MANUAL DE *TOL*

Contenido

Contenido ... 1

1 Introducción ... 4

1.1 Trac de *TOL-Project*. Wiki ... 4

1.2 Sistema de tiques ... 5

1.3 Código fuente ... 6

1.4 Descarga e instalación ... 9

1.4.1 Instalación en *Windows* ... 9

1.4.2 Instalación en *Linux* ... 9

1.5 Programas *TOL* ... 10

1.5.1 La consola de comandos ... 10

1.5.2 *TOLBase* ... 11

2 El lenguaje *TOL* ... 15

2.1 Nociones básicas ... 15

2.1.1 Sintaxis ... 15

2.1.2 Variables ... 16

2.1.3 Números (`Real`) ... 18

2.1.4 Cadenas de texto (`Text`) ... 19

2.1.5 Funciones (`Code`) ... 20

2.2 Conjuntos ... 22

2.2.1 Conjuntos (`Set`) ... 22

2.2.2 Instrucciones de control ... 25

2.2.3 Consultas sobre conjuntos ... 29

2.2.4 Estructuras (`Struct`) ... 31

2.3 Estadística ... 33

2.3.1 Estadística descriptiva ... 33

2.3.2 Probabilidad ... 34

2.3.3 Matrices (`Matrix`) ... 35

2.3.4 Modelos lineales ... 38

2.3.5 Matrices virtuales (`VMatrix`) ... 39

2.4 Variables temporales ... 41

2.4.1 Fechas (`Date`) ... 41

2.4.2 Fechados (TimeSet) 43
2.4.3 Series temporales (Serie) 45
2.4.4 Diferencias finitas. Polinomios de retardos (Polyn) 49
2.4.5 Ecuaciones en diferencias. Cocientes de polinomios (Ratio) 50
2.4.6 Modelación con series temporales 51
2.5 Nociones avanzadas 53
2.5.1 Programación modular (NameBlock) 53
2.5.2 Clases (Class) 55
2.5.3 Diseño de clases 58
2.5.4 Otros elementos del lenguaje 61
2.5.5 Uso de la memoria 61
3 Uso de *TOL* 62
3.1 Sistema de archivos 62
3.1.1 Archivos .tol 62
3.1.2 Lectura y escritura de archivos 62
3.1.3 Serialización en *TOL*. *OIS* 62
3.1.4 Integración con el sistema operativo 62
3.2 Comunicación 62
3.2.1 Comunicación con el sistema operativo 62
3.2.2 Obtención de *urls* 62
3.2.3 Acceso a base de datos (DB<Function>) 65
3.3 Configuración de *TOL* 66
3.3.1 Módulo de configuración (TolConfigManager) 66
3.3.2 Otros mecanismos de configuración 67
3.4 Sistema de paquetes 67
3.4.1 Paquetes 67
3.4.2 Instalación y uso de paquetes (#Require) 68
3.4.3 Gestión de paquetes (TolPackage) 70
3.4.4 Repositorios de paquetes 71
3.4.5 Interfaz gráfica del gestor de paquetes 72
3.4.6 Código fuente de los paquetes 72
4 Paquetes de *TOL* 74

4.1 StdLib ... 74
4.1.2 Bloque TolCore ... 75
4.1.3 Paquete StdLib ... 75
4.2 GuiTools ... 76
4.2.1 Gestión de imágenes (ImageManager) ... 76
4.2.2 Gestión de menús contextuales (MenuManager) ... 77
4.2.3 Edición de un contenedor ... 80
4.3 TolExcel ... 82
4.4 Otros paquetes ... 85
4.4.1 Paquetes de *MMS* ... 86
4.4.2 TolGlpk ... 86
4.4.3 NonLinGloOpt ... 86
4.4.4 TolIpopt ... 86
4.4.5 MatQuery ... 86
4.4.6 BysMcmc ... 86
Índices ... 88
Índice de figuras ... 88
Índice de tablas ... 88

1 Introducción

Este manual tiene como objetivo introducir a los analistas en el uso del lenguaje *TOL*. *TOL* es un lenguaje básicamente orientado al análisis de información indexada en el tiempo y es especialmente útil para el análisis estadístico y la modelación de procesos dinámicos. Para ello cuenta con un álgebra del tiempo y series temporales así como de estructuras eficientes para el manejo y análisis de la información.

A lo largo del manual se introducirán los elementos sintácticos y semánticos del lenguaje así como algunas funciones y operadores para el manejo de los diferentes tipos de datos.

Este documento es también de obligada lectura para aquellos que se inicien en la modelación basada en *MMS* y no conozcan nada de programación en *TOL*.

Las características fundamentales de *TOL* son:

- Orientado al manejo de series temporales: dispone de un objeto para representar secuencias temporales, el cual es la base para la representación de series temporales. Véase la sección 2.4.
- Interpretado: esta característica dota al analista de cierta flexibilidad a la hora de construir prototipos ya que el ciclo de prueba y error es muy ágil (comparado con un lenguaje compilado).
- Fuertemente tipado: todas las expresiones en *TOL* tienen un tipo de dato concreto lo cual posibilita al intérprete aplicar optimizaciones en la evaluación del código. Véase la sección 2.1.1.
- Auto-evaluable: permite evaluar código *TOL* dentro del código *TOL*. Esto posibilita la creación de programas *TOL* parametrizados. Aunque es una construcción muy flexible no se debe abusar de la misma ya que las expresiones evaluadas de esta manera deben ser analizadas por el *parser* cada vez.
- Orientado a objetos: aunque no es un lenguaje puramente orientado a objetos, sí implementa los conceptos de clases con herencia múltiple facilitando la implementación de soluciones bajo el diseño orientado a objetos. Esta es una característica implementada a partir de la versión 2.0.1. Véase la sección 2.5.2.

El entorno de trabajo de *TOL* está formado por un sitio web el cuál es el sitio de atención a los usuarios de *TOL* y también el punto desde donde podemos descargar las herramientas de evaluación de *TOL*. En este capítulo describiremos dicho entorno de trabajo.

1.1 Trac de *TOL-Project*. Wiki

El desarrollo de *TOL*, proyecto *TOL-Project*, se centraliza en el sitio web http://www.tol-project.org, el cual es punto de encuentro entre los usuarios de *TOL* y los desarrolladores del lenguaje.

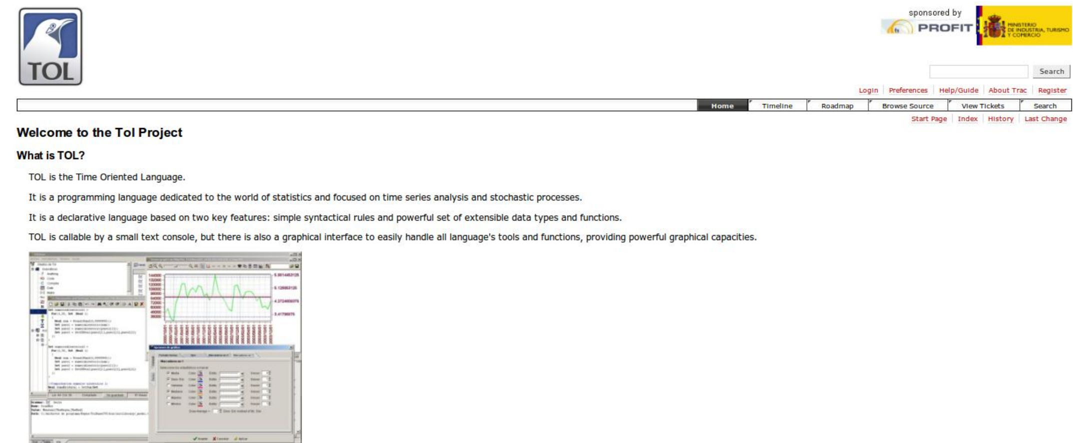

Figura 1.1.1: Sitio web de *TOL-Project*.

TOL-Project está basado en el sistema *trac* (http://trac.edgewall.org) el cual ofrece el marco colaborativo (*wiki*) a los desarrolladores de *TOL* para crear contenido informativo sobre el lenguaje así como un sistema para el reporte y seguimiento de incidencias (tiques) sobre *TOL* y su uso por parte de la comunidad de usuarios.

El primer paso recomendado a un nuevo usuario de *TOL* es registrarse en el *trac* de *TOL-Project*. A partir de ese momento podrá interactuar con la comunidad de usuarios y desarrolladores a través del sistema de reporte y seguimiento de tiques. Únicamente se le exige un nombre de usuario único, la elección de una contraseña y por supuesto ganas de colaborar con *TOL*.

1.2 Sistema de tiques

Hay muchas formas de colaborar en el desarrollo de *TOL*, la más básica es usarlo y reportar dudas, errores y sugerencias de trucos y posibles mejoras. En el mejor de los casos se puede aportar una solución a un error y ofrecerla para su integración en futuras versiones de *TOL*.

Create New Ticket

Properties

Summary:

Reporter: josp

Description:

Assign to: | Type: defect

Priority: normal | Milestone: Mantainance

Component: Kernel | Version: 3.1

Severity: normal | Keywords:

Cc:

I have files to attach to this ticket

Preview | Create ticket

Figura 1.2.1: Formulario de creación de un nuevo tique.

Cuando se reporta un tique hay que aportar un resumen (campo Summary) y una descripción (campo Description) de la solicitud. Existe también la opción de asignar una prioridad (campo

Priority) lo cual indicaría la importancia que se le está dando a esa solicitud, conjuntamente se puede asociar un nivel de severidad (campo Severity)

Otros campos importantes en la generación de un tique son:

- Component: módulo o componente de *TOL* con el que está relacionado el tique. En caso de dudas, podemos dejar el valor por omisión. Cada componente tiene asociada un desarrollador, de modo que si dejamos vacío el campo Assign to se asignará automáticamente.
- Cc: nombres de usuario y/o direcciones de correo a las cuales se les quiere hacer llegar las notificaciones sobre los cambios en la solicitud.
- Milestone: en un hito o categoría en las cual debe quedar resuelta la solicitud. El *milestone* se utiliza para la planificación de versiones, aunque en *TOL-Project* se usa más para clasificar conceptualmente la solicitud.
- Version: el número de la versión de *TOL* donde se detecto el error, es muy importante pues el primer paso es reproducir la situación lo más exactamente posible y para eso se requiere usar la misma versión de *TOL*.
- Keywords: es una lista de palabras claves asociadas a la solicitud, es muy útil a la hora de realizar búsquedas por términos específicos.

También existe también la posibilidad de adjuntar archivos que ayuden a comprender situación descrita, por ejemplo: código fuente en *TOL*, imágenes del momento del error, etc.

Nótese que el texto incluido en la descripción del tique puede hacer uso de la capacidad que ofrece el *wiki* del *trac* para la escritura formateada del contenido. El *wiki* soporta un formato hipertexto con una sintaxis muy simple. Puede verse una descripción de la sintaxis en https://www.tol-project.org/wiki/WikiFormatting.

Téngase en cuenta que es muy importante aportar toda la información posible en el tique, en particular, en el caso de reportes de errores, puede ser de gran valor incluir en la descripción o en un archivo adjunto un trozo de código que permita al desarrollador reproducir el error en su entorno de trabajo. Esto no siempre es posible o no se tiene el tiempo suficiente para aislar el error, no importa, es más importante aportar una descripción, la traza del error o el log de *TOL* que no informar y dejar el error escondido.

1.3 Código fuente

A partir del año 2002, *TOL* adoptó el modelo de desarrollo bajo una licencia de código abierto, en particular se aplica la licencia *GNU GPL*. Este modelo de desarrollo nos ha permitido hacer uso de paquetes de software desarrollados bajo licencias de código abierto compatibles impulsando enormemente el desarrollo de *TOL*.

Gran parte de los algoritmos de *TOL* descansan en el uso de otros paquetes disponibles en forma de código fuente, entre los que podemos destacar:

- *ALGLIB*: librería multiplataforma para el análisis numérico y procesamiento de datos. http://www.alglib.net
- *BLAS*: librería para el álgebra lineal básica. http://www.netlib.org/blas/index.html
- *Boost Spirit*: componente de *Boost* para la implementación de analizadores sintácticos descendentes recursivos. http://spirit.sourceforge.net, http://www.boost.org

- *BZip2*: compresor de datos de alta cualidad. http://www.bzip.org
- *CLUSTERLIB*: librería para la segmentación, escrita en *C*. http://bonsai.ims.u-tokyo.ac.jp/~mdehoon/software/cluster/software.htm#source
- *DCDFLIB*: librería *C/C++* para la evaluación de funciones de distribución. http://people.scs.fsu.edu/~burkardt/cpp_src/dcdflib/dcdflib.html
- *Google Sparse Hash*: implementación de la estructura de datos *hash_map*, extremadamente eficiente. http://code.google.com/p/google-sparsehash
- *GSL*: librería científica de *GNU*. http://www.gnu.org/software/gsl
- *KMLOCAL*: implementación, en *C++*, eficiente para la segmentación basado en *K-Means*. http://www.cs.umd.edu/users/mount/Projects/KMeans
- *LAPACK*: paquete de algebra lineal, compañera inseparable de *BLAS*. http://www.netlib.org/lapack/index.html
- *Optimal_bw*: implementación eficiente de la estimación de la densidad núcleo univariada con elección óptima de ancho de banda. http://www.umiacs.umd.edu/~vikas/Software/optimal_bw/optimal_bw_code.htm
- *SuiteSparse CHOLMOD*: librería para la factorización *cholesky* en almacenamiento disperso. http://www.cise.ufl.edu/research/sparse/cholmod
- *ZipArchive*: librería *C++* para la compresión en formato *ZIP* de datos y archivos. http://www.artpol-software.com.

TOL está implementado en *C++* y corre en sistemas operativos *Linux* y *Windows* y para interactuar con el lenguaje existen diferentes interfaces como pueden ser: consola de línea de comandos, servidor de comandos remotos, interfaz gráfica de escritorio o interfaz web; también es posible hacer uso del lenguaje mediante un enlace dinámico con la biblioteca de *TOL*, actualmente esto es posible desde *C++*, *Java*, *Tcl* y *Visual Basic*.

Aunque este manual no está dirigido a desarrolladores vamos a indicar como acceder y explorar el código fuente de *TOL*. Esto puede llegar a ser útil para revisar parte de la biblioteca estándar la cual está implementada en *TOL* y cuyo código fuente se aloja en el mismo repositorio que el del núcleo de *TOL* escrito en *C++*.

Hoy en día no se concibe desarrollar software sin el apoyo de un sistema de control de versiones. Existen diferentes programas para tal propósito algunos propietarios, otros de uso libre y muy conocidos como *CVS*, *GIT*, *FOSSIL*, *SVN*, entre otros. El código fuente de *TOL* se aloja en el sistema de control de versiones *SVN* (http://subversion.apache.org). Con el *SVN* podemos gestionar varias versiones del código, explorar toda la historia de cambios, conocer las diferencias en código fuente que hay de una versión a otra, identificar el desarrollador responsable de un cambio, entre otras facilidades.

Podemos explorar el código fuente de *TOL* desde el propio *TRAC* o apoyándonos en un programa cliente de *SVN*. La mayoría de las distribuciones de *Linux* ofrecen la opción de instalar *SVN* desde su repositorio de paquetes, en el caso de *Windows* podemos descargar un programa cliente desde http://subversion.apache.org/packages.html.

A través de la interfaz web podemos explorar el contenido del repositorio y acceder a los archivos contenidos en los directorios registrados en el *SVN*. El código fuente de *TOL* se encuentra alojado debajo del directorio *tolp*, dentro del cual podemos identificar los directorios *trunk* y *branches*, entre otros. El directorio *trunk* contiene el código base a partir del cual se

construye la versión en desarrollo de *TOL*, la cual suele contener funcionalidades poco estables y probadas. En el directorio *branches* se ubican las diferentes versiones liberadas cada una con su propia historia de cambios. Estas versiones suelen ser más estables que la versión *trunk*, aunque con menos funcionalidades.

