operaciones se prefijan con el nombre del TAD seguido de un *underscore* ("_"). `SistemaPlanetario_`, `Planeta_` y `NumeroAstronomico_`.

Variables Locales y Parámetros
En camelCase, sustantivo, en plural para arreglos.

Variables Globales.
En PascalCase, sustantivo, en plural para arreglos.

Enumeraciones.
El nombre del `typedef` de la enumeración y sus elementos en PascalCase y en singular.

Constantes Simbólicas (#define).
En `UPPER_CASE_CON_UNDERSCORES`.
De **[K&R1988] "Chapter 6 – A Tutorial Introduction – 1.4 Symbolic Constants"**
<< Las constantes simbólicas se escriben por convención en mayúsculas para que puedan ser rápidamente distinguidas de los nombres de variables escritos en minúscula. >>

Cadenas
Implementadas sin tamaño máximo (malloc y free).

Funciones privadas
Definidas como `static`. No forman parte de la especificación.

Operaciones de Creación
Las operaciones de creación serán implementadas como funciones que retornan un puntero a un objeto del tipo del TAD. Deberán usar malloc para obtener memoria para ese objeto. Si no hay memoria disponible, retornarán `NULL`. Por ejemplo:

```
NumeroAstronomico *NumeroAstronomico_CrearDesdeCadena(
    const char *unaCedena
);
```

Operaciones de Destrucción
No forman parte de la especificación. Permiten liberar los recursos tomados durante la creación.

```
void NumeroAstronomico_Destruir(
    NumeroAstronomico *unNumeroAstronomico
);
```

Valores de los TADs como parámetros
Un valor de un TAD será pasado como parámetro mediante un puntero a ese valor (i.e. semántica de referencia). Si el parámetro es del tipo entrada/salida no tendrá el calificador `const`, si es de entrada sí. Por ejemplo:

```
SistemaPlanetario* SistemaPlanetario_AgregarPlaneta(
    SistemaPlanetario* unSistemaPlanetario, /*inout*/
    const Planeta*     unPlaneta               /*in*/
);

int NumeroAstronomico_EsOverflow(
    const NumeroAstronomico* unNumeroAstronomico /*in*/
);
```

Operaciones de Modificación (Mutación, Setter)

Estas operaciones seguirán el modelo impuesto por funciones como `strcat` de ANSI C.

La función `strcat` recibe dos parámetros: Primero, la cadena que será modificada concatenándole al final otra cadena, y segundo, la cadena que será concatenada al final de la primera. La función retorna el puntero a la primera cadena.

```
char *strcat(char *s1, const char *s2);
```

Este modelo permite evitar construcciones del tipo:

```
char s1[4+1]="ab", *s2="cd";
strcat(s1, s2);
puts(s1);
```

al ser reemplazadas por:

```
char s1[4+1]="ab", *s2="cd";
puts( strcat(s1, s2) );
```

Al `strcat` no ser diseñada como una función `void`, podemos utilizar el resultado de su invocación como argumento de `puts`.

Análogamente

```
Planeta* Planeta_SetNombre(Planeta* unPlaneta, const char* unNombre);
```

permite reemplazar:

```
Planeta* unPlaneta=Planeta_Crear(…);
Planeta_SetNombre(unPlaneta, "Krypton");
puts( Planeta_GetNombre(unPlaneta) );
```

por:

```
Planeta* unPlaneta=Planeta_Crear(…);
puts( Planeta_GetNombre( Planeta_SetNombre(unPlaneta, "Krypton") ) );
```

Como `Planeta_SetNombre` no fue diseñada como una función void, sino como una función que retorna un Planeta, su invocación ser utilizada como argumento de `Planeta_GetNombre`.

Nombres de archivos

- Cada TAD se implementará en una biblioteca. El código fuente de la implementación estará en un archivo *TAD*.c, la declaración de la parte pública en el archivo encabezado *TAD*.h, y se generará la biblioteca *TAD*.lib.
- El código fuente del programa de aplicación que prueba el TAD será *TADAplicacion*.c, y se generará *TADAplicacion*.exe.
- Sólo se entregarán los códigos fuentes: *TAD*.c, *TAD*.h y *TADAplicacion*.c.
- No se aceptarán *TAD*.lib y *TADAplicacion*.exe.

Guía de secuencia de actividades para la generación de los TADs

A continuación se presenta una guía de una posible secuencia de actividades para la construcción de TADs.

En base a una *correcta especificación*, se diseña un *correcto programa de prueba*, luego se implementa el TAD. Si la implementación realizada pasa el programa de prueba, la *implementación es correcta*.

Se debe considerar que *el proceso es en general iterativo*, en el sentido de que *la especificación siempre es la entrada para la implementación* y que debe estar *completamente definida*, pero que *hay veces que la implementación retroalimenta a la especificación para mejorarla* y volver a comenzar el proceso.

1. Comprensión del contexto y del problema que el TAD ayuda a solucionar.
2. Diseño de la Especificación.
3. Diseño de los casos de prueba a nivel especificación.
4. Implementación.

4.1. Diseño de prototipos.
4.2 Codificación de prototipos.
4.3 Diseño programa de prueba (aplicación)
4.4 Diseño y codificación de implementación de valores.
4.5 Codificación de funciones públicas y privadas (static).
4.6 Construcción de biblioteca.
4.7 Ejecución de programa de prueba.
4.8 ¿Hubo algún error? Entonces volver a 4.4

Normativas de Presentación

Para que la Cátedra acepte la presentación del trabajo por parte del equipo, se deben cumplir los requisitos de tiempo y forma. Si no se cumple alguno de los requisitos, la presentación no será considerada.

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 **tres grandes sub-secciones**:

  Nombre del TAD.
  1. Especificación. Especificación completa, extensa y sin ambigüedades de los valores y de las operaciones del TAD.
  2. Implementación. Biblioteca que implementa el TAD.
  2.1. **Listado** de código fuente del **archivo encabezado**, **parte pública**, *TAD*.h.
  2.2. **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.
  2.3. **Salidas**. Captura impresa de la salida del **proceso de traducción** (BCC32 y TLIB).
  3. Aplicación de Prueba
  3.1. **Código Fuente**. Listado del código fuente de la aplicación de prueba, *TAD*Aplicacion.c.
  3.2. Salidas
  3.2.1. Captura impresa de la salida del **proceso de traducción** (BCC32).
  3.2.2. Captura impresa de las salidas de la **aplicación de prueba**.
  3.2.3. Impresión de archivos de prueba de entrada y de archivos de salida generados durante la prueba.

El TP será acompañado por:

A. Copia Digitalizada. CD ó disquette (preferentemente CD) con copia de **solamente los 3 archivos de *código fuente* de cada TAD** (e.g.: *Producto.c, Producto.h, ProductoAplicacion.c, Ticket.c, Ticket.h, TicketAplicacion.c, Sesion.c, Sesion.h y SesionAplicacion.c*) **No se debe entregar ningún otro archivo**.
B. Formulario de Seguimiento de Equipo.

Tiempo

- En cada curso se diseñarán un **cronograma de entregas parciales** del TP, previas a la entrega final.
- La **entrega final** del TP será durante la **semana del lunes 12 de Junio, el día exacto se establece en cada curso**.
- **Primer recuperatorio**: Durante la **semana del lunes 26 de Junio, el día exacto se establece en cada curso**.
- **Segundo recuperatorio**: Durante la **semana del lunes 10 de Julio, el día exacto se establece en cada curso**.
- La entrega del TP se complementa con la **evaluación individual de cada integrante del equipo** en fecha y forma a determinar en cada curso

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 2006.1

TP – Sistema de Punto de Venta – TAD Producto

20060701

*Por **Ing. José María Sola***

*Con la colaboración de **Mariano de María***

Introducción

El TAD Producto permite, a programadores de las aplicaciones comerciales desarrollados por la empresa, la abstracción del concepto producto para la venta de una empresa comercial. Las siguientes dos secciones son ***guías*** y restricciones para el diseño del TAD pero no son ni la especificación y ni la implementación.