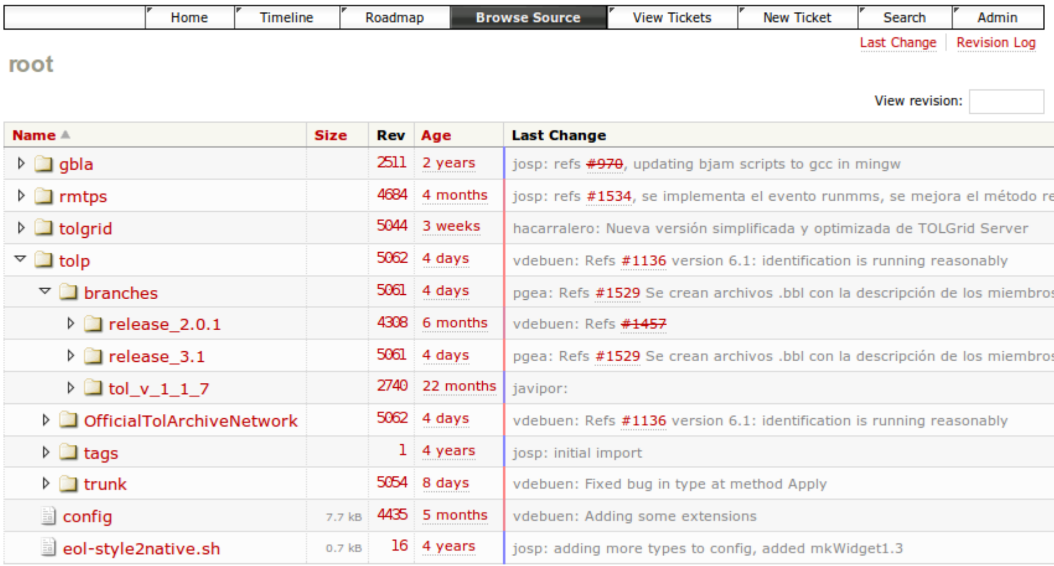

Figura 1.3.1: Explorador web del código fuente de *TOL* en el *trac* de *TOL-Project*.

Con el *SVN*, cada cambio realizado por un desarrollador implica un incremento en un número interno que identifica una foto de todo el repositorio, aunque internamente el cambio se almacena como diferencia respecto al estado anterior. A ese número se le conoce como *revisión*.

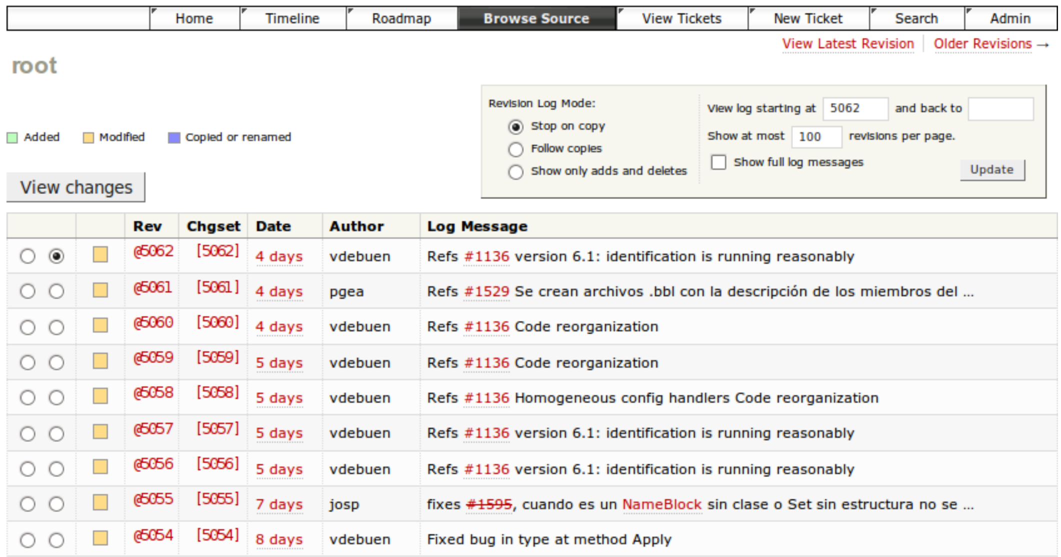

Figura 1.3.2: Explorador de revisiones del código de *TOL* en el *trac* de *TOL-Project*.

Otras de las funcionalidades accesibles desde la interfaz web es la exploración de la historia de cambios. Podemos ver qué cambios en el código condujeron el código a su última revisión, así como los cambios que hay entre un número de revisión y otro más reciente. Para ello debemos seleccionar las revisiones que queremos comparar y a continuación presionar el botón View Changes.

1.4 Descarga e instalación

Como se ha mencionado anteriormente *TOL* puede utilizarse en *Windows* y *Linux*. Los programas binarios que generamos para ambos sistemas, hasta el momento de escribir este manual, son de *32 bits*. Así, aunque el ordenador sea de *64 bits* debemos usar *TOL* compilado para *32 bits*.

1.4.1 Instalación en *Windows*

Para instalar los intérpretes de *TOL* en *Windows* debemos descargar el programa instalador desde https://www.tol-project.org/wiki/DownloadTol. En esa página se muestra una tabla con las versiones disponibles para instalar *TOL*.

Tol Downloads

Date released	Release	Name	Description	Download	Other binaries
unreleased	Development	Tol.3.2		Win32 Installer	History
2012-02-16	OFFICIALLY RECOMENDED	Tol.3.1	What is new	Win32 Installer	History Also called 2.0.2
2011-05-03	Old stable	Tol.2.0.1	What is new	Win32 Installer	History
2009-10-23	Unmantained	Tol.1.1.7.bridge	What was new	Win32 Installer	
2009-02-25	Unmantained	Tol.1.1.7	What was new	Win32 Installer	History
2007-11-13	Unmantained	Tol.1.1.6	What was new	Win32 Installer	History
2007-02-22	Unmantained	Tol.1.1.5	What was new	Win32 Installer	History
2006-11-11	Unmantained	Tol.1.1.4	What was new	Win32 Installer	
2005-02-16	Unmantained	Tol.1.1.3	What was new	Win32 Installer	
2003-03-05	Unmantained	Tol.1.1.2	What was new	Win32 Installer	
2001-04-07	Unmantained	Tol.1.1.1	What was new	Win32 Installer	

Figura 1.4.1: Tabla de versiones disponible para descargar en *Windows*.

En la primera fila de la tabla aparece la versión en desarrollo. En las filas sucesivas se listan las versiones liberadas en orden creciente de antigüedad. El enlace para la descarga aparece en la columna etiquetada con Download

Al ejecutar el instalador, por ejemplo el descargado desde el enlace http://packages.tol-project.org/win32/tolbase-v3.2-setup.exe obtendremos como resultado la instalación de los archivos binarios que incluyen el interfaz gráfico *TOLBase*, la consola de línea de comandos (programar *tol* y *tolsh*) y la librería *vbtol* para hacer uso de *TOL* desde *Visual Basic*.

Hay algunas funciones de usuario implementadas en *TOL* que descansan en *R* invocándolo externamente vía línea de comandos. Estas funciones requieren entonces que tanto *R* como los paquetes *quadprog*, *coda*, *Rglpk* y *slam* sean instalados. Para ello debe descargar e instalar *R* desde http://www.r-project.org y a continuación ejecutar en una consola de R las siguientes instrucciones:

```
install.packages("quadprog")
install.packages("coda")
install.packages("Rglpk")
install.packages("slam")
```

1.4.2 Instalación en *Linux*

En *Linux* no disponemos de un instalador similar al de *Windows*, por eso lo más usual es compilar el código fuente, lo cual requiere de ciertas habilidades sobre el proceso de compilación en *Windows*, así como el dominio de algunos comandos *Linux* para la instalación de los requisitos de compilación. En este manual no se describirán los pasos para compilar el código fuente de los intérpretes de *TOL*.

Dicho esto, sí que existe un empaquetado de *TOL* que permite instalar *TOL* en la distribución de *Linux* denominada *CentOS*. Dicho empaquetado ha sido probado en la versión *5.4* de *CentOS*.

Esta distribución para instalar *TOL* en *CentOS* puede descargarse de: http://packages.tol-project.org/linux/binaries.

En este caso, el proceso de instalación se realizaría siguiendo los siguientes pasos:

- Instalar los prerrequisitos

```
sudo rpm -Uvh sysreq/epel/5/i386epel-release-5-4.noarch.rpm
sudo yum install atlas-sse2.i386
sudo ln -s /usr/lib/atlas/liblapack.so.3 /opt/tolapp-3.1/lib/liblapack.so
sudo ln -s /usr/lib/atlas/libf77blas.so.3 /opt/tolapp-3.1/lib/libblas.so
sudo yum install glibc-devel.i386 gsl.i386 R-core.i386 R-devel.i386
echo 'options(repos="http://cran.r-project.org")' > /tmp/Rinstall.R
echo 'install.packages("coda")' >> /tmp/Rinstall.R
echo 'install.packages("quadprog")' >> /tmp/Rinstall.R
echo 'install.packages("Rglpk")' >> /tmp/Rinstall.R
sudo R BATCH -f /tmp/Rinstall.R
rm /tmp/Rinstall.R
```

- Descargar la distribución

```
cd /tmp
wget http://packages.tol-project.org/linux/binaries/TOLDIST_3.1_p012.tar.bz2
tar zxf TOLDIST_3.1_p012.tar.bz2
cd TOLDIST_3.1_p012
```

- Instalarla

```
sudo ./install --prefix=/opt/tolapp
```

Las librerías y programas de *TOL* se instalarán debajo del directorio apuntado por el argumento `--prefix`. A partir de ese momento podemos interactuar con la consola de modo texto de *TOL* mediante el siguiente comando:

```
/opt/tolapp/bin/tolsh -d
```

```
TOL interactive shell activated...
15:29:30 TOL>
```

Ahora ya podemos ejecutar sentencias *TOL*, como por ejemplo:

```
WriteLn(Version);
```

```
v3.1 p012 2012-06-14 19:33:46 +0200 CEST i686-linux-gnu
```

1.5 Programas *TOL*

Al instalar el software de *TOL*, tanto en *Windows* como en *Linux*, obtenemos un conjunto de programas y librerías que nos permiten desarrollar soluciones escritas en el lenguaje *TOL*. De estos programas, los más usados son la consola de comandos (*tol* o *tolsh*) y el interfaz gráfico *TOLBase*.

1.5.1 La consola de comandos

Los programas *tol* y *tolsh* son intérpretes del lenguaje *TOL* usados fundamentalmente para el procesamiento en lote de programas escritos en *TOL*. También podemos invocarlo en modo interactivo y ejecutar expresiones *TOL* escritas en la consola del *DOS* (*Windows*) o *SH* (*Linux*).

Cuando es ejecutado en modo interactivo, cada expresión evaluada genera un resultado que es almacenado en una pila de objetos, que nos permite en futuras expresiones evaluadas en la misma sesión.

Los programas *tolsh* y *tol* operan de manera similar, excepto que *tolsh* implementa un modo servidor que le permite permanecer en ejecución escuchando en un puerto *TCP/IP* por órdenes de evaluación remota. Las órdenes de evaluación remota les llegan, normalmente, desde otro programa cliente *TOL*. Un programa cliente *TOL* puede ser el propio *tolsh*, *TOLBase* u otro vinculado dinámicamente a la librería de *TOL*.

Al invocar el intérprete *TOL* vía línea de comandos podemos especificar varios archivos .tol que serán interpretados en el orden que se indiquen. Además podemos hacer uso de las siguientes opciones:

- `-i`: no incluye la librería estándar de *TOL*.
- `-c"..."`: evalúa la expresión *TOL* especificada entre comillas.
- `-d`: inicia *TOL* en modo interactivo. Después de evaluar los archivos y expresiones especificadas en la línea de comando, muestra al usuario una línea de entrada donde teclear expresiones *TOL*, las expresiones *TOL* son evaluadas al presionar la tecla de retorno de carro. El resultado de la evaluación se muestra en la pantalla y se solicita una nueva expresión.
- `-vE`: habilita la salida de mensajes de tipo error. Los mensajes de tipo error son los emitidos por la función `Error` o `WriteLn` con el argumento tipo de mensaje `"E"`.
- `-mE`: deshabilita la salida de mensajes de tipo error.
- `-vW`: habilita la salida de mensajes de tipo advertencia (*warning*). Los mensajes de tipo advertencia son los emitidos por la función `Warning` o `WriteLn` con argumento tipo de mensaje `"W"`.
- `-mW`: deshabilita la salida de mensajes de tipo advertencia (*warning*).
- `-vS`: habilita la salida de mensajes del sistema. Los mensajes de sistema son los mensajes de notificación emitidos por las funciones y algoritmos internos de *TOL*, por ejemplo `Estimate`.
- `-mS`: deshabilita la salida de mensajes del sistema.
- `-vU`: habilita la salida de mensajes de usuario. Los mensajes de usuario son los mensajes emitidos por `WriteLn` con argumento tipo de mensaje igual a `"U"`.
- `-mU`: deshabilita la salida de mensajes de usuario.
- `-vT`: habilita la salida de mensajes de tipo traza. Los mensajes de traza son los mensajes emitidos por `WriteLn` con argumento tipo de mensaje igual a `"T"` y los emitidos por la función interna `Trace`.
- `-mT`: deshabilita la salida de mensajes de tipo traza.
- `-v?A?`: habilita todos los tipos de salidas.
- `-m?A?`: deshabilita todos los tipos de salidas.

1.5.2 *TOLBase*

TOLBase es un programa cliente de *TOL* que ofrece una interfaz gráfica para interactuar con el intérprete de *TOL* y los objetos creados en él como resultado de la evaluación de código *TOL*.

En la figura 1.5.1 podemos ver los elementos iniciales con los que nos encontramos al abrir *TOLBase*. La imagen muestra la ventana principal conocida como *inspector de objetos*. Éste está formado por tres paneles:

- Un árbol de objetos.
- Un panel con los elementos del objeto seleccionado en el árbol.
- Un panel con la pestaña de evaluación (Eval) conocida como *consola*, la pestaña de salida de las evaluaciones (Salida) y una pestaña de información (Info).

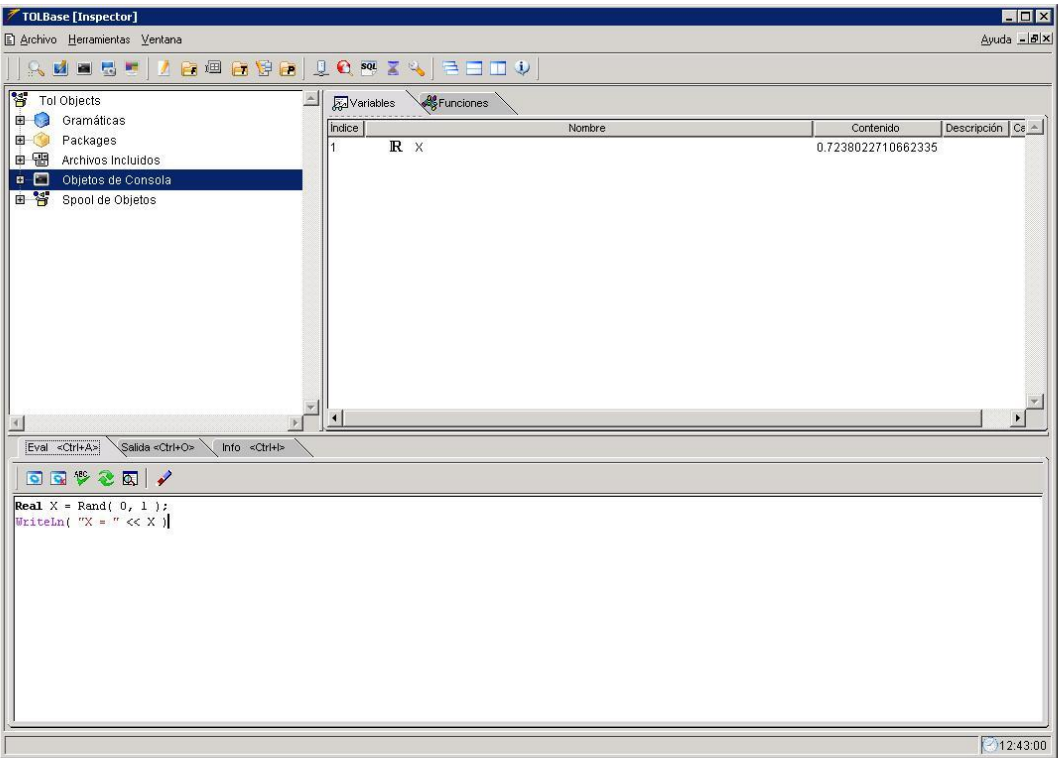

Figura 1.5.1: Ventana principal de *TOLBase*: inspector de objetos.

El árbol de objetos tiene cinco nodos principales. Estos nodos listan objetos *TOL* que son a la vez contenedores de otros objetos *TOL*. Estos nodos principales son:

- Gramáticas: despliega el conjunto de tipos de datos implementados en *TOL*, para cada tipo de dato existe un sub-árbol que a su vez contiene las variables globales creadas en esa sesión de evaluación.
- Packages: despliega los paquetes de *TOL* cargados en esa sesión de *TOL*, debajo de cada paquete se despliega un sub-árbol con los miembros del paquete. Véase la sección 3.4.
- Archivos Incluidos: despliega el conjunto de archivos .tol incluidos en esa sesión de evaluación. Desde cada archivo se puede desplegar el sub-árbol de objetos creados en la evaluación de dicho archivo.
- Objetos de Consola: contiene la lista de objetos evaluados en la *consola*, los objetos se listan en el orden que fueron creados.
- Spool de Objetos: Para aplicar opciones de menú contextual sobre una selección de objetos, por ejemplo, graficar un grupo de series temporales, los objetos deben pertenecer a un contenedor común. El *spool* es un contenedor virtual de objetos cuyo propósito es reunir en un único contenedor objetos que pertenecen a contenedores

distintos. Los objetos son insertados en el *spool* mediante una opción de menú contextual ejecutada sobre un objeto ya existente.