Valores

Un valor del TAD Producto representa una entidad que se puede vender. Cabe aclarar que no representa un ítem físico en particular, sino un ítem en general. Esto quiere decir que lo que interesa modelar son los ítems que una empresa puede vender y no todas las entidades físicas en particular. Tomando como ejemplo una librería, un producto puede ser *"bolígrafo azul, $0,80"*, y *no "el bolígrafo azul X de $0,80 que se encuentra en tal estante y en tal caja"*. Un valor del TAD producto tiene un *código*, pertenece a un *tipo de producto*, tiene un *nombre*, tiene un *proveedor* representado por su código, un *precio de venta*, y un *porcentaje de IVA* a aplicar.

El código del producto es alfanumérico y de un largo variable; su forma es la siguiente: *<número hexadecimal><identificador><tres dígitos decimales>*; los últimos tres dígitos representan el porcentaje de I.V.A. (e.g. 210 y 105). Ejemplos de código de producto son: ***0xFFABC12abc210*** y ***0xB1abc_efg105***. Los tipos de producto son Comestible, Textil, Servicio y Genérico. El *nombre* es una cadena no vacía de caracteres y de longitud *menor o igual a 40*. El código del *proveedor* es una cadena de *igual a 10*. El precio es un número racional que puede ser representado en notación decimal punto fijo, cuya parte entera es mayor o igual a 0 y menor a 1000 y su parte fraccionaria tiene dos dígitos.

(**1.1**) Especificar las cadenas utilizando ER y GR.

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

```
typedef enum {
    Comestible, Textil, Servicio, Generico
} TipoDeProducto;

#define LONGITUD_MAXIMA_NOMBRE_PRODUCTO    40
#define LONGITUD_CODIGO_PROVEEDOR          10

typedef struct {
    char           *elCodigo;
    TipoDeProducto elTipoDeProducto;
    char           elNombre[LONGITUD_MAXIMA_NOMBRE_PRODUCTO + 1];
    char           elProveedor[LONGITUD_CODIGO_PROVEEDOR];
    int            elPrecio;
} Producto;
```

(**1.2**) Buscar la forma de implementar el precio (que es valor racional, pertenece al conjunto **Q**) en un objeto del tipo `int` (que es un valor entero, pertenece al conjunto **Z**).

Operaciones

Operaciones de Creación

1. **Crear** : Código × TipoDeProducto × Nombre × Proveedor × Precio → Producto.
(**1.3**) Se debe construir una función privada que indica si la cadena dato es un código correcto o no.
`static int EsCodigoDeProducto(const char *cadena);`
Esta función es un predicado que define el lenguaje (conjunto) de los códigos de producto (i.e. dada una cadena retorna verdadero si es una palabra del lenguaje de códigos de producto). Implementar con un autómata finito reconocedor.
La forma de las palabras del lenguaje es: *<número hexadecimal><identificador><tres dígitos decimales>*. El *número hexadecimal* y el *identificador* siguen la misma sintaxis que ANSI C.

(**1.4**) Definir formalmente la gramática regular que lo genera y el autómata que la reconoce. Escribir la expresión regular que lo representa. Dibujar el diagrama de transiciones. (**1.5**) **Leer [MUCH2000] capítulos 11 y 12**.
(**1.6**) En la creación, aplicar malloc para reservar espacio para la estructura del producto y para el código del producto.
(**1.7**) Especificar la operación de creación de dos maneras: **con precondiciones** y **sin precondiciones**.