TOLBase ofrece otras muchas facilidades para la edición de programas .tol y la visualización de los diferentes tipos de objetos que se pueden crear. Estas facilidades serán exploradas en los capítulos siguientes en la medida que vayamos visitando los diferentes tipos de datos disponibles.

Sólo como una muestra en la figura 1.5.2 podemos ver una sesión de *TOL* en la cual se ha evaluado 3 expresiones en la *consola*, se ha graficado una serie temporal y se ha abierto un archivo *.tol* para su edición.

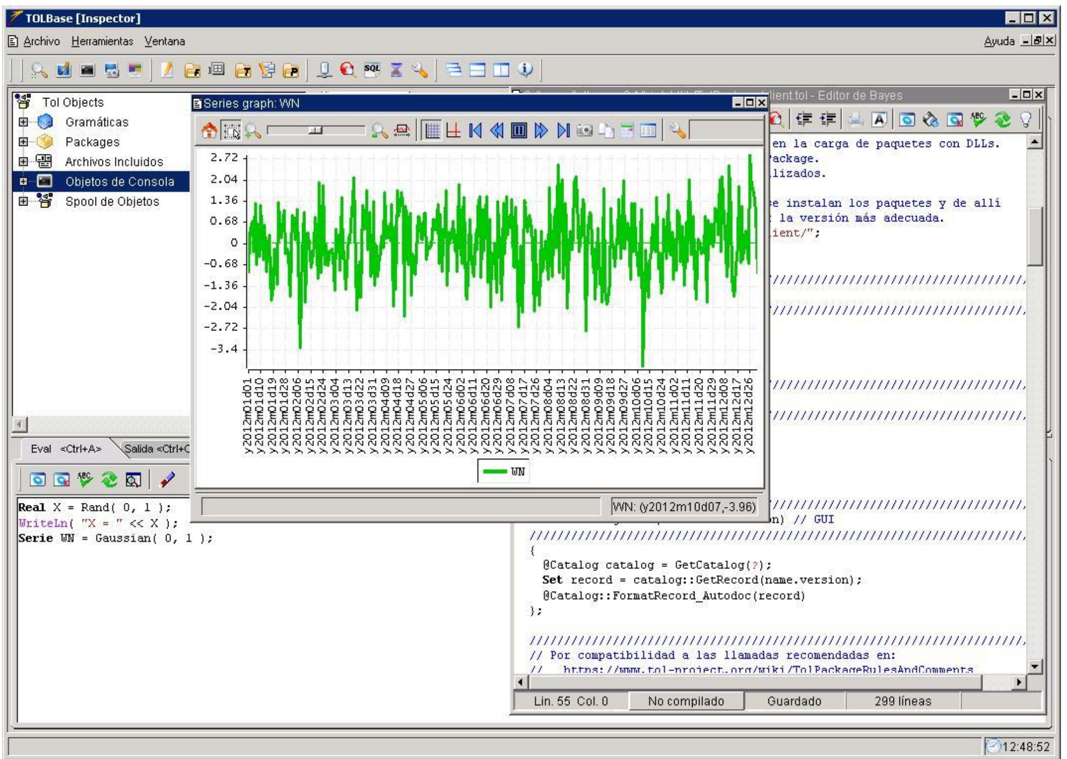

Figura 1.5.2: *TOLBase* mostrando algunas de sus funcionalidades: consola de evaluación de código *TOL*, gráficos de series temporales y edición de archivos.

Para ejecutar un conjunto de sentencias de la sentencia escritas en la *consola* podemos usar el botón Compilar . La acción asociada a dicho botón es evaluar todas las expresiones seleccionadas en la consola o todo el código de la consola si no hay selección activa. Esta acción también está asociada a la tecla F9 y disponible en el menú contextual desplegado sobre la *consola*.

TOL impide la creación de un objeto con un nombre que ya existe, por lo que para facilitar la evaluación continuada de un trozo de código con el cual estamos experimentando se ofrece la funcionalidad de destruir los objetos creados. El botón Decompilar , disponible en la *consola*, destruye los objetos creados en evaluaciones previas y que están accesibles debajo del nodo Objetos de Consola en el inspector de objetos. Esta acción está vinculada a la tecla F8.

También podemos encontrar muy útil la funcionalidad de chequeo de sintaxis lo cual nos permite verificar si el código *TOL* escrito sigue las reglas sintácticas del lenguaje, esto sin necesidad de evaluar el código con la consiguiente creación de los objetos *TOL*. Esta opción está

disponible a través del botón Syntax y está asociada a la tecla F7. A veces el mensaje de error que emite *TOL* es un error sintáctico y puede resultar difícil encontrar el sitio exacto del error, en tal caso una técnica que puede ayudar a encontrarlo es aplicar el chequeo de sintaxis a trozos de código seleccionado hasta aislarlo en un trozo pequeño más fácil de diagnosticar.

TOLBase mantiene un histórico de las expresiones evaluadas en la *consola* en un archivo interno. Dicho archivo se puede recuperar haciendo uso de la acción asociada al botón Mostrar archivo de histórico que abre una ventana de edición de archivo con el contenido del histórico de comandos ejecutados en la *consola* de los más recientes a los más antiguos.

La ventana de edición de archivo ofrece también las funcionalidades anteriores, además de las clásicas funciones de un editor de texto. En el caso de un archivo editado, el resultado de su evaluación aparece en el inspector de objetos como un nodo debajo del nodo Archivos Incluidos. Este nodo archivo contiene el sub-árbol cuyos nodos son los objetos *TOL* resultado de la evaluación. Así mismo la destrucción del contenido de un archivo evaluado tiene como efecto la eliminación de memoria los objetos pertenecientes a este archivo y sus descendientes.

2 El lenguaje *TOL*

2.1 Nociones básicas

2.1.1 Sintaxis

Como veíamos en la Introducción, *TOL* es un lenguaje de programación interpretado, y las sentencias de código se pueden ir ejecutando sin hacerles previamente ningún tratamiento.

Si estamos utilizando *TOLBase*, disponemos de algunas utilidades añadidas que nos permiten compilar y decompilar sentencias de código o archivos .tol, organizar el código o inspeccionar las variables. Sin embargo en este capítulo de sintaxis nos ceñiremos al uso de *TOL* como lenguaje y a su uso no importa sobre qué aplicación.

En *TOL*, con el término *compilar* se hace referencia a la acción de interpretar un determinado código, llevando a cabo las acciones que se indiquen y creando las variables que se declaren.

A lo largo de este capítulo incorporaremos líneas de código a modo de ejemplo e indicaremos la salida que nos devuelve el intérprete de *TOL* al compilarlas.

Por lo común, *TOL* no devuelve ninguna salida al compilar el código, dejando la salida para mensajes informativos, advertencias o errores. Para comprobar el resultado de nuestras acciones debemos solicitar su impresión o inspeccionarlos con alguna interfaz gráfica para *TOL*.

Gramáticas

TOL dispone de distintos tipos de datos, denominados *gramáticas*, con los que construir las sentencias de código. Prácticamente toda sentencia en *TOL* devuelve una de estas variables.

Las gramáticas reconocidas por *TOL* son: `Anything`, `Real`, `Complex`, `Text`, `Set`, `Date`, `TimeSet`, `Serie`, `Polyn`, `Ratio`, `Matrix`, `VMatrix` y `NameBlock`.

Téngase en cuenta que la gramática `Anything` no es estrictamente un tipo de variable, sino que se utiliza para designar de manera general a cualquier gramática.

Sintaxis

La sintaxis básica de las sentencias *TOL* es una gramática, el código de la sentencia y un punto y coma (`;`) para acabar:

```
<Grammar> <code...>;
```

En los ejemplos de código *TOL* de este documento utilizamos los símbolos <> (paréntesis angulares) para indicar que esa parte ha de ser sustituida por una expresión válida. Téngase en cuenta, por tanto, que éstos no forman parte de la sintaxis.

Otras características del lenguaje a tener en cuenta son que:

- Distingue mayúsculas y minúsculas.
- Permite partir las líneas para facilitar su lectura. La sentencia en *TOL* no acaba con el salto de línea, sino con el punto y coma (`;`).

Automáticamente las sentencias pasan un chequeo sintáctico previo a la compilación. Si este test falla, la compilación no se lleva a cabo y se muestra un mensaje de error para facilitar su detección.

Comentarios en el código (// y /* */)

Una buena costumbre al escribir el código es acompañarlo con comentarios que faciliten su compresión y nos sirvan para recordar lo que motivó una u otra decisión.

TOL dispone, como ocurre en otros lenguajes, de dos modos de comentar las líneas de código:

- La doble barra (//) para comentar una línea completa o desde su aparición hasta el final de la línea.
- Los pares barra-asterisco y asterisco-barra (/* y */) que nos permite encerrar como si fuesen paréntesis la parte del código que no debe ser compilada, pudiéndose comentar así incluso varias líneas de código de una vez.

Por ejemplo:

```
// Se crea una variable 'suma' con el resultado de sumar 1+1
Real suma = 1+1;
```

2.1.2 Variables

Como ya hemos dejado entrever, la programación en *TOL* se apoya fuertemente en la creación de variables. Para definir una variable hemos de asignarle un valor, ya que en *TOL* no se pueden crear variables sin asignar.

Asignación (=)

Para crear una variable hemos de usar el operador = (signo igual). La sintaxis es:

```
<Grammar> <name> = <value>;
```

Por ejemplo:

```
Real a = 1.5;
Text b = "Hola";
```

Aunque, en general, las sentencias en *TOL* siempre devuelven una variable, si no tenemos interés en utilizarla no es necesario asignarles un nombre.

Por ejemplo:

```
Real 1.5;
Text "Hola";
```

Nombre de las variables

Los nombres de las variables en *TOL* pueden construirse con una combinación cualquiera de caracteres alfanuméricos siempre que el primero de ellos no sea de tipo numérico.

Aunque *TOL* admite el uso de los caracteres del ASCII extendido de nuestro sistema (por lo común LATIN1) para la elección de nombres se recomienda usar únicamente los caracteres del ASCII estándar.

A continuación indicamos los distintos conjuntos de caracteres según su posible uso:

<table>
<tr><td>Letras mayúsculas</td><td>ABCDEFGHIJKLMNOPQRSTUVWXYZ</td></tr>
<tr><td>Letras minúsculas</td><td>abcdefghijklmnopqrstuvwxyz</td></tr>
<tr><td>Otros caracteres considerados alfabéticos</td><td>_</td></tr>
<tr><td>Cifras</td><td>0123456789</td></tr>
<tr><td>Otros caracteres considerados numéricos</td><td>#'.</td></tr>
<tr><td>Caracteres considerados operadores</td><td>!\"$%&()*+,-/:;<=>?@[\\]^`{|}~</td></tr>
</table>

Tabla 2.1.1: Clasificación de los caracteres ASCII en *TOL*.

<table>
<tr><td>Letras mayúsculas</td><td>ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞŸŠŒŽ</td></tr>
<tr><td>Letras minúsculas</td><td>àáâãäåæçèéêëìíîïðñòóôõöøùúûüýþÿšœž</td></tr>
<tr><td>Otros caracteres considerados alfabéticos</td><td>ƒªµºß</td></tr>
<tr><td>Caracteres considerados operadores (ignorados actualmente)</td><td>€‚„…†‡ˆ‰‹‘’“”•–—™›
¡¢£¤¥¦§¨©«¬-®¯°±²³´¶·¸¹»¼½¾¿×÷</td></tr>
</table>

Tabla 2.1.2: Clasificación de los caracteres extra de LATIN1 (ISO 8859-1) en *TOL*.

Aunque los criterios para elegir los nombres de las variables pueden ser muy variados a continuación incorporamos algunas recomendaciones:

- Comenzar los nombres con minúscula, reservando los nombres con mayúscula para las variables globales.
- Utilizar el estilo *CamelCase* para unir palabras (cada nueva palabra empieza con mayúscula) o usar alguno de los separadores válidos: el punto (`.`) o el guión bajo (_).
- Evitar las abreviaturas, en la medida de lo posible.
- Utilizar un único idioma para confeccionar los nombres, preferiblemente el inglés o el idioma local, y ser cuidadosos con la ortografía.
- Ser coherentes con los criterios elegidos.

Reasignación (`:=`)

La mayor parte de las variables en *TOL* admiten la reasignación, es decir el cambio de su valor. Para ello hemos de usar el operador `:=` (dos puntos igual).

Por ejemplo:

```
Real a = 1;
Real a := 2; // Se reasigna 'a'
```

Si usamos el operador de asignación (=) obtendremos un error:

```
Real a = 1;
Real a = 2;
```

```
ERROR: [1] Variable 'a' ya definida como "a "

No se ha podido crear la variable "Real a".

ERROR: [2] Conflicto entre variables.
Se ha intentado modificar "a" a través de la variable "a"
```

Asignación por valor

Nótese que la asignación de la mayor parte de gramáticas en *TOL* es *por valor*, de modo que cada asignación construye una nueva variable.

Por ejemplo:

```
Real a = 1;  // se crea 'a' con el valor 1
Real b = a;  // se crea 'b' con el valor de 'a' que es 1
```

```
Real a := 2; // se cambia el valor de 'a' por 2
Real b;      // pero 'b' sigue valiendo 1
```

2.1.3 Números (`Real`)

TOL dispone de un tipo de datos, la gramática `Real`, para manejar todo tipo de números reales, y por tanto no distingue los números enteros de los números con coma flotante.

Por ejemplo:

```
Real a = 1;
Real b = 0.25;
Real c = Sqrt(3);
Real d = -1/2;
```

Valor desconocido (`?`)

TOL dispone de un valor real especial, que es el valor desconocido y que se indica usando el símbolo de interrogación (`?`).

```
Real n = ?;
```

Este es también el valor que ofrecen algunas funciones matemáticas cuando el valor de respuesta no existe o no pertenece a los números reales:

```
Real Sqrt(-1);
Real Log(0);
Real ASin(3);
Real 0/0;
```

Otros reales

TOL también tiene implementados algunos valores reales especiales como:

- El número e: `Real E (2.718281828459045)`
- El número pi (π): `Real Pi (3.141592653589793)`
- El valor infinito: `Real Inf (1/0)`
- El valor lógico verdadero: `Real True (1)`
- El valor lógico falso: `Real False (0)`
- El tiempo transcurrido: `Real Time (valor variable)`

Operadores y funciones matemáticas

TOL incorpora además de los habituales operadores aritméticos y lógicos, un surtido conjunto de funciones matemáticas y estadísticas. A continuación mostramos algunas de ellas mediante algunas líneas de ejemplo. Para más detalles consúltese la documentación de las funciones.

Ejemplos:

```
// Generamos un número aleatorio entre 0 y 1000
Real x = Rand(0, 1000);
// Encontramos su parte entera y su parte decimal:
Real n = Floor(x);
Real d = x-n;

// Es bastante común usar la siguiente aproximación:
//   Log(1+x) ~ x
// cuando 'x' es pequeña.
// Calculamos cual es el error al aplicarla sobre la parte decimal:
Real log_1_plus_d = Log(1+d);
Real relative_error = Abs(log_1_plus_d - d)/log_1_plus_d * 100; // en %
```

```
// Encuentro mi propia versión de pi
// aprovechando que la tangente de pi/4 es 1
Real my_pi = 4 * ATan(1);
```

Números complejos (`Complex`)

TOL también incorpora un tipo de datos capaz de manejar números complejos, la gramática `Complex`.

La declaración de un número complejo se hace en su forma binómica: como suma de una parte real y una parte imaginaria que es producto de un real y la unidad imaginaria elemental: `Complex i`.

Por ejemplo:

```
// Se puede crear un número complejo indicando sus partes real e imaginaria:
Complex z = 3 - 2*i;
```

Para recuperar la parte real e imaginaria de un número complejo, así como para obtener el módulo y argumento de su notación polar disponemos de las funciones:

- La función `CReal` para la parte real.
- La función `CImag` para la parte imaginaria.
- La función `CAbs` para el valor absoluto o módulo del número complejo.
- La función `CArg` para el argumento o fase del número complejo.