Operaciones de Proyección y Mutación (Getters y Setters)

4.1. **GetIva** : Producto → Porcentaje 4.2. **SetIva** : Producto × Porcentaje → Producto
(**1.6**) No existe un campo I.V.A. ¿Cómo pueden implementarse las anteriores dos operaciones?
5.1. **GetTipo** : Producto → TipoDeProducto 5.2. **SetTipo** : Producto × TipoDeProducto → Producto
6.1. **GetCodigo** 6.2. **SetCodigo**
7.1. **GetNombre** 7.2. **SetNombre**
8.1. **GetProveedor** 8.2. **SetProveedor**
9.1. **GetPrecio** 9.2. **SetPrecio**

Operaciones de Salida

10. **Mostrar** : Producto × Flujo → Flujo. Escribe en un flujo de texto un formulario con los datos del producto.
Ejemplo: Sea p ∈ Producto ∧ Mostrar(p, $stdout_1$) = $stdout_2$ entonces en el flujo *$stdout_2$* queda escrito un formulario con la siguiente formato:

```
          Código :\t□□□□□□\n
            Tipo :\t□□□□□□\n
          Nombre :\t□□□□□□\n
       Proveedor :\t□□□□□□\n
          Precio :\t$□□□□\n
          I.V.A. :\t□□□□%\n
```

El precio y el IVA siempre se muestran con dos dígitos decimales.
(**1.8**) Analizar por que *$stdout_1$* es diferente a *$stdout_2$*.
(**1.9**) Analizar la necesidad de precondiciones.

Operaciones de Persistencia

Permiten guardar y recuperar valores Producto en memoria externa, en formatos de texto y binario.
Binario
11. **Read** : Flujo → Producto × Flujo 12. **Write** : Producto × Flujo → Flujo
La secuencia de datos en un flujo binario es la siguiente:

Longitud del código	Código	Tipo	Nombre	Proveedor	Precio
`sizeof(int)` bytes	*n* bytes indicados por el anterior campo	1 byte	40 bytes	10 bytes	`sizeof(int)` bytes

Texto
13. **Scan** : Flujo → Producto × Flujo 14. **Print** : Producto × Flujo → Flujo
La secuencia de datos en un flujo de texto es la siguiente:
{*código*, *tipo*, *nombre*, *proveedor*, *precio*}**\n**
tipo ::= **C** | **T** | **S** | **G** el *precio* tiene la siguiente forma: $d^+.dd$ (ver **%f** de ***fprintf*** y ***fscanf***).

Prototipos de algunas de las funciones que implementan las operaciones

A modo de guía y de ejemplo, a continuación se presentan algunos prototipos:

```
Producto   *Producto_Crear(
       const char     *unCodigo,
       TipoDeProducto  unTipo,
       const char     *unNombre,
       const char     *unProveedor,
       float           unPrecio
);
void        Producto_Destruir(Producto *unProducto);
const char *Producto_GetNombre(const Producto *unProducto); /* Considerar otros prototipos */
Producto   *Producto_SetNombre(Producto *unProducto, const char *unNombre);
void        Producto_Mostrar(const Producto *unProducto, FILE *out);
Producto   *Producto_Read(FILE *in);
void        Producto_Print(const Producto *unProducto, FILE *out);—
```

Sintaxis y Semántica de Lenguajes 2006.1

TP – Sistema de Punto de Venta – TAD Ticket

20060525

*Por **Ing. José María Sola***

*Con la colaboración de **Facundo Merighi***

Introducción

El TAD Ticket permite a programadores de aplicaciones comerciales modelar el concepto de ticket, como un comprobante de venta de productos a un cliente. Las siguientes dos secciones son ***guías*** y restricciones para el diseño del TAD, pero no son ni la especificación ni la implementación.

Valores

Un ticket tiene un *número*, se paga mediante un *medio de pago*, es para un *cliente*, tiene una *bonificación*, un *recargo*, fue *emitido en una fecha y hora*, y por último tiene la lista de *productos que se vendieron* que son tuplas de la forma (*cantidad*, *nombre del producto, monto por I.V.A.* y *precio*). El cliente tiene un *nombre*, *dirección* y es de un *tipo*. La fecha es de la forma dd/mm/aaaa y la hora hh:MM. Los medios de pago son Efectivo, Débito, Crédito y Orden de Compra; los tipos de cliente son ConsumidorFinal, Excento, Inscripto. Los tipos se implementan con enums.

```
typedef enum {
    Efectivo,
    Debito,
    Credito,
    OrdenDeCompra
} MedioDePago;
```

```
typedef enum{
    ConsumidorFinal,
    Exento,
    Inscripto
} TipoDeCliente;
```

```
typedef struct {
    int laCantidadVendida;
    const char *elNombreDelProducto;
    float elIva;
    float elPrecio;
} ProductoVendido;
```

```
typedef struct {

    unsigned long int elNumero;
    MedioDePago elMedioDePago;

    struct {
        const char *elNombre;
        const char *laDireccion;
        TipoDeCliente elTipo;
    } elCliente;

    float laBonificacion;
    float elRecargo;

    struct {
        char elDia;
        char elMes;
        int elAnio;
    } laFecha;

    struct {
        char lasHoras;
        char losMinutos;
    } laHora;

    int laCantidadDeVentas;
    ProductoVendido *losProductosVendidos;

} Ticket;
```

El miembro `losProductosVendidos` es un puntero al comienzo de un arreglo de elementos del tipo `ProductoVendido`. La memoria necesaria para el arreglo varía en función de los elementos que se agreguen y se saquen, se deberá utilizar las funciones `calloc` ó `malloc` y `realloc`. Sea `Ticket *unTicket;` una variable correctamente inicializada, la siguiente expresión accede al último elemento del arreglo:

```
unTicket->losProductosVendidos[ unTicket->laCantidadDeVentas - 1 ]
```

(**2.1**) Presentar mediante un gráfico la estructura que implementa este tipo de dato.

Operaciones

1. **Crear** : Número × NombreDeCliente × Dirección × TipoDeCliente → Ticket

Operaciones sobre las Ventas

2. **VenderProducto** : Ticket × Producto × Cantidad → Ticket
3. **GetVenta** : Ticket × Índice → ProductoVendido
4. **CancelarVenta** : Ticket × Índice → Ticket. La venta indicada no forma más parte del ticket.
5. **GetCantidadDeVentas** : Ticket → Natural.

Operaciones de venta de Productos Genéricos
La funcionalidad que estas operaciones proveen es la de vender un producto aunque este no existan en el sistema. Un producto vendido genérico tiene la forma (**1**, **"genérico"**, **0.00**, *precio*); es decir, su cantidad siempre es 1, su nombre siempre es la cadena "genérico" y su I.V.A es 0. El precio es el resultado de evaluar una expresión. Las expresiones provienen de un flujo de datos o de una cadena. Estas operaciones son útiles cuando se desea programar un sistema con la capacidad de vender tanto productos registrados como productos no registrados (genéricos). Ejemplos de uso serían:

VenderProducto(t_1, *unProducto*, *2*) = t_2
VenderProductoGenericoDesdeCadena(t_2, "**21 + 33 * 57**") = (t_3, *Verdad*)
VenderProductoGenericoDesdeCadena(t_3, "*** 21 / 34 57 + X**") = (t_3, *Falso*)

6. **VenderProductoGenericoDesdeCadena** : Ticket × Cadena → Ticket × Lógico
7. **VenderProductoGenericoDesdeFlujo** : Ticket × Flujo → Ticket × Flujo × Lógico

Ambas operaciones implementan un evaluador de expresiones matemáticas simples, desarrollado con autómata reconocedor-accionador. El valor lógico indica si la expresión es válida.
Las 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 -. La primera versión de la operación evalúa la expresión desde una cadena; la segunda evalúa la expresión desde un flujo, carácter a carácter.
(**2.2**) Implementar las siguientes funciones privadas que serán invocadas por las anteriores operaciones:

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

Ambas funciones retornan 0 cuando la expresión es inválida.
(**2.3**) Definir formalmente la gramática regular que genera y el autómata que reconoce el lenguaje de expresiones matemáticas simples. Escribir la expresión regular que lo representa. Dibujar el diagrama de transiciones. (**2.4**) **Leer [MUCH2000] capítulos 11 y 12**.