Por ejemplo:

```
// Sea el número complejo:
Complex z = 3 + 4*i;
// La parte real y la imaginaria pueden obtenerse con:
Real z_re = CReal(z); // -> 3
Real z_im = CImag(z); // -> 4
// El módulo y el argumento (de la notación polar) con:
Real z_mod = CAbs(z); // -> 5
Real z_arg = CArg(z); // -> 0.9272952180016122
```

Además de las operaciones aritméticas convencionales, disponemos del operador ~ (virgulilla) que nos permite obtener el número complejo conjugado de un número dado. Por ejemplo:

```
Complex z = 3 + 2*i;
Complex u = z / CAbs(z); // (0.832050294337844)+i*(0.554700196225229)
Complex uC = ~u;         // (0.832050294337844)+i*(-0.554700196225229)
```

Nótese que algunas de las funciones, que sobre números reales devolvían valores desconocidos, tienen una implementación cuya salida está en el dominio de los números complejos. Por ejemplo:

```
Complex Sqrt(-1);  // (0)+i*(1)
Complex Log(-0.5); // (-0.693147180559945)+i*(3.14159265358979)
```

2.1.4 Cadenas de texto (`Text`)

La gramática `Text` nos permite crear variables con cadenas de texto de longitud indefinida. La cadena de texto ha de encerrarse entre comillas dobles (`""`).

Para introducir en la propia cadena las comillas dobles (`"`) y algunos otros caracteres especiales se utiliza la barra invertida (`\`) como carácter de escape.

Comillas dobles	" (ASCII 34)	\"	
Barra invertida	\ (ASCII 92)	\\	Se acepta simple cuando no hay ambigüedad.
Salto de línea	(ASCII 13)	\n	También se puede introducir explícitamente.
Tabulación	(ASCII 9)	\t	También se puede introducir explícitamente.

Tabla 2.1.3: Caracteres escapados en la cadenas de texto en *TOL*.

Por ejemplo:

```
Text "La palabra \"ayuntamiento\" contiene todas las vocales";
Text "Hay dos tipos de letras:\n * Las mayúsculas.\n * Las minúsculas.";
```

Nótese que esta última cadena puede escribirse también como:

```
Text "Hay dos tipos de letras:
 * Las mayúsculas.
 * Las minúsculas.";
```

Mensajes (`WriteLn`)

Una función muy utilizada al programar y un tanto particular en su sintaxis (ya que no necesita devolver ninguna variable) es aquélla que nos permite escribir mensajes en la salida del intérprete. Está disponible en dos formatos: `Write` y `WriteLn`. Esta última, mucho más extendida, añade un salto de línea al final del mensaje sin tener que incluirlo explícitamente.

```
WriteLn("Hola mundo");
```

```
Hola mundo
```

Operaciones con cadenas

A menudo deseamos modificar o editar una cadena de texto. A veces incluso encontramos cadenas que albergan determinada información que deseamos obtener. Para ello *TOL* dispone de algunas funciones que facilitan estas tareas.

A continuación ilustramos algunas de ellas mediante un ejemplo. Para más detalles consúltese la documentación de las funciones.

Ejemplo:

```
// Disponemos de una cadena que contiene el valor de un atributo
// y queremos extraerlo.
// El valor que nos interesa se encuentra tras el carácter ':'
Text string = "language:es";
// Localizamos la posición del caracter ':'
Real position = TextFind(string, ":");
// Determinamos la longitud de la cadena
Real length = TextLength(string);
// Obtenemos el valor de la cadena buscado
Text language = Sub(string, position+1, length);
// Imprimimos por la salida el valor encontrado
WriteLn("Configuración de idioma: '"+language+"'.");
```

```
Configuración de idioma: 'es'.
```

2.1.5 Funciones (`Code`)

Las funciones en *TOL* son un nuevo tipo de variables, cuya gramática es `Code`.

La sintaxis para la definición de una función es:

- La gramática de la salida. En *TOL*, toda función tiene que devolver una variable. Si no se desea retornar una salida o ésta carece de interés se recomienda devolver un número real.
- El nombre de la función. Éste está sometido a las mismas reglas que el nombre de una variable en general. Al igual que con el resto de variables el nombre es opcional.
- Los argumentos. Entre paréntesis (`()`) se indican los argumentos separados por comas. Para cada argumento hemos de indicar su gramática y el nombre con el que nos referiremos a él en el cuerpo de la función. Toda función ha de recibir un argumento como mínimo. Si no se necesita indicar un argumento, se recomienda usar un argumento de tipo `Real` que no se use en el cuerpo de la función.
- El cuerpo. Entre llaves (`{}`) se indican las distintas sentencias de código de la función. La última línea, que excepcionalmente no necesita tiparse (anteponerle su gramática) y que no lleva punto y coma (`;`) será la salida de la función.

```
<Grammar> [<Name>](<Grammar> <name>[, <Grammar> <name>, ...]) {
  ...
};
```

En la expresión anterior, los paréntesis angulares (<>) indican que ese código ha de ser sustituido convenientemente y los corchetes (`[]`) indican que el código es opcional.

Ejemplos:

```
// Creamos una función que devuelve el número de dígitos
// de la parte entera de un número real
Real IntegerPart_Length(Real number) {
  Real unsignedIntegerPart = Floor(Abs(number));
  // determinamos el número de cifras de la parte entera
  // haciendo uso de que los números están en base decimal
  Floor(Log10(unsignedIntegerPart))+1
};
```

```
// Esta función imprime un saludo cada vez que es llamada.
// No necesita argumentos ni su salida tiene interés.
Real PrintHello(Real void) {
  WriteLn("Hello!");
1};
Real PrintHello(?); // se usa ? para subrayar que su valor no importa
```

```
Hello!
```

Nótese que la indentación del código no es parte de la sintaxis del lenguaje aunque sí una manera práctica y clara de declarar las funciones.

Ámbito local ({})

Téngase en cuenta que todo el código entre llaves (`{}`) como el del cuerpo de una función, se evalúa en un espacio de definición temporal que se destruye (salvo la salida o última línea) al completar la evaluación del bloque. En el ámbito del bloque se tiene acceso tanto a las variables globales como a las de otros niveles superiores, del mismo modo que se accede a las variables locales, teniendo preferencia estas últimas en caso de ambigüedad.

Por ejemplo:

```
Text scope = "Global";
Real value = {
  Text scope = "Local";
  WriteLn("Scope 1: "<<scope);
```

```
  Real phi = Rand(0, Pi);
  Sin(phi)^2 + Cos(phi)^2
}; // -> 1
WriteLn("Scope 2: "<<scope);
```

```
Scope 1: Local
Scope 2: Global
```

2.2 Conjuntos

2.2.1 Conjuntos (`Set`)

Una gramática especialmente útil es la que permite crear conjuntos. En *TOL* el tipo de variable `Set` representa a un conjunto ordenado de otras variables (incluyendo otros conjuntos). Los elementos del conjunto pueden ser de diferentes gramáticas y pueden tener nombre o no.

Para crear un conjunto se utilizan los dobles corchetes (`[[` y `]]`) o alguna función que lo devuelva.

Por ejemplo:

```
Set digits = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]];
Set even_numbers = Range(2, 20, 2);
Set vowels = SetOfText("a", "e", "i", "o", "u");
```

Nótese que la sintaxis para la creación de los conjuntos sigue siendo: gramática, nombre, signo igual y contenido, y el término `Set`, hace referencia al tipo de objeto: un conjunto (en ingles *set*) y no a la acción de asignar (también *set* en inglés).

Para acceder a los elementos del conjunto se utilizan los corchetes simples (`[` y `]`) o la función `Element` indicando el índice (o el nombre si tiene) del elemento deseado.

Por ejemplo:

```
Set colors = [["red", "green", "blue"]];
Text first_color = colors[1];
```

Operaciones elementales

Las operaciones más elementales sobre conjuntos están relacionadas con la capacidad de contar, aumentar y reducir su número de elementos. A continuación enumeramos las funciones que nos permiten hacer esto:

- La función `Card` nos permite conocer el tamaño o cardinal del conjunto, es decir, su número de elementos.
- La función `Concat` (como el operador <<) nos permite crear un nuevo conjunto concatenando (poniendo uno a continuación del otro) dos o más conjuntos.
- La función `Append` nos permite añadir a un conjunto dado un conjunto de nuevos elementos.
- La función `Remove` nos permite eliminar un elemento de un conjunto.

Por ejemplo:

```
// Creamos dos conjuntos, 's1' y 's2'
Set s1 = [["A", "E", "I", "O", "U"]];
Set s2 = [["Y", "W"]];
// Creamos un conjunto 's' con los elementos de 's1' y 's2'
Set s = s1 << s2; // equi
// Borramos el último elemento de 's'
```

```
Real s_size = Card(s);
Set Remove(s, s_size);
// Añadimos dos nuevos elementos a 's'
Set Append(s, [["J", "K"]]);
Real s_size := Card(s);
WriteLn("Tamaño final de 's': "<<s_size);
```

```
Tamaño final de 's': 8
```

Conjunto vacío (`Empty`)

La sintaxis de los dobles corchetes no nos permite crear un conjunto vacío, para ello es necesario hacer una copia del conjunto especial `Empty`.

```
Set my_elements = Copy(Empty);
```

Asignación por referencia

Téngase en cuenta que la asignación de conjuntos, a diferencia de lo que ocurre con reales y textos, es por referencia, es decir, el nuevo conjunto no es una copia, sino otra referencia o alias para acceder al mismo conjunto.

Por ejemplo:

```
Set a = [[1, 2]];  // se crea 'a' con los números 1 y 2
Set b = a;         // se crea 'b' que es una referencia al mismo conjunto 'a'
Set a := [[3, 4]]; // se cambia el contenido de 'a' por los números 3 y 4
Real b[1];         // los valores de 'b' también cambiaron, b[1] es 3
```

Copia (`Copy`)

Para forzar la copia y la creación de un nuevo conjunto, existe la función `Copy`:

```
Set a = [[1, 2]];  // se crea 'a' con los números 1 y 2
Set b = Copy(a);   // se crea 'b' como una copia del conjunto 'a'
Set a := [[3, 4]]; // si se cambia el contenido de 'a'
Real b[1];         // el conjunto 'b' se mantiene y b[1] es 1
```

Sin embargo, aunque la función Copy nos permite la creación de un nuevo conjunto, los elementos de uno y otro conjunto son los mismos. Si los elementos del conjunto original cambian, los del conjunto copia también:

```
Set a = [[1, 2]];  // se crea 'a' con los números 1 y 2
Set b = Copy(a);   // se crea 'b' como una copia del conjunto 'a'
Real a[1] := 3;    // se se cambia el valor de 'a[1]'
Real b[1];         // el valor de 'b[1]' también cambia, ahora es 3
```

Copia completa (`DeepCopy`)

Para conseguir hacer la copia completa de un conjunto, tanto del conjunto en sí como de sus elementos, disponemos de la función `DeepCopy`:

```
Set a = [[1, 2]];    // se crea 'a' con los números 1 y 2
Set b = DeepCopy(a); // se crea 'b' como una copia completa del conjunto 'a'
Real a[1] := 3;      // si se cambia el valor de 'a[1]'
Real b[1];           // el valor de 'b[1]' se mantiene, sigue siendo 1
```

<u>Resumen:</u>

En la tabla 2.2.1 se presenta de manera resumida el proceder interno de *TOL* en los casos descritos anteriormente:

`Set a = [[1, 2]]`	Se crea el `Real 1`. Se crea el `Real 2`. Se crea el conjunto `[[]]` y se añaden los dos reales. Se crea la referencia '`a`' que accede al conjunto.
`Set b = a;`	Se crea la referencia '`b`' que accede al mismo conjunto que '`a`'.
`Set c = Copy(a);`	Se crea un nuevo conjunto `[[]]` y se añaden los elementos de '`a`'. Se crea la referencia '`b`' que accede al nuevo conjunto.
`Set d = DeepCopy(a);`	Se crea un nuevo `Real 1` Se crea un nuevo `Real 2` Se crea un nuevo conjunto `[[]]` y se añaden estos reales. Se crea la referencia '`b`' que accede al nuevo conjunto.

Tabla 2.2.1: Proceder interno de *TOL* en la creación de conjuntos.

Características de los conjuntos

La flexibilidad de *TOL* a la hora de definir los conjuntos nos permite utilizarlos con propósitos muy diferentes. Podemos identificar diferentes tipos de conjuntos según algunas de sus características. Según éstas, los conjuntos pueden ser:

- **Ordenados:** Los elementos de un conjunto en programación, por su naturaleza, siempre tienen un orden. Con esta característica sin embargo queremos destacar de un conjunto si el orden es importante y afecta a su funcionalidad.
- **Indexados:** Esta característica está bastante relacionada con el orden, pues ambas nos sirven para localizar un elemento en el conjunto. Para que un conjunto esté indexado, todos sus elementos tienen que tener un nombre y que éste sea único. La indexación consiste en la construcción de un índice que asocia los nombres de los elementos con sus posiciones y facilita el acceso a los elementos por su nombre.
- **Sencillos o sin elementos repetidos:** Si bien habitualmente al hablar de conjuntos se piensa en colecciones de elementos distintos, en este documento admitiremos que un conjunto puede presentar elementos repetidos, y diremos que un conjunto es sencillo para destacar la ausencia de estos elementos duplicados.
- **Homogéneos:** Diremos que un conjunto es homogéneo cuando todos sus elementos son de la misma naturaleza o gramática.

Álgebra de conjuntos

Además de las operaciones elementales con conjuntos, TOL incorpora las operaciones básicas del álgebra de conjuntos:

- El operador `+` para la unión de conjuntos.
- El operador `*` para la intersección de conjuntos.
- El operador `-` para la diferencia de conjuntos.
- El operador `<:` para comprobar la pertenencia de un elemento a un conjunto.

Nótese que estas operaciones están definidas naturalmente para conjuntos sencillos (sin repetidos). Para obtener un conjunto libre de elementos repetidos, disponemos de la función `Unique`.

Ejemplo:

```
// Consideramos los siguientes conjuntos de caracteres
Set c1 = Characters("conjunto"); // (card: 8)
Set c2 = Characters("disjunto"); // (card: 8)
// Eliminamos los caracteres repetidos
Set s1 = Unique(c1); // se pierde una 'n' y una 'o' (card: 6)
Set s2 = Unique(c2); // se queda igual (card: 8)
// Operamos con los dos conjuntos
Set union = s1 + s2;         // (card: 9)
Set intersection = s1 * s2; // (card: 5)
Set s1_minus_s2 = s1 - s2;  // (card: 1)
Set s2_minus_s1 = s2 - s1;  // (card: 3)
```

Conjuntos indexados

Para indexar un conjunto y agilizar el acceso por nombre a sus elementos, disponemos de la función `SetIndexByName`. Todos los elementos del conjunto a indexar tienen que tener nombre y éste ser único en el conjunto.

Junto a esta función disponemos de otras funciones como `HasIndexByName`, que permite determinar si un conjunto está indexado y `FindIndexByName` que permite encontrar el índice o posición de un elemento en un conjunto por su nombre.

Recuérdese que también puede accederse a los elementos del conjunto por nombre usando los corchetes simples (`[]`).

Ejemplo:

```
// Creamos un conjunto con valores para cada día de la semana
// e indexamos el conjunto:
Set weekdays = [[
  Real monday = 101;
  Real tuesday = 102;
  Real wednesday = 103;
  Real thursday = 104;
  Real friday = 105;
  Real saturday = 106;
  Real sunday = 107
]];
Real SetIndexByName(weekdays);
// Encontramos el valor para el viernes
Real weekdays["friday"]; // -> 105
// Encontramos el valor para el sábado
Real sunday_index = FindIndexByName(weekdays, "sunday"); // -> 7
Real sunday_value = weekdays[sunday_index]; // -> 107
```

2.2.2 Instrucciones de control

TOL dispone de algunas funciones o instrucciones de control que nos permiten modificar el flujo de la ejecución de las líneas del programa. A continuación describimos las más importantes:

Instrucción condicional (`If`)

Una de las instrucciones de control más elementales de un lenguaje es el `If`, ya que nos permite la evaluación de un código u otro según sea el valor de una determinada condición.

En *TOL* el `If` se implementa como una función de tres argumentos (el tercero opcional) con la siguiente sintaxis.

```
Anything If(Real condition, Anything code_then[, Anything code_else])
```

El primer argumento es la condición: una expresión que ha de devolver un valor real verdadero (1) o falso (0). Los dos argumentos siguientes corresponden con el código que se ejecutará en cada caso respectivamente.

La salida del `If`, como en el caso de la definición de las funciones, corresponde con la última línea del código ejecutado.

Si el valor de la condición es un número distinto de cero se considerará verdadero, salvo si tiene el valor desconocido que se mostrará una advertencia y se devolverá un valor por defecto.