Operaciones de Cierre

8. **BonificarEnPesos** : Ticket × Monto → Ticket
9. **BonificarEnPorcentaje** : Ticket × Porcentaje → Ticket
10. **RecargarEnPesos**
11. **RecargarEnPorcentaje**
12. **SetFecha** : Ticket × Año × Mes × Día → Ticket
13. **SetHora** : Ticket × Horas × Minutos → Ticket
14. **Emitir** : Ticket × MedioDePago → Ticket. Escribe por *stdout* el ticket según el siguiente formato:

dd/*MM*/*aaaaa***t***hh**:**mm***t*Numero: *número***n****n**
Saludo al cliente
n
*cantidad***t***nombre del producto***t***%I.V.A.****t***total***n**
*cantidad***t***nombre del producto***t***%I.V.A.****t***total***n**
...
*cantidad***t***nombre del producto***t***%I.V.A.****t***total***n**
nBonificacion:**t***d.dd***n**
Recargo:\t*d.dd***n**
I.V.A.:\t*d.dd***n**
Total:\t*d.dd***n**
*medio de Pago***n**
(**2.3**) Para escribir el mensaje *Saludo al cliente* en *stdout* se utilizará la biblioteca Saludos construida en el TP #0.
(**2.4**) Crear y utilizar la función privada: `static const char *GetMedioDePagoAsString(MedioDePago elMedioDePago);`

Operaciones de Persistencia

15. **Read**, 16. **Write**, 17. **Scan**, 18. **Print**

(**2.5**) Definir una representación medianamente eficiente del Ticket en un flujo binario.
El formato de un Ticket en un flujo de texto:

{\t*número***,** *medio de pago***,** *cantidad de ventas***,\n**
\t{*nombre, dirección, tipo***},\n**
\t*bonificación*, *recargo***,\n**
\t{*día*, *mes*, *año***}, {***horas*, *minutos***},\n**
\t{*cantidad vendida***,** *nombre del producto***,** *iva***,** *precio***}\n**
\t{*cantidad vendida***,** *nombre del producto***,** *iva***,** *precio***}\n**
\t...
\t{*cantidad vendida***,** *nombre del producto***,** *iva***,** *precio***}\n**
}\n
Los medios de pago se representan con un carácter: **E** para *Efectivo*, **D** para *Débito*, **C** para *Crédito* y **O** para *Orden de Compra*.
Los tipos de cliente se representan con un carácter: **C** para *Consumidor final*, **E** para *Exento*, e **I** para *Inscripto*.

Operaciones de Proyección (Getters)

15. **GetNumero** 16. **GetMedioDePago**
17. **GetNombreDeCliente** 18. **GetDireccionDeCliente** 19. **GetTipoDeCliente**
20. **GetDia** 21. **GetMes** 22. **GetAnio** 23. **GetHoras** 24. **GetMinutos**
25. **GetBonificacion** 26. **GetRecargo**

Operaciones de Consulta

27. **GetSubTotal** : Ticket → Monto. No incluye bonificaciones, ni recargos, ni I.V.A.
28. **GetTotalIva.** Monto total de I.V.A.
29. **GetTotal**. Monto neto total a cobrar.