Por ejemplo:

```
Real If(3+4==6, 1, 0);
```

Nótese que si el tercer argumento no se indica y no se verifica la condición, no se construye ninguna salida. Esto no es un problema salvo que utilicemos la salida del `If`, por ejemplo asignándola a una variable.

```
Real answer = If(3==4, 5);
```

```
ERROR: [1] answer no se pudo crear.
```

Operaciones lógicas

Para la construcción del valor de condición, disponemos de las siguientes funciones y operadores lógicos:

- La función `And` y el operador `&` que devuelven verdadero, sólo si todos los argumentos son verdaderos.
- La función `Or` y el operador `|` que devuelven verdadero, cuando al menos un argumento es verdadero.
- La función `Not` y el operador `!` que devuelve lo contrario del valor sobre el que se aplican.

Y de las siguientes funciones y operadores de comparación:

- La función `Eq` y el operador `==` que devuelven verdadero si los argumentos son iguales.
- La función `NE` y el operador `!=` que devuelven verdadero si los argumentos no sin iguales.
- La función `LT` y el operador `<` que devuelven verdadero cuando el primer argumento es menor que el segundo.
- La función `GT` y el operador `>` que devuelven verdadero cuando el primer argumento es mayor que el segundo.
- La función `LE` y el operador `<=` que devuelven verdadero cuando el primer argumento es menor o igual que el segundo.
- La función `GE` y el operador `>=` que devuelven verdadero cuando el primer argumento es mayor o igual que el segundo.

A la hora de construir expresiones lógicas es interesante recordar cómo al negar una expresión se sustituyen las operaciones por sus contrarias. Los pares son: `And/Or`, `Eq/NE`, `LT/GE` y `GT/LE`.

<u>Por ejemplo:</u>

```
// Dado tres números:
Real a = Round(Rand(0, 1));
Real b = Rand(0, 6);
Real c = Rand(0, 9);
// y la siguiente condición:
Real condition = ( a==1 & (b>3 | c<=5) );
// El valor lógico contrario a éste:
Real Not(condition);
// puede también reescribirse como:
Real a!=1 | (b<=3 & c>5);
```

Instrucción condicional múltiple (`Case`)

Una extensión de la instrucción de control condicional para más de un caso es la función `Case`, ésta nos permite evaluar diferentes códigos según se verifique una u otra condición. Las comprobaciones se van evaluando secuencialmente, cuando se cumple una condición se ejecuta el código correspondiente y se interrumpe el resto de comprobaciones.

La función `Case` ha de recibir siempre un número par de argumentos, los argumentos en las posiciones impares devuelven el valor lógico de la condición: verdadero o falso, y los de las posiciones pares, el código que se ejecutará en el caso de que se cumpla.

La sintaxis es:

```
Anything Case(Real condition1, Anything code1
  [, Real condition2, Anything code2 [, ...]]);
```

Bucle condicional (`While`)

Otra de las instrucciones de control más elementales de *TOL* es el `While`, que nos permite la ejecución cíclica de un determinado código mientras se verifique una condición.

La sintaxis del código es:

```
Anything While(Real condition, Anything code);
```

<u>Ejemplo:</u>

```
// Se crea un ciclo que imprima por la salida números enteros
// hasta que se encuentre un cero
// o se hayan impreso al menos 100 números
Real print = 0;
Real rand_number = Floor(Rand(0, 100));
Real While(print<100 & rand_number!=0, {
  WriteLn(FormatReal(rand_number, "%2.0lf"));
  Real print := print + 1; // se ha impreso otro
  Real rand_number := Floor(Rand(0, 100)) // se busca otro entero
});
```

Argumentos condicionales

Nótese que las funciones de control `If`, `Case` y `While` reciben argumentos que son líneas de código que sólo se ejecutarán si la condición de control se satisface.

Otras funciones útiles para crear ciclos de ejecución de código, pero que no utilizan la estructura *condición-código*, son el `For` y el `EvalSet`.

Bucle sencillo (`For`)

La función `For` nos permite evaluar una función incrementalmente sobre un conjunto de números enteros. La sintaxis de la función es:

```
Set For(Real begin, Real end, Code action);
```

Los argumentos `begin` y `end` son dos números enteros que indican el principio y fin del ciclo.

El tercer argumento ha de ser una función de un sólo argumento de tipo `Real`:

```
Anything (Real integer) { ... }
```

A diferencia de la implementación de este bucle en otros lenguajes, en *TOL* es una función que devuelve un conjunto con tantos elementos como ciclos, siendo cada elemento la salida de la llamada correspondiente a la función que se indica como tercer argumento.

Por ejemplo:

```
// El siguiente código construye un conjunto con cinco conjuntos
// cada uno de éstos con un número entero y su cuadrado.
Set table = For(1, 5, Set (Real n) {
  [[n, n^2]]
});
```

Nótese que a diferencia de las instrucciones de control anteriores, el tercer argumento del `For` es una función que puede haber sido declarada previamente. Para entender esto mejor, véase la siguiente alternativa al código anterior:

```
Set function(Real n) { [[n, n^2]] };
Set table = For(1, 5, function);
```

Nótese también que la implementación del `For` en *TOL* como función que devuelve un `Set` facilita la construcción de conjuntos de datos que en otros lenguajes suelen implementarse de un modo similar al siguiente:

```
Set table = Copy(Empty);
Set For(1, 5, Real (Real n) {
  Set table := table << [[ [[n, n^2]] ]]
  1
});
```

Bucle sobre un conjunto (`EvalSet`)

El `EvalSet` es otra instrucción de control que nos permite aplicar una función a todos los elementos de un conjunto (`Set`). Como en el caso del `For`, la función devuelve un conjunto con todas las respuestas de la función al actuar sobre cada uno de los elementos.

Ejemplo:

```
// Obtenemos un conjunto con los índices en el ASCII
// de los caracteres de una frase:
Set chars = Characters("Hola mundo");
Set asciis = EvalSet(chars, Real (Text t) { ASCII(t) });
```

Nótese que como la función `ASCII` es una función que recibe (sin ambigüedad) como único argumento un texto (que es el tipo de datos de los elementos del conjunto) puede utilizarse sin definir una nueva función:

```
Set asciis = EvalSet(Characters("Hola mundo"), ASCII);
```

La función complementaria a `ASCII`, que devuelve el carácter a partir de su número ASCII es `Char`.

Una idea:

Una manera de ampliar las posibilidades que nos ofrece la función `For` a otro conjunto de números es utilizar un `EvalSet`, junto a una función `Range`:

```
Set countdown = EvalSet(Range(10, 0, -1), Real (Real n) {
  WriteLn(FormatReal(n));
  Sleep(1)
});
```

2.2.3 Consultas sobre conjuntos

Además de las operaciones sobre conjuntos mencionadas en las secciones anteriores, *TOL* dispone de funciones para la selección, ordenación y clasificación de conjuntos.

Nótese que estas funciones nos permiten, en la medida su alcance, realizar acciones de consulta sobre los conjuntos que se asemejan a las consultas de otros lenguajes, como por ejemplo, el comando `SELECT` y las cláusulas `ORDER BY` y `GROUP BY` del *SQL*.

Selección (`Select`)

La función `Select` nos permite seleccionar los elementos de un conjunto que verifican una determinada condición. La función recibe el conjunto y la función de condición que se ha de aplicar a cada uno de sus elementos para su selección y devuelve el conjunto con los elementos seleccionados. La sintaxis de la función es:

```
Set Select(Set set, Code condition_function);
```

Donde la condición (`condition_function`) ha de ser una función de un sólo argumento del mismo tipo que los elementos del conjunto (en general `Anything`) y cuya salida ha de ser un valor verdadero o falso que indique si el elemento debe ser seleccionado o no:

```
Real <True|False> (Anything element) { ... }
```

Ejemplo:

```
// Disponemos de un conjunto de números aleatorios distribuidos normalmente:
Set sample = For(1, 1000, Real (Real i) { Gaussian(0, 1) });
// y seleccionamos sólo los que son mayores que -1
Set subsample = Select(sample, Real (Real x) { x > (Real -1) });
```

Ordenación (`Sort`)

La función `Sort` nos permite ordenar los elementos de un conjunto, de acuerdo a una determinada relación de orden. La sintaxis de la función es:

```
Set Sort(Set set, Code order_function);
```

Donde el criterio de ordenación (`order_function`) ha de ser una función de dos argumentos del mismo tipo que los elementos del conjunto (en general `Anything`) y cuya salida ha de ser un valor real que indique cual de los dos ha de ir primero: `−1` si ha de ser el primer argumento, `1` si ha de ser el segundo o `0` si es indiferente de acuerdo al criterio de ordenación:

```
Real <-1|1|0> (Anything element1, Anything element2) { ... }
```

TOL dispone de una función que permite comparar y por tanto ordenar los distintos tipos de datos, se trata de la función `Compare`, aunque como es natural podemos definir nuestro propio criterio de ordenación.

Ejemplos:

```
// Sea el conjunto de las letras de la palabra "conjunto"
Set chars = Characters("conjunto");
// Podemos ordenarlas alfabéticamente como:
Set sorted = Sort(chars, Compare); // [["c","j","n","n","o","o","t","u"]]
```

```
// Sea el conjunto de pares nombre-valor:
Set pairs = [[
  [["alpha", 2.3]],
  [["beta", Real -0.5]],
  [["gamma", 0.4]],
  [["delta", 8]]
]];
// Podemos ordenarlos de mayor a menor según su valor del siguiente modo:
Set Sort(pairs, Real (Set pair1, Set pair2) {
  (-1) * Compare(pair1[2], pair2[2])
});
```

Clasificación (`Classify`)

La función `Classify`, nos permite agrupar los elementos de un conjunto en nuevos conjuntos (o clases) de acuerdo a una determinada relación de equivalencia.

Por defecto, la función de clasificación (`Classify`) espera como segundo argumento una relación de orden como la de la función de ordenación (`Sort`) que se utilizará como relación de equivalencia, agrupando en el mismo conjunto, todos los elementos que devuelvan `0` al compararlos entre sí.

Por ejemplo:

```
// Sea el conjunto de las letras de la palabra "conjunto"
Set chars = Characters("conjunto");
// Si las agrupamos usando la función Compare encontraremos 6 grupos.
// Dos de ellos (el de las 'o' y el de las 'n') con dos elementos.
Set groups = Classify(chars, Compare);
Set EvalSet(groups, Real (Set group) {
  Text firstElement = group[1];
  Real size = Card(group);
  WriteLn("El grupo de las '"<<firstElement<<"' tiene "<<size<<" elementos");
  Real 0
});
```

```
El grupo de las 'c' tiene 1 elementos.
El grupo de las 'j' tiene 1 elementos.
El grupo de las 'n' tiene 2 elementos.
El grupo de las 'o' tiene 2 elementos.
El grupo de las 't' tiene 1 elementos.
El grupo de las 'u' tiene 1 elementos.
```

La función nos permite indicar opcionalmente el tipo de relación utilizada para la clasificación. La sintaxis completa de la función es:

```
Set Classify(Set set, Code function[, Text relationType="partial order"])
```

Por ejemplo, si queremos ordenar un conjunto de la forma más natural posible, utilizando una relación de equivalencia que devuelva si dos elementos son iguales o no, indicaremos `"equivalence"` como tercer argumento. Por ejemplo:

```
// Consideremos un conjunto de números aleatorios naturales:
Set numbers = For(1, 100, Real (Real i) { Floor(Rand(100, 1000)) });
// Y lo clasificamos por grupos según sea su última cifra:
Set groups = Classify(numbers, Real (Real number1, Real number2) {
  Real unit1 = number1 % 10; // última cifra del primer número
  Real unit2 = number2 % 10; // última cifra del segundo
  unit1==unit2
}, "equivalence");
```

2.2.4 Estructuras (`Struct`)

Aunque un conjunto puede estar formado por elementos cualesquiera sin ninguna razón de ser en particular, podemos destacar dos tipos de conjuntos por su contenido:

- Conjuntos de tipo contenedor: que contienen un número indefinido de elementos de la misma naturaleza (homogéneos). Por ejemplo, son *contenedores* un conjunto de unidades de estudio, el conjunto de conexiones a bases de datos abiertas, o alguno de estos conjuntos sencillos:

```
// Los números impares menores que 100
Set container1 = Range(1, 99, 2);
// Las letras minúsculas del ASCII
Set container2 = For(97, 122, Text (Real ascii) { Char(ascii) });
```

- Conjuntos de tipo entidad: que contienen un número definido de elementos, por lo común con un orden y un significado concreto y no necesariamente de la misma naturaleza, que representan una unidad de un determinado concepto. Lo habitual es que puedan existir o crearse múltiples elementos con las mismas características o estructura. Ejemplos de *entidades* serían, las coordenadas de un elemento de una matriz (un par fila-columna), los argumentos de una función o algunos de estos conjuntos sencillos:

```
// Una regla de sustitución: el primer texto será sustituido por el segundo
Set entity1 = [["á", "a"]];
// Las características de una función: nombre, salida, número de argumentos
Set entity2 = [["Char", "Text", 1]];
```

Estructuras

Es bastante deseable que los conjuntos de tipo entidad sigan una estructura prefijada, de modo que no quepa duda del significado de cada elemento y se facilite su uso.

En *TOL*, este concepto de estructura para un conjunto viene dado por un tipo de definición especial denominada `Struct`. Las estructuras (entes de tipo `Struct`) son un tipo de información que desempeña el papel de una gramática de tipo `Set` especializada, y que nos permite la construcción de conjuntos estructurados.

Definición de estructuras

La definición de una estructura contiene el número de elementos y el tipo de los mismos, que tendrá cada conjunto que creemos con ella. Además del tipo de cada elemento en la definición de la estructura asignamos de un nombre con el que localizarlo sin ambigüedad.

Para definir una nueva estructura, hemos de indicar el nombre y gramática de cada campo usando la siguiente sintaxis:

```
Struct @<StructName> {
  <Grammar1> <Field1>;
  <Grammar2> <Field2>;
  ...
```

```
};
```

donde el código entre paréntesis angulares (<>) ha de sustituirse por su valor correspondiente.

Ejemplo:

```
// Definimos @Vector3D para representar vectores tridimensionales:
Struct @Vector3D {
  Real X;
  Real Y;
  Real Z
};
```

Nótese que los nombres de las estructuras han de comenzar con el carácter `@` (arroba) y así diferenciarlas claramente del resto de variables, ya que tienen un tratamiento sintáctico en ocasiones más parecido al de una gramática que al de una variable.

Conjuntos estructurados

Denominamos *conjuntos estructurados* a los objetos creados siguiendo la definición de una estructura. Como conjuntos que son, admiten el mismo tratamiento que el resto de conjuntos (variables de tipo `Set`).

Para la creación de un conjunto estructurado usamos el nombre de la estructura como si fuese una función con tantos argumentos como elementos tienen la estructura:

```
Set conjunto = @<StructName>(<value1>, <value2>, ...);
```

Por ejemplo:

Si para crear un conjunto con tres números reales, utilizamos:

```
Set set = [[1, 2, 3]];
```

Para crear un conjunto estructurado de tipo `@Vector3D`, hemos de hacer:

```
Set vector3D = @Vector3D(1, 2, 3);
```

También podemos poner una estructura a un conjunto ya creado y que concuerde con la definición de la estructura en número y tipo de sus elementos, a través de la función `PutStructure`. Por ejemplo:

```
Set set = [[1, 2, 3]];
Set PutStructure("@Vector3D", set);
```

Acceso a los elementos (->)

La estructuración de los conjuntos de tipo entidad, no sólo nos sirve para reforzar el carácter de este tipo de conjuntos y asegurar cierta coherencia con el número y tipos de sus elementos, sino que nos facilita el acceso por el nombre del campo.

Para acceder a los elementos por el nombre del campo usaremos el operador flecha (->) con la sintaxis:

```
<Grammar> element = structuredSet-><FieldName>;
```

o la función `Field` equivalente:

```
<Grammar> element = Field(structuredSet, "<FieldName>");
```

Por ejemplo:

```
Set vector = @Vector3D(4, 3, -2);
```

```
Real coordZ = vector->Z; // es equivalente a vector[3]
```

Nótese que el nombre de un elemento no ha de coincidir con su nombre en la estructura:

```
Real a = 5;
Set vector = @Vector3D(a, 1-a, 0);
WriteLn("El nombre de la coordenada X es: '"<<Name(vector->X)<<"'");
```

```
El nombre de la coordenada X es: 'a'
```

Información sobre las estructuras

Para conocer los campos definidos en una estructura disponemos de la función `StructFields` que devuelve un conjunto con las características de cada elemento. Por ejemplo:

```
Set StructFields("@Vector3D");
```

Nótese que esta función recibe como argumento un texto con el nombre de la estructura.

Para conocer si un conjunto está o no estructurado y cuál es el nombre de dicha estructura disponemos de la función `StructName`.

2.3 Estadística

TOL es un lenguaje de programación con una orientación fuertemente estadística, y como tal, incorpora numerosas herramientas para el análisis estadístico de datos, tanto desde el aspecto más descriptivo como desde el inferencial.

2.3.1 Estadística descriptiva

Para la obtención de estadísticos, *TOL* dispone de un conjunto de funciones que admiten un número indefinido de argumentos reales.

```
Real <Function>(Real x1, Real x2, ...);
```

Además de las funciones aritméticas `Sum` y `Prod`, que respectivamente suman y multiplican números reales, podemos destacar las siguientes funciones que nos permiten obtener:

- Promedios como la media (o media aritmética) (`Avr`), la media geométrica (`GeometricAvr`) o la media armónica (`HarmonicAvr`).
- Medidas de dispersión como la varianza (`Var`) o la desviación estándar (`StDs`).
- Medidas sobre cómo se distribuyen los datos: los valores máximo (`Max`) y mínimo (`Min`), la mediana (`Median`) o un cuantil con probabilidad `p` cualquiera:

```
Real Quantile(Real p, Real x1, Real x2, ...);
```

- Otras medidas relacionadas con los momentos de la distribución como el coeficiente de asimetría (`Asymmetry`) o el coeficiente de curtosis (`Kurtosis`), y de manera general cualquier momento de orden `n`, centrado (`CenterMoment`) o no (`Moment`):

```
Real Moment(Real n, Real x1, Real x2, ...);
Real CenterMoment(Real n, Real x1, Real x2, ...);
```

Datos ausentes (?)

Téngase en cuenta que los datos reales con valor desconocido (`?`) son tratados como *datos ausentes* por todas estas funciones estadísticas. Es decir, estos datos, no se tienen en cuenta, o se descartan en el cálculo.

Por ejemplo:

```
Real Sum(2, 3, ?, 4, ?); // -> 9
Real Avr(2, 3, ?, 4, ?); // -> 3
```

Estadísticos sobre conjuntos

Estos mismos estadísticos pueden obtenerse sobre conjuntos de números reales a través de las funciones correspondientes para conjuntos con el nombre Set<Statistic>.

Por ejemplo:

```
Set values = SetOfReal(1.0, -0.3, 0, 2.1,0.9);
Real mu = SetAvr(values);
Real sigma2 = SetVar(values);
```

2.3.2 Probabilidad

En teoría de la probabilidad, una variable aleatoria representa a una variable cuyos valores corresponden con realizaciones de un fenómeno aleatorio estocástico (no determinista). Estos valores podrían ser, por ejemplo, los posibles valores de un experimento aún no realizado o los de un valor actualmente existente pero desconocido.

Distribuciones de probabilidad

Aunque los valores de una variable aleatoria permanezcan indeterminados, sí que podemos conocer la probabilidad asociada a la ocurrencia de uno u otro valor. La relación que asigna a los diferentes valores su probabilidad es conocida como *distribución de probabilidad*.

Las variables aleatorias, así como sus distribuciones de probabilidad, pueden ser discretas o continuas, según si sus valores están restringidos a de un conjunto finito (o infinito numerable) o no.

Distribuciones discretas

Cada uno de los distintos valores que puede adoptar una variable aleatoria discreta tiene una determinada probabilidad asociada. La función que devuelve para cada valor esta probabilidad se conoce como *función de probabilidad*.

TOL incorpora las funciones de probabilidad de las principales distribuciones discretas, con la siguiente nomenclatura Prob<Distribution>:

- La distribución binomial: `ProbBinomial`.
- La distribución binomial negativa: `ProbNegBinomial`.
- La distribución Poisson: `ProbPoisson`.
- La distribución geométrica: `ProbGeometric`.
- La distribución hipergeométrica: `ProbHyperG`.
- La distribución uniforme discreta: `ProbDiscreteUniform`.

De un modo análogo podemos obtener la correspondiente *función de distribución*, definida como la probabilidad de que la variable tome un valor menor o igual que uno dado, siguiendo la nomenclatura: Dist<Distribution>.

La *inversa de la función de distribución*, que nos permite encontrar el valor para el cual la probabilidad acumulada es una dada, está implementado como: Dist<Distribution>Inv.

Distribuciones continuas

Los infinitos valores posibles de una variable aleatoria continua no tienen una probabilidad asociada, sino una densidad de probabilidad que nos permite determinar la probabilidad de que un valor se halle en un determinado intervalo.

Esta función conocida como *función de densidad* está implementada en *TOL* como `Dens<Distribution>` para las principales distribuciones de probabilidad continuas, entre las que destacamos:

- La distribución chi cuadrado: `DensChi`.
- La distribución exponencial: `DensExp`.
- La distribución t de Student: `DensT`.
- La distribución normal: `DensNormal`.
- La distribución log-normal: `DensLogNormal`.
- La distribución gamma: `DensGamma`.
- La distribución beta: `DensBeta`.
- La distribución F de Snedecor: `DensF`.
- La distribución uniforme (continua): `DensUniform`.

Como ocurría con las distribuciones discretas, *TOL* también ofrece las correspondientes funciones de distribución y sus inversas con la nomenclatura: `Dist<Distribution>` y `Dist<Distribution>Inv` respectivamente.

Números aleatorios

Para facilitar el muestreo de variables aleatorias, para las que su distribución de probabilidad es conocida, *TOL* incorpora funciones generadoras de números aleatorios.

Por ejemplo, la función `Rand` nos permite obtener números al azar entre dos números reales dados, se tratan de realizaciones de una distribución continua uniforme.

Para generar estas secuencias de números aleatorios se utiliza un generador de números pseudoaleatorios que depende de un valor semilla. Este valor, que se inicializa al comenzar una sesión *TOL*, puede consultarse y modificarse con las funciones `GetRandomSeed` y `PutRandomSeed` respectivamente.

Otra función muy común para obtener números aleatorios es la función `Gaussian` que nos permite obtener realizaciones de una distribución normal o gaussiana.

Además de las dos funciones ya mencionadas, *TOL* dispone de funciones para obtener muestras de otras distribuciones de probabilidad comunes, como la distribución chi cuadrado (`RandChisq`), la distribución exponencial, (`RandExp`), la distribución gamma (`RandGamma`) o la log-normal (`RandLogNormal`).

2.3.3 Matrices (`Matrix`)

Aunque *TOL* nos permite trabajar de manera general con conjuntos cualesquiera de números reales, introduce un nuevo tipo de variable, `Matrix`, para representar las matrices o conjuntos bidimensionales de números reales.

Las matrices se pueden construir, como una tabla de números siguiendo la siguiente sintaxis:

```
Matrix <m> = ((<m11>, <m12>, ...), (<m21>, <m22>, ...), ...);
```

Por ejemplo:

```
Matrix a = ((1, 2, -3), (4, -5, 6));
```

Téngase en cuenta que todas las filas han de tener el mismo número de columnas.

También podemos construir matrices fila (con una sola fila) o matrices columna (con una sola columna) mediante las funciones `Row` y `Col` respectivamente.

Por ejemplo:

```
Matrix row = Row(1, 2, -3);
Matrix column = Col(1, 4);
```

Elementos de una matriz

El número de filas y columnas de una matriz podemos obtenerlos a través de las funciones `Rows` y `Columns` respectivamente.

Para el acceso a los elementos de una matriz disponemos de la función `MatDat` que indicando fila y columna nos devuelve el valor del elemento. Para modificar el valor de un elemento utilizaremos la función `PutMatDat` indicando como tercer argumento el nuevo valor.

Composición de matrices

TOL incorpora dos operaciones que nos permiten componer matrices concatenando matrices por filas o por columnas:

- El operador << (y la función `ConcatRows`) para concatenar matrices por filas.
- El operador `|` (y la función `ConcaColumns`) para concatenar matrices por columnas.

Como es lógico, para concatenar por filas, las matrices han de tener el mismo número de columnas, y para concatenar por columnas, el mismo número de filas.

Por ejemplo:

```
Matrix a = ((1, 2), (4, 5));
Matrix b = Col(3, 6);
Matrix c = Row(7, 8, 9);
Matrix (a | b) << c;
```

Del mismo modo que podemos combinar matrices para componer una matriz mayor, podemos obtener submatrices a partir de una matriz dada. Para ello destacamos las siguientes funciones:

- La función `SubRow` que nos permite obtener una matriz formada por una selección de filas de otra matriz. Recibe como argumento la matriz original y un conjunto de índices.
- La función `SubCol` que obtiene una matriz a partir de una selección de columnas.
- La función `Sub` que nos permite obtener una submatriz indicando la fila y columna del primer elemento y el alto (número de filas)y ancho (número de columnas)de la submatriz.

Por ejemplo:

```
Matrix a = ((1, 2), (3, 4), (5, 6)),
Matrix SubRow(a, [[1, 3]]); // -> ((1, 2), (5, 6))
```

Operaciones con matrices

Además de los operadores aritméticos + y − que nos permiten sumar y restar dos matrices o una matriz y un real, *TOL* incorpora el operador `*` para realizar el producto de matrices.

Recuérdese que el producto de matrices, está definido para dos matrices rectangulares para las que coincide el número de columnas de la primera con el número de filas de la segunda.

Otras funciones características de las matrices implementadas en *TOL* son:

- La trasposición de matrices: `Tra`.
- La inversión de matrices mediante el método de reducción de Gauss: `GaussInverse`.
- La factorización de Cholesky para matrices simétricas y positivas. La función `Choleski` nos devuelve la matriz triangular inferior (o triángulo de Cholesky) de la descomposición.

Operaciones elemento a elemento

Para realizar el producto de matrices elemento a elemento (conocido como producto de Hadamard) o el cociente elemento a elemento, disponemos de:

- El operador `$*` (o la función `WeightProd`) para el producto.
- El operador `$/` (o la función `WeightQuotient`) para el cociente.

Nótese que el operador `^`, así como todo el conjunto de funciones matemáticas sobre número reales (exponenciales, trigonométricas, hiperbólicas, etc.) están implementadas sobre matrices en el sentido del producto de Hadamard, actuando individualmente sobre cada elemento.

Si bien la mayor parte de las funciones matemáticas sobre números reales disponen de su versión matricial, en general podemos aplicar una función cualquiera a todos los elementos de una matriz mediante la instrucción `EvalMat` (análoga a `EvalSet`) que recorre todas las filas o columnas de la matriz.

Por ejemplo:

```
// Aplicaremos la función f(x) = x * Exp(x)
// a los elementos de una matriz, de dos modos distintos:
Matrix a = ((1, 2, -5), (3, 0, 7));
Matrix b_1 = a $* Exp(a);
Matrix b_2 = EvalMat(a, Real (Real x) { x * Exp(x) });
```

También disponemos de una instrucción condicional sobre matrices `IfMat` que nos permite construir matrices con una estructura condicional.

Por ejemplo:

```
// Creamos una matriz aplicando el logaritmo a otra
// y devolviendo 0 si el elemento es menor o igual que 1.
Matrix a = ((1, 2, -5), (3, 0, 7));
Matrix b = IfMat(GT(a, 1), Log(a), 0);
```

Estadísticos sobre matrices

Los estadísticos introducidos en la sección 2.3.1 pueden obtenerse sobre matrices mediante las correspondientes funciones con el nombre `Mat<Statistic>`.

Por ejemplo:

```
Matrix a = Gaussian(100, 1, 0, 0.5);
Real a_mu = MatAvr(a);
Real a_sigma = MatStDs(a);
```

2.3.4 Modelos lineales

Los modelos estadísticos nos permiten describir una variable aleatoria como una función de otras variables. Esta relación no es determinista sino estocástica, en el sentido de que el modelo no es más que una descripción aproximada del mecanismo que ha generado las observaciones y que se presupone estocástico.

Regresión lineal (`LinReg`)

El modelo más sencillo que podemos utilizar para describir una variable aleatoria es aquel en el que ésta puede explicarse como una combinación lineal de otras variables más una componente puramente aleatoria o ruido blanco:

$$Y = \sum_i X_i \beta_i + E$$

donde Y es la variable observada que desea describirse (denominada *output* del modelo), X_i son las variables explicativas (o *inputs* del modelo), β_i son los parámetros y E es la parte no explicada o error del modelo.

En general, los parámetros que describen la relación entre las variables no son conocidos y necesitan estimarse utilizando las observaciones que se disponen de las variables.

En el caso del modelo lineal anterior, para obtener una estimación de los parámetros basta con resolver la regresión lineal correspondiente. Para ello, *TOL* dispone de la función `LinReg` que resuelve el modelo expresado matricialmente.

Por ejemplo:

```
// Construiremos una muestra de una variable aleatoria Y
// como la suma de dos variables X1 y X2 + un ruido E
Matrix X1 = Rand(10, 1, 0, 10);
Matrix X2 = Rand(10, 1, 2, 5);
Matrix E = Gaussian(10, 1, 0, 0.1);
Real beta1 = 0.5;
Real beta2 = -0.3;
Matrix Y = X1*beta1 + X2*beta2 + E;
// Estimamos el modelo: Y = X1*beta1 + X2*beta2
Set estimation = LinReg(Y, X1|X2);
Real estimated_beta1 = estimation["ParInf"][1]->Value;
Real estimated_beta2 = estimation["ParInf"][2]->Value;
```

Modelos lineales generalizados (`Logit`, `Probit`)

En el análisis de variables aleatorias, encontramos algunas de naturaleza discreta que no se dejan modelar naturalmente con un modelo lineal como el anterior.

Sin embargo, cuando los valores de esta variable se pueden hacer corresponder con los de unas determinadas distribuciones, el modelo puede reescribirse con la ayuda de una función de enlace (*link*) en lo que se conoce como modelo lineal generalizado:

$$\mathrm{E}(Y) = link^{-1}\left(\sum_i X_i\beta_i\right)$$

donde $\mathrm{E}(Y)$ expresa la esperanza de la variable aleatoria Y.

Cuando la variable aleatoria solo toma dos valores distintos (variable dicotómica o Bernouilli) disponemos de dos funciones de enlace (con dominio entre (0,1) e imagen en toda la recta real) para la construcción del modelo generalizado:

- La función *logit* que es la inversa de la función de distribución logística.
- La función *probit* que es la inversa de la función de distribución normal.

Para la resolución de los modelos lineales generalizados *logit* y *probit*, *TOL* incorpora los estimadores máximo-verosímiles `Logit` y `Probit` respectivamente, con una sintaxis similar a `LinReg`.

```
// Creamos una variable dicotómica Y a partir de una probabilidad pY
// construida aplicando la función de distribución normal
// a una combinación lineal de X1 y X2
Matrix X1 = Rand(500, 1, 0, 5);
Matrix X2 = Round(Rand(500, 1, 0, 1));
Real beta1 = 0.8;
Real beta2 = -0.5;
Matrix pY = EvalMat(X1*beta1 + X2*beta2, Real (Real x) { DistNormal(x) });
Matrix Y = EvalMat(pY, Real (Real x) { Real Rand(0,1)<x });
// Estimamos el modelo probit:  E(Y) = Probit(X1*beta1 + x2*beta2)
Set estimation = Probit(Y, X1|X2);
Real estimated_beta1 = MatDat(estimation[1], 1, 1);
Real estimated_beta2 = MatDat(estimation[1], 2, 1);
```

Configuración

Los estimadores anteriores (`Probit` y `Logit`) hacen uso de algunas variables de configuración, creadas como variables globales:

- El número máximo de iteraciones para los procesos iterativos: `Real MaxIter`.
- La tolerancia de los métodos numéricos: `Real Tolerance`.

2.3.5 Matrices virtuales (`VMatrix`)

Las matrices virtuales, gramática `VMatrix`, son un nuevo tipo de datos para la declaración de matrices, que encapsula el tratamiento de matrices especiales que no pueden tratarse de forma eficiente con el tipo `Matrix`, permitiendo formatos internos polimórficos especializados para distintos tipos de estructuras matriciales.

Las matrices virtuales en realidad engloban varios subtipos que se clasifican en virtud de los siguientes conceptos:

- El motor (*engine*): Cada motor de cálculo requiere sus propios tipos de datos *ad-hoc* para sacar el máximo partido de sus algoritmos. Se trata de incluir los principales sistemas de álgebra matricial para tratar los problemas más usuales de matrices densas, dispersas (o *sparse*), estructuradas (Toeplitz, Vandermonde, etc.) e incluso para poder definir matrices como operadores lineales genéricos. Los motores para los que existe interfaz de matriz virtual actualmente son: *BLAS&LAPACK* y *CHOLMOD*.