Prototipos de algunas de las funciones que implementan las operaciones

A modo de guía y de ejemplo, a continuación se presentan algunos prototipos:

```
Ticket          * Ticket_Crear(
                      unsigned long int unNumero,
                      const char       *unNombreDeCliente,
                      const char       *unaDireccion,
                      TipoDeCliente     unTipoDeCliente
);
void             Ticket_Destruir(Ticket *unTicket);
Ticket          *Ticket_SetHora(
                      Ticket         *unTicket,
                      unsigned int    lasHoras,
                      unsigned int    losMinutos
);
ProductoVendido Ticket_GetVenta(const Ticket *unTicket, unsigned int unIndice);
int              Ticket_VenderProductoGenericoDesdeFlujo(
                      Ticket          *unTicket,
                      FILE            *in
);
Ticket          *Ticket_Emitir(Ticket *unTicket, MedioDePago unMedioDePago);
Ticket          *Ticket_Scan(FILE *in);
void             Ticket_Write(const Ticket *unTicket, FILE *out);—
```

Sintaxis y Semántica de Lenguajes 2006.1

TP – Sistema de Punto de Venta – TAD Sesión

20060611

*Por **Ing. José María Sola***

*Con la colaboración de **Tomás Colombo** y **Facundo Merighi***

Introducción

El TAD Sesión permite a programadores de aplicaciones comerciales modelar el periodo que un cajero determinado está registrando ventas con tickets para una sucursal, en una caja determinada. Las siguientes dos secciones son ***guías*** y restricciones para el diseño del TAD, pero no son ni la especificación ni la implementación.

Valores

Seis-uplas del tipo (*código de caja, sucursal*, *nombre del cajero*, *apertura*, *cierre*, *tickets*). La apertura y cierre son instantes determinados (i.e. una fecha y hora). La componente tickets representa el conjunto de tickets que el cajero registra durante la sesión.

```
typedef enum {
    Centro, Sur, Norte, Oeste
} Sucursal;

typedef struct {
    unsigned int elAnio     : 12;
    unsigned int elMes      :  4;
    unsigned int elDia      :  5;
    unsigned int lasHoras   :  6;
    unsigned int losMinutos :  6;
} Instante;

typedef struct {
    unsigned int      elCodigoDeCaja;
    Sucursal          laSucursal;
    const char       *elNombreDelCajero;
    unsigned long int elMayorNumeroDeTicket; /* Su utilidad depende de la implementación de AddTicket */
    Instante          laApertura;
    Instante          elCierre;
    int               laCantidadDeTickets;
   Ticket           **losTickets;
} Sesion;
```

Aunque en su uso no varía, para comprender que significan los dos puntos (**:**) y los números en la estructura nombrada por typedef como *Instante*. **(3.1) Leer [K&R1988] "Chapter 6 – Structures – 6.9 Bit-fields"**.

La implementación del conjunto `losTickets` sigue el mismo modelo de ANSI C para los parámetros de la función `int main(int argc, char *argv[])`. **(3.2) Leer [K&R1988] "Chapter 5 – Pointers and Arrays – 5.10 Command Line Arguments"**. El conjunto de tickets se implementa con un arreglo de punteros a `Tickets`. El miembro `laCantidadDeTickets` indica la longitud del arreglo, y `losTickets` es un puntero al primer elemento del arreglo. Los elementos del arreglo son del tipo de dato *puntero a Ticket*, por lo que `losTickets` debe ser del tipo de dato *puntero a puntero a Ticket*. Al implementar las operaciones que manipulan estos conjuntos, deberán utilizar `calloc` ó `malloc` y `realloc`.

Sea `Sesion *unaSesion;` una variable correctamente inicializada, la siguiente expresión accede al último elemento del arreglo:

```
unaSesion->losTickets[ unaSesion->laCantidadDeTickets - 1 ]
```

Operaciones

1. **Crear** : Sucursal × CódigoDeCaja × NombreDelCajero → Sesión
2. **Abrir** : Sesión × Día× Mes × Año × Horas × Minutos → Sesion
3. **Cerrar** : Sesión × Día× Mes × Año × Horas × Minutos → Sesión
4. **EscribirInforme** : Sesión × Flujo → Flujo. Escribe en el flujo dato un informe con el siguiente formato:

Sucursal: sucursal**t Caja:** caja**t Cajero:** cajero**n**
Apertura: dd/mm/aaaa hh:MM**n**
Cierre: dd/mm/aaaa hh:MM**n****n**
Ticket\tFecha\tHora\tCliente\tMedio de Pago\tTotal sin I.V.A.\tTotal con I.V.A.\n
número de ticket**t**dd/mm/aaaa hh:MM**t**nombre cliente**t**medio de pago**t**total sin iva**t**total con iva**n …**
número de ticket**t**dd/mm/aaaa hh:MM**t**nombre cliente**t**medio de pago**t**total sin iva**t**total con iva**n**
nBonificación: d.dd**tRecargos:** d.dd**tI.V.A.:** d.dd**tTotal:** d.dd**n**

(**3.3**) Las sucursales se escriben como cadenas: **Centro**, **Norte**, **Sur** y **Oeste**. Para ello se debe construir la siguiente función privada:

```
static const char *GetSucursalAsString(Sucursal unaSucursal);
```

Los medios de pago se representan con un carácter: **E** para *Efectivo*, **D** para *Débito*, **C** para *Crédito* y **O** para *Orden de Compra*.

Operaciones sobre el Conjunto de Tickets

5. **AddTicket** : Sesión × Ticket → Sesión
6. **RemoveTicket** : Sesión × NúmeroDeTicket → Sesión
7. **GetTicket** : Sesión × NúmeroDeTicket → Ticket
8. **GetTicketPorIndice**: Sesión × Índice → Ticket
9. **GetCantidadDeTickets** : Sesión → Natural
10. **GetMayorNumeroDeTicket** : Sesión → Natural

Operaciones de Consulta (Getters, Proyección)

Las siguientes operaciones son opcionales.
11. **GetSucursal** 12. **GetNombreDelCajero** 13. **GetApertura** 14. **GetCierre**.

Operaciones de Persistencia

15. **Read** 16. **Write** 17. **Scan** 18. **Print**

(**3.4**) Diseñar una secuencia de datos eficiente que represente la Sesión en un flujo binario. Explicar ese diseño.
La secuencia de datos en un flujo de texto es la siguiente:

{\n
t*sucursal**, **caja**, **cajero**, **cantidad de tickets**,**n**
t{*año**, **mes**, **día**, **horas**, **minutos**},**n**
t{*año**, **mes**, **día**, **horas**, **minutos**},**n**
t*ticket**,**n …**
t*ticket**,**n**
}\n

Las sucursales se representan con un carácter: **C** para *Centro*, **S** para *Sur*, **N** para *Norte* y **O** para *Oeste*.
Los tickets son escritos en el flujo por la operación ***print*** del TAD Ticket.

Operaciones de Estadística

19. **GetCantidadPromedioDeVentasPorTicket**
20. **GetMedioDePagoMasUtilizado**
21. **GetMontoPorMedioDePago** : Sesion × MedioDePago → Monto
22. **GetTotalDeBonificaciones**
23. **GetMontoPorNombreDeProducto** : Sesion × Nombre → Monto
24. **GetTicketConMayorMontoFinal**

Prototipos de algunas de las funciones que implementan las operaciones

A modo de guía y de ejemplo, a continuación se presentan algunos prototipos:

```
Sesion * Sesion_Crear(
    Sucursal     laSucursal,
    unsigned int elCodigoDeCaja,
    const char  *elNombreDelCajero
);
void    Sesion_Destruir(Sesion *unaSesion);
Sesion *Sesion_AddTicket(Sesion *unaSesion, const Ticket *unTicket);
Ticket *Sesion_GetTicket(const Sesion *unaSesion, unsigned long int unNumeroDeTicket);
float   Sesion_GetCantidadPromedioDeVentasPorTicket(const Sesion *unaSesion);
float   Sesion_GetMontoPorMedioDePago(
       const Sesion *unaSesion,
       MedioDePago unMedioDePago
);—
```