- El tipo de celda (*cell*): Inicialmente se ha implementado sólo el tipo de celda `Real` con doble precisión (64 bits) pero se podrían ampliar a precisión simple (32 bits) y alta precisión (80 bits) en los paquetes en que estén disponibles.
- El modo de almacenamiento (*store*): Cada motor de cálculo puede ofrecer distintas formas de almacenar los datos que definen una matriz en función de su estructura interna y del tipo de algoritmos que se ejecutarán sobre la misma.

Las operaciones realizables con matrices virtuales dependen de cada subtipo lo cual complica algo su uso, pero como contrapartida se tiene acceso a métodos altamente especializados y eficaces.

Podemos destacar los siguientes tipos de matrices virtuales.

Matrices densas (`Blas.R.Dense`)

La matriz de tipo `Blas.R.Dense` que incorpora el tipo de datos básico de *BLAS&LAPACK* en el formato nativo de *FORTRAN* en el que las celdas de cada columna son consecutivas, en contraposición al formato por filas considerado como nativo en *C/C++* y usado en el tipo `Matrix`.

Matrices dispersas (`Cholmod.R.Sparse`)

Las matrices dispersas (*sparse* en inglés) se definen para la resolución de sistemas de ecuaciones lineales de gran tamaño, en los que las matrices están formadas en gran parte por ceros.

La matriz virtual de tipo `Cholmod.R.Sparse` es el tipo fundamental de matrices dispersas de *CHOLMOD* y ofrece una amplia gama de operaciones implementadas de forma muy eficiente.

Otros dos tipos más específicos que se introducen junto a éste son:

- El tipo `Cholmod.R.Factor`: como una especialización utilizada para almacenar la factorización de Cholesky de una matriz dispersa de una forma especialmente eficaz para resolver los sistemas lineales asociados a dicha descomposición. Véanse las descripciones de las funciones `CholeskiFactor` y `CholeskiSolve`.
- El tipo `Cholmod.R.Triplet`: que se implementa a efectos de almacenamiento externo e interfaz con otros sistemas. Se trata simplemente de ternas fila-columna-valor que vinculan el valor de cada celda no nula al número de fila y columna al que corresponde, lo cual conjuga un ahorro importante de memoria con matrices suficientemente dispersas. Véanse las descripciones de las funciones `Triplet` y `VMat2Triplet`.

Operaciones con matrices virtuales

Las matrices virtuales, `VMatrix`, se implementan con una funcionalidad similar a la de las matrices, de modo que disponemos conjuntos de funciones análogos a los de las matrices (`Matrix`):

- Acceso y edición de las celdas (con `VMatDat` y `PutVMatDat`).
- Operaciones con submatrices (por ejemplo: `SubRow` o `ConcatColumns`).
- Operadores aritméticos (como el producto matricial `*` o el de Hadamard `$*`).
- Funciones matemáticas y lógicas (como `Log`, `Floor` o `LT`).
- Estadísticos sobre la matriz (como `VMatAvr` o `VMatMax`).

Nótese que en las funciones en las que aparece, el término `Mat` (de las funciones de la gramática `Matrix`) se sustituye por `VMat`.

Creación de matrices virtuales

El uso de matrices virtuales está vinculado al uso de determinadas funcionalidades, que las utilizan como argumentos de entrada o salida.

Para la construcción de matrices virtuales disponemos de algunas funciones constructoras elementales como: `Constant` o `Zeros` para matrices con todas las celdas iguales, `Eye` o `Diag` para matrices diagonales, o `Rand` o `Gaussian` para matrices con números aleatorios.

Además disponemos de los siguientes mecanismos de conversión entre matrices:

- De matrices a matrices virtuales: `Mat2VMat`.
- De matrices virtuales a matrices: `VMat2Mat`.
- Entre distintos tipos de matrices virtuales: `Convert`.

2.4 Variables temporales

2.4.1 Fechas (`Date`)

Una de las principales características que distinguen a *TOL* de otros lenguajes matemáticos interpretados es su conjunto de herramientas para la gestión de las estructuras temporales. No es casual que el nombre escogido para denominar al lenguaje sea *Time Oriented Language* (lenguaje orientado al tiempo).

Haber postergado la presentación de las variables temporales de *TOL* hasta esta sección, no pretende quitarles importancia, sino más bien al contrario concederles una sección propia y una exposición cuidada.

La principal gramática para la gestión de las estructuras temporales no es otra sino la fecha, `Date`, que nos sirve para representar los instantes e intervalos temporales.

Las fechas en *TOL* se expresan y declaran siguiendo el siguiente convenio:

```
Date y<year>[m<month:01>[d<day:01>]];
```

donde el código entre paréntesis angulares (<>) ha de sustituirse por los correspondientes valores numéricos, los corchetes (`[]`) indican código opcional y los dos puntos (`:`) preceden a los valores por defecto. El año ha de indicarse con cuatro cifras, mientras que el mes y el día se indican con dos.

<u>Ejemplos:</u>

```
Date y1992m07d25; // -> 25/07/1992
Date y2000;       // -> 01/01/2000
Date y2007m03;    // -> 01/03/2007
```

Aunque, lo más habitual sea utilizar las fechas para indicar días o intervalos temporales superiores como semanas, meses o años, *TOL* permite declarar fracciones inferiores del día indicando la hora minuto y segundo del instante. El patrón se completa así del siguiente modo:

```
Date y<year>[m<month:01>[d<day:01>[h<hour:00>[i<minute:00>[s<second:00>]]]]];
```

Por ejemplo:

```
Date y2011m11d11;          // -> 11/11/2011 00:00:00
Date y2000m02d04h06i08s10; // -> 04/02/2000 06:08:10
Date y2012m09d10h13;       // -> 10/09/2012 13:00:00
```

Fechas y números

Las fechas también pueden construirse explícitamente a partir de estos números que la definen mediante la función `YMD` que recibe opcionalmente hasta 6 argumentos con el año, mes, día, hora, minuto y segundo de la fecha deseada.

Por ejemplo:

```
Date YMD(2011, 11, 11);          // -> 11/11/2011 00:00:00
Date YMD(2000, 2, 4, 6, 8, 10); // -> 04/02/2000 06:08:10
Date YMD(2012, 9, 10, 13);       // -> 10/09/2012 13:00:00
```

Inversamente, podemos obtener estos números a partir de las funciones: `Year`, `Month`, `Day`, `Hour`, `Minute` y `Second` que devuelven respectivamente el año, mes, día, hora, minuto y segundo de la fecha indicada.

Otra función de este tipo, especialmente útil, es `WeekDay` que devuelve el día de la semana con un número entero de 1 a 7 siendo el 1 el lunes.

Fechas especiales

TOL dispone de algunas fechas especiales ya implementadas:

- El día de hoy: `Date Today (valor variable)`.
- El instante actual (ahora): `Date Now (valor variable)`.
- Una fecha anterior a todas: `Date TheBegin`.
- Una fecha posterior a todas: `Date TheEnd`.
- Una fecha desconocida: `Date UnknownDate`.

Nótese que mientras que la variable `Now`, muestra también la hora, minuto y segundo del instante actual, `Today` sólo devuelve el año, mes y día (hora 00:00:00).

Índice temporal

Cada valor de fecha tiene un número asociado que depende del número de días transcurridos desde un día dado, correspondiendo el valor 1 al primer día del año 1900.

Las funciones `DateToIndex` e `IndexToDate` nos permiten transformar la fecha en número y el número en fecha respectivamente.

Por ejemplo:

```
Real DateToIndex(y2000);    // -> 36525
Real IndexToDate(41161);    // -> y2012m09d10
Date IndexToDate(50000.5); // -> y2036m11d22h12
```

Nótese que el valor numérico que asocia *Microsoft Excel* a las fechas es similar salvo por una unidad, ya que *Excel* considera por error que el año 1900 fue bisiesto:

```
Real excel_time = DateToIndex(<date>) + 1; // desde marzo de 1900
```

Otros sistemas de numeración para el tiempo como el *tiempo Unix* (que considera los segundos transcurridos desde 1970) puede obtenerse fácilmente a partir del índice en TOL a través de una operación aritmética:

```
Real unix_time = ( DateToIndex(<date>) - DateToIndex(y1970) ) * 86400;
```

2.4.2 Fechados (`TimeSet`)

Comúnmente con las fechas expresamos no sólo un instante, sino todo el intervalo de tiempo hasta el instante siguiente. De modo que el día 1 de enero de 2002 (`y2012`) puede representar tanto al primer día del año como al año completo.

Para dotar de este valor añadido a una fecha hemos de acompañarla de un fechado (diario o anual para el ejemplo anterior) que nos permita conocer cuál es la siguiente fecha.

En *TOL* existe un tipo de datos especial denominado `TimeSet` que representa a un conjunto de fechas o fechado y que nos sirve como soporte o *dominio temporal* para entender una fecha como un intervalo.

Fechados predefinidos

TOL incorpora un conjunto de fechados ya predefinidos, muy útiles en esta función de dominio temporal, entre los que destacamos los siguientes:

- Fechado anual, formado por todos los primeros de año: `TimeSet Yearly`.
- Fechado mensual, formado por todos los primeros de mes: `TimeSet Monthly`.
- Fechado diario, formado por todos los días: `TimeSet Daily`.
- Fechado semanal, formado por todos los lunes: `TimeSet Weekly`.
- Fechado trimestral, formado por los primeros de mes de enero, abril, julio y octubre: `TimeSet Quarterly`.
- Fechado semestral, formado por los primeros de mes de enero y julio: `TimeSet HalfYearly`.

Operaciones con fechas sobre fechados

Hay dos funciones muy útiles para operar con fechas que hacen uso del fechado como dominio temporal:

- La función sucesora de una fecha: `Succ`, que nos permite encontrar una fecha un número de veces posterior a otra en un determinado fechado.
- La función diferencia de fechas: `DateDif`, que nos permite determinar la distancia temporal entre dos fechas en un fechado.

Ejemplo:

```
// El día anterior al 1 de marzo de 2012 es:
Date prev_Mar.2012 = Succ(y2012m03, Daily, -1); // -> y2012m02d29
// ya que el año 2012 es bisisesto.
// El número de días de ese primer trimestre de 2012 es:
Real DateDif(Daily, y2012, Succ(y2012, Quarterly, 1)); // -> 91
```

Conjuntos temporales

Un fechado también puede utilizarse para representar un conjunto arbitrario de fechas, selección de todas las fechas existentes en otro fechado.

Un ejemplo, implementado en *TOL* por su carácter de fiesta móvil es el fechado de pascua (`Easter`) que representa al conjunto temporal de todos los domingos de pascua (del cristianismo occidental).

Con este criterio, *TOL* incorpora dos conjuntos temporales especiales:

- El conjunto de todos los días: `TimeSet C` (que coincide con el fechado diario).
- El conjunto sin ningún día: `TimeSet W`.

Creación de conjuntos temporales

Si bien el conjunto de fechados predefinidos en *TOL* puede bastarnos en su función de dominio temporal, para su uso como conjuntos de fechas necesitamos una mayor flexibilidad. Por ello, *TOL* incorpora un conjunto de funciones que nos permiten definir nuevos fechados:

- Las funciones `Y`, `M`, `D`, `H`, `Mi` y `S` nos permiten obtener subconjuntos temporales formados respectivamente por todas las fechas de un año, de un mes, de un día, de una hora, de un minuto o de un segundo.
- La función `WD` nos permite crear un subconjunto de fechas según el día de la semana.
- La función `Day` nos permite crear un fechado con una fecha dada, mientras que la función `DatesOfSet` hace lo propio con todas las fechas de un conjunto.

Álgebra del tiempo

Para facilitar la construcción de conjuntos temporales, *TOL* incorpora además las operaciones del álgebra de conjuntos:

- El operador `+` y la función `Union` que nos permiten unir conjuntos temporales.
- El operador `*` y la función `Intersection` que nos permiten obtener la intersección de conjuntos temporales.
- El operador `−` que nos permite restar a un conjunto temporal todas las fechas de otro.
- La función `Belong` que nos permite comprobar si una fecha pertenece o no a un conjunto temporal.

Ejemplos:

Como ejemplo de uso de las funciones anteriores podemos replicar el fechado trimestral:

```
TimeSet myQuarterly = D(1)*(M(1)+M(4)+M(7)+M(10));
```

Construir un fechado con los días de lunes a viernes:

```
TimeSet mondayToFriday = Daily-WD(6)-WD(7);
```

O crear un fechado con los primeros domingos de mes, apoyándonos en el hecho de que éstos se encuentran irremediablemente entre los primeros siete días del mes:

```
TimeSet firstSundays = (D(1)+D(2)+D(3)+D(4)+D(5)+D(6)+D(7))*WD(7);
```

La función intervalo (`In`)

La función `In` nos permite restringir un fechado a un intervalo de fechas, por ejemplo:

```
// Todos los meses entre el año 2000 y el 2012 (éste incluido)
TimeSet In(y2000, y2012m12, Monthly);
```

La función periódica (`Periodic`)

La función `Periodic` nos permite obtener un subconjunto temporal periódico (con periodo entero) a partir del periodo y una de las fechas del subconjunto.

Por ejemplo, una manera alternativa de obtener el conjunto de todos los lunes, es construir un subconjunto temporal del fechado diario con periodo 7:

```
// Se utiliza que el día 10/09/2012 es lunes
TimeSet mondays_alternative = Periodic(y2012m09d10, 7, Daily);
```

La función sucesora (`Succ`)

La función `Succ` nos permite trasladar un conjunto de fechas un determinado número de unidades dentro de otro fechado.

Por ejemplo, para obtener el conjunto formado por todos los últimos días del mes, (ya que el número varía de un mes a otro) podemos partir del conjunto de los primeros de mes y trasladarlos un día hacia atrás:

```
TimeSet lastDaysOfMonth = Succ(D(1), -1, Daily);
```

Nótese que la función sucesora para fechados recibe el segundo y tercer argumento en un orden distinto a la función sucesora para fechas:

```
Date Succ(Date date, TimeSet dating, Real integer);
TimeSet Succ(TimeSet timeSet, Real integer, TimeSet dating);
```

Una extensión de la función sucesora (`Succ`) que nos permite crear un fechado con todo un intervalo de traslaciones es la función `Range`. Por ejemplo:

```
// Conjunto de los tres últimos días de mes
TimeSet Range(D(1), -3, -1, Daily);
```

2.4.3 Series temporales (`Serie`)

Una serie temporal es una sucesión de datos ordenados cronológicamente, en los que cada uno de ellos corresponde a un determinado momento en un fechado. *TOL* incorpora a diferencia de otros lenguajes de un tipo específico para la gestión de series temporales, la gramática `Serie`.

Series temporales ilimitadas

A la hora de definir series temporales *TOL* permite crear series sin restringirlas a un determinado intervalo, son lo que podemos denominar series temporales ilimitadas.

Podemos crear series ilimitadas utilizando algunas de las siguientes funciones:

Elementales:

- La función pulso (`Pulse`) que devuelve una serie que vale 1 para la fecha indicada y 0 para el resto. Por ejemplo:

```
Serie pulse2012 = Pulse(y2012, Yearly);
```

- La función compensación (`Compens`) que devuelve una serie que vale 1 para la fecha indicada, -1 para la siguiente fecha y 0 para el resto.
- La función escalón (`Step`) que devuelve una serie que vale 0 hasta la fecha indicada y 1 desde esa fecha en adelante.

- La función tendencia (`Trend`) que devuelve una serie que vale 0 hasta la fecha indicada, 1 en esa fecha y va incrementándose de uno en uno para las fechas siguientes.
- La función línea recta (`Line`) que devuelve una serie con valores en la recta que pasa por los dos puntos (pares fecha-valor) indicados.

Con soporte en un fechado:

- La función indicadora de un fechado (`CalInd`) que devuelve una serie que vale 1 cuando la fecha pertenece al fechado y 0 cuando no. Se espera que el fechado de la serie contenga al fechado usado en la definición. Por ejemplo:

```
// La serie diaria de los primeros de mes:
Serie day1 = CalInd(Monthly, Daily);
```

- La función cardinal de un fechado (`CalVar`) que devuelve una serie con la cantidad de fechas que tiene el fechado entre una fecha dada y su sucesora en el fechado de la serie. Se espera que el fechado de la serie esté contenido en el fechado usado en la definición. Por ejemplo:

```
// La serie del número de días de cada mes:
Serie num_days = CalVar(Daily, Monthly);
```

Aleatorias:

- La función distribución uniforme (`Rand`) que devuelve una serie con valores aleatorios entre dos números dados.
- La función distribución normal (`Gaussian`) que devuelve una serie con valores aleatorios distribuidos normalmente con la media y desviación estándar indicadas.

En ocasiones, se usa la denominación «series infinitas» para referirse a las series ilimitadas, aunque en rigor, éstas serían sólo las que no tienen fin, aunque tengan principio.

Series temporales delimitadas

Para obtener series temporales delimitadas podemos utilizar la función `SubSer` que nos permite restringir una serie a un determinado intervalo. Por ejemplo:

```
Serie SubSer(Pulse(y2012, Yearly), y2000, y2020);
```

También podemos construir series explícitamente a partir de las filas de una matriz:

```
Matrix data = (
 (1.2, 2.4, 1.5, 1.0, 3.1, 2.0),
 (0.8, 7.1, 1.1, 4.2, 5.1, 2.2)
);
Set MatSerSet(data, Yearly, y2000);
```

Aunque quizá lo más interesante es obtenerlas a partir de alguna fuente de datos, como un archivo o una base de datos. Para ver cómo construir series temporales a partir de las distintas fuentes de datos véanse las secciones 3.1.2 y 3.2.3.

Para conocer la fecha inicio y fin de una serie, disponemos de las funciones `First` y `Last` respectivamente.

Téngase en cuenta que *TOL* considera los datos anteriores y posteriores de una serie delimitada, como omitidos, de modo que intenta evitar que las series que comiencen o acaben con este valor, recortándolas automáticamente.

Por ejemplo:

```
Matrix data = Row(?, 1.2, 2.4, ?, 1.0, 3.1, 2.0);
Serie series = MatSerSet(data, Yearly, y2000)[1];
Date First(series); // -> y2001
```

Datos de una serie

Para acceder a los elementos de una serie temporal disponemos de la función `SerDat` que nos devuelve el valor de la serie para una fecha dada. Para modificarlo usaremos la función `PutSerDat` indicando como primer argumento la fecha y como segundo el nuevo valor.

Operadores sobre series temporales

Para operar con series temporales, disponemos de los operadores más comunes de las operaciones con números reales. Los operadores aritméticos (`+`, `-`, `*`, `/`) nos permiten incluso operar entre una serie y un número real, devolviendo la serie temporal resultado de operar cada valor de la serie con el número indicado.

Téngase en cuenta, que cuando se opera con dos o más series temporales, éstas tienen que tener el mismo fechado y la operación se realizará únicamente sobre el intervalo intersección, es decir, sobre las fechas que pertenezcan a todas las series.

Concatenación de series temporales

Para poder completar los datos de una serie, disponemos de los operadores `<<` (con prioridad en la derecha) y `>>` (con prioridad en la izquierda) que nos permiten concatenar dos series temporales poniendo una a continuación de otra, indicando los valores de qué serie tienen prioridad en caso de solapamiento.

Si este solapamiento desea controlarse explícitamente disponemos de la función `Concat` de tres argumentos, que concatena dos series temporales usando una fecha para indicar hasta cuando tomar los datos de la primera serie (la de la izquierda).

Por ejemplo:

```
Serie zeros = SubSer(CalInd(W, Yearly), y2000, y2008);
Serie ones = SubSer(CalInd(C, Yearly), y2005, y2012);
Serie cc1 = zeros >> ones; // SumS -> 4
Serie cc2 = zeros << ones; // SumS -> 8
Serie cc3 = Concat(zeros, ones, y2006); // SumS -> 6
```

Operaciones con series temporales

No sólo disponemos de los operadores aritméticos, sino de todo el conjunto de funciones matemáticas sobre número reales (exponenciales, trigonométricas, hiperbólicas, etc.) que al aplicarlas sobre series temporales actúan independientemente sobre cada valor de la serie temporal.

Aunque la mayor parte de las funciones matemáticas sobre números reales disponen de su versión para series temporales, en general podemos aplicar una función cualquiera a todos los valores de una serie mediante la instrucción `EvalSerie` (análoga a `EvalSet`).

También disponemos de una instrucción condicional sobre series temporales, `IfSer`, que nos permite construir series con una estructura condicional.

<u>Por ejemplo:</u>

```
// Creamos una serie con valores omitidos a modo de ejemplo
Serie series = MatSerSet(Row(1, 2, ?, 1, 3, ?, 4, 2), Yearly, y2000)[1];
// Sustituimos estos valores desconocidos por ceros
Serie IfSer(IsUnknown(series), 0, series);
```

Estadísticos sobre series temporales (`<Statistic>S`)

Las funciones estadísticas que presentamos en la sección 2.3.1 también están implementadas explícitamente para series temporales con la siguiente nomenclatura: `<Statistic>S`.

A éstas podemos añadirles algunas otras implementadas específicas para series temporales como:

- El número de datos de una serie: `CountS`.
- El primer valor de una serie: `FirstS`.
- El último valor de una serie: `LastS`.

Téngase en cuenta, que la sintaxis de todas estas funciones admite opcionalmente como segundo y tercer argumento un par de fechas con las que delimitar el intervalo de aplicación del estadístico:

```
Real <Statistic>S(Serie series[, Date begin, Date end])
```

<u>Por ejemplo:</u>

```
Serie series = SubSer(Gaussian(0, 1, Monthly), y2000, y2012m12);
Real mean = AvrS(series);
Real variance_2011 = VarS(series, y2011, y2011m12);
Real FirstS(series) <= MaxS(series); // -> True
```

Cambio de fechado (`DatCh`)

Las funciones estadísticas anteriores son especialmente útiles para cambiar el fechado de una serie a otro superior (menos detallado o con intervalos más grandes).

La función `DatCh` (*dating change*) nos permite construir una serie en un nuevo fechado, a partir de los datos de otra, con la siguiente sintaxis:

```
Serie DatCh(Serie series, TimeSet dating, Code statistic);
```

recibiendo como tercer argumento la función que se utilizará para obtener cada valor de la nueva serie a partir de los datos de un intervalo, y que ha de responder con la forma extendida de las funciones estadísticas sobre series temporales:

```
Real (Serie series, Date begin, Date end)
```

Dependiendo de la naturaleza de los datos y del cambio de fechado, es necesario utilizar distintos estadísticos. Por ejemplo, si los datos representan el valor de una magnitud que depende del tamaño del intervalo, debemos acumular los valores con `SumS` al hacer el cambio de fechado. Un caso típico sería una serie con las ventas de un determinado producto. Si por el contrario, el valor de la magnitud no varía con el tamaño de los intervalos, usaremos un estadístico de promedio como `AvrS`, o alguno de los que nos permiten seleccionar un valor del intevalo como: `FirstS`, `LastS`, `MinS` o `MaxS`.

2.4.4 Diferencias finitas. Polinomios de retardos (Polyn)

Para el análisis de las series temporales, como sucesiones matemáticas, es muy común hacer uso de las *diferencias finitas*.

Las diferencias finitas desempeñan un papel en el estudio de las series temporales y de las ecuaciones en diferencias, similar al que desempeñan las derivadas en el análisis de funciones y el cálculo diferencial.

Diferencia regular

Denominamos diferencia (o diferencia regular) de una serie temporal a la acción y el resultado de restar a cada valor de la serie su valor inmediatamente anterior.

Si X_t es una serie temporal, su diferencia es: $\Delta X_t = X_t - X_{t-1}$

Para expresar las diferencias, *TOL* incorpora un nuevo tipo de dato, la gramática `Polyn`, con la que representar operadores polinómicos de retardos que actúen sobre las series temporales.

Retardo

El operador elemental de retardo en *TOL* es `Polyn B` (del inglés *backward*) y al actuar sobre una serie temporal sustituye el valor en una fecha por el valor en el instante inmediatamente anterior. Podemos decir que sobre el soporte de fechas de la serie temporal, el operador `B` desplaza los datos hacia adelante.

Para aplicar las variables `Polyn` sobre las series temporales se utiliza el operador `:` (dos puntos).

Por ejemplo:

```
Serie X_t = SetSer(Range(1, 8, 1), Yearly, y2001);
Serie X_t.minus.1 = B : X_t;         // retardo de X_t
Serie Dif.X_t = X_t - X_t.minus.1; // diferencia de X_t
```

Podemos ver los resultados de las operaciones anteriores en forma de tabla:

t	2001	2002	2003	2004	2005	2006	2007	2008	2009
X_t	1	2	3	4	5	6	7	8	
X_t.minus.1		1	2	3	4	5	6	7	8
Dif.X_t		1	1	1	1	1	1	1	

Tabla 2.4.1: Ejemplo en forma de tabla del resultado de aplicar un retardo y una diferencia a una serie temporal.

Polinomios

Si aplicamos el operador de retardo sucesivas veces, podemos obtener retardos y diferencias de orden superior a 1 y así definir operadores polinómicos de manera general como una combinación lineal de potencias del operador de retardo `B`.

Por ejemplo:

```
Polyn p = 1 + 0.5*B - 0.8*B^2;
```

Inversamente, podemos conocer el grado y coeficientes de un polinomio dado a través de las funciones `Degree` y `Coef` respectivamente.

Otra función muy útil es `Monomes` que nos descompone un polinomio en un conjunto de monomios.

Nótese que *TOL* implementa la aplicación sucesiva de operadores como el producto de variables `Polyn`. De modo que podemos definir diferencias de orden superior a 1 como potencias del operador diferencia `Polyn (1-B)`.

```
Polyn dif = 1-B;
Polyn dif_order.2 = (1-B)^2 // segunda diferencia
```

Diferencias estacionales

Las diferencias estacionales son aquellas diferencias en las que a cada valor de la serie se le resta otro un determinado número de instantes anterior.

Si X_t es una serie temporal, una diferencia estacional de periodo p es: $\Delta_p X_t = X_t - X_{t-p}$

Estas diferencias son muy útiles en el análisis de series temporales que presentan una periodicidad, como por ejemplo, las series en fechado mensual que manifiestan ciclos anuales admiten diferencias estacionales de periodo 12:

```
Polyn dif_period.12 = 1-B^12; // diferencia estacional
```

No hay que confundir la estacionalidad de una diferencia con sus sucesivas aplicaciones. Por ejemplo, podemos hablar de una diferencia estacional de orden 2 y periodo 12:

```
Polyn dif_period.12_order.2 = (1-B^12)^2;
```

Operador adelanto

TOL incorpora también un operador de adelanto, `Polyn F` (del inglés *forward*), inverso al operador de retardo.

```
Polyn B*F; // == Polyn 1
```

Su comportamiento al actuar sobre una serie temporal sustituye el valor en una fecha por el valor en el instante inmediatamente posterior.

2.4.5 Ecuaciones en diferencias. Cocientes de polinomios (`Ratio`)

En modelación es muy habitual utilizar *ecuaciones en diferencias* para describir las series temporales objeto de análisis. De manera general podemos expresar la ecuación en diferencias que describe una serie Y_t como un polinomio de retardos $p(B)$ aplicado a esta serie igualado a una constante u otra serie temporal X_t.

$$p(B)Y_t = X_t$$

TOL incorpora un nuevo tipo de datos, la gramática `Ratio`, para expresar la inversión del polinomio de retardos, con la que se resuelve la ecuación en diferencias:

$$Y_t = \frac{1}{p(B)} X_t$$

Así, una variable de tipo `Ratio` se define como el cociente de dos variables de tipo `Polyn`.

Veamos un ejemplo:

```
// Partimos de una serie "y" y un polinomio "p"
// y encontramos la serie "x" que verifica:
```

```
//   p:y = x
Serie y = SubSer(Gaussian(0, 1, Daily), y2000, y2000m12d31);
Polyn p = 1 - 0.5*B;
Serie x = p:y;

// Resolvemos la ecuación en diferencias y volvemos a encotrar "y" (y_new):
//    y_new = (1/p) : x
Ratio r = 1/p;
Serie y0 = SubSer(y, First(y), First(y)); // valores iniciales
Serie y_new = DifEq(r, x, y0);
```

Nótese que para resolver la ecuación en diferencias es necesario incorporar un determinado conjunto de *valores iniciales*. Intuitivamente podemos comprender esto si nos damos cuenta de cómo la serie diferenciada sufre una perdida de información: tiene menos datos que la serie sin diferenciar. Esta información es necesario incorporarla al hacer el proceso inverso, de un modo similar a como ocurre en la integración en el cálculo diferencial.

El segundo miembro de la ecuación en diferencias puede ser también una serie diferenciada:

$$p(B)Y_t = q(B)X_t$$

De modo que, en general, las variables de tipo `Ratio` se pueden definir como un conciente de polinomios:

$$Y_t = \frac{q(B)}{p(B)}X_t$$

Dada una variable de tipo `Ratio`, podemos obtener los polinomios numerador y denominador, mediante las funciones `Numerator` y `Denominator` respectivamente.

2.4.6 Modelación con series temporales

En la modelación con series temporales es bastante frecuente encontrar correlación entre los distintos valores de la serie, de modo que, al menos en parte, podemos explicar un determinado valor de serie en función de sus valores anteriores.

Éste es el fundamento de los modelos autorregresivos (AR), en el que el output es descrito como combinación lineal de retardos suyos:

$$Y_t = \sum_i \varphi_i \, Y_{t-i} + \, E_t$$

donde Y_t es la serie temporal observada, φ_i son los parámetros autorregresivos y E_t es la parte no explicada o error del modelo.

Modelo ARIMA

Una extensión de estos modelos autorregresivos, incorporando diferencias y medias móviles son los modelos ARIMA (por sus siglas en inglés: *AutoRegressive Integrated Moving Average*) que podemos describir mediante la siguiente ecuación:

$$\left(1 - \sum_i \varphi_i B^i\right)(1-B)^d Y_t = \left(1 - \sum_i \theta_i B^i\right) E_t$$

donde el primer polinomio de retardos, con los parámetros φ_i, corresponde con la parte autorregresiva (AR), el segundo polinomio son las diferencias de orden d y el tercero, con los parámetros θ_i, la parte de media móvil (MA).

Bloques ARIMA (`@ARIMAStruct`)

Para la definición de modelos ARIMA, *TOL* incorpora la estructura `@ARIMAStruct` con cuatro campos, en los que además de los tres polinomios descritos, se puede indicar un valor para la periodicidad del modelo y que se utiliza para crear bloques ARIMA estacionales (*seasonal* ARIMA o SARIMA).

En general, podemos describir un modelo ARIMA como un conjunto de bloques ARIMA.

Por ejemplo:

```
// Creamos la estructura correspondiente a un ARIMA(2,0,0)-SARIMA.12(0,1,1)
Set arima = [[
  @ARIMAStruct(1, 1-0.5*B-0.5*B^2, 1, 1);
  @ARIMAStruct(12, 1, 1-0.5*B^12, 1-B^12)
]];
```

Etiquetas ARIMA

Para facilitar la especificación de los bloques ARIMA, *TOL* incorpora un sistema de etiquetado de los modelos ARIMA con la siguiente forma:

```
"P<period>DIF<order>AR<degrees>MA<degrees>"
```

en la que debemos sustituir el código entre paréntesis angulares (<>) por los correspondientes valores numéricos.

Para el polinomio de diferencias sólo es necesario indicar el número de diferencias a aplicar, ya que el grado de la diferencia se obtiene del valor del periodo. Mientras que para los polinomios AR y MA han de indicarse explícitamente los distintos grados separados con puntos (`.`).

En el caso en el que el polinomio ARIMA esté formado por más de un bloque, la etiqueta puede componerse separando con el guión bajo (_) las distintas partes de cada bloque.

Por ejemplo, la etiqueta del modelo ARIMA definido antes, es:

```
"P1_12DIF0_1AR1.2_0MA0_1"
```

como combinación de:

```
"P1DIF0AR1.2MA0" // un AR con dos parámetros (grados 1 y 2)
"P12DIF1AR0MA1"  // una diferencia con estacionalidad 12
                 //   y un MA con un parámetro (grado 1)
```

La librería estándar de *TOL*, `StdLib` (véase la sección 4.1) dispone de dos funciones para obtener los bloques ARIMA a partir de la etiqueta y viceversa: `GetArimaFromLabel` y `GetLabelFromArima`.

Estimación de modelos ARIMA (`Estimate`)

Además de la estructura ARIMA descrita, para modelar con series temporales podemos incorporar términos explicativos, de modo que la ecuación del modelo sea:

$$Y_t = \sum_i \beta_i X_{i,t} + N_t$$

donde el término del ruido, N_t, es el que presenta la estructura ARIMA descrita anteriormente.

TOL incorpora un estimador máximo verosímil, `Estimate`, que nos permite estimar conjuntamente, tanto los parámetros lineales de la regresión (β_i) como los parámetros de la parte ARIMA (φ_i y θ_i).

Para especificar las distintas características del modelo en la función `Estimate` es necesario crear un conjunto con la estructura `@ModelDef`, indicando entre otros argumentos: la serie output, los polinomios de la estructura ARIMA y las series inputs. Concretamente los inputs del modelo se indican mediante pares polinomio-serie con estructura `@InputDef` permitiéndonos así incorporar polinomios de retardo a los términos explicativos.

Ejemplo:

```
// Creamos una serie temporal con una estructura MA(2)
// a la que añadimos una componente externa (filtro)
// para realizar una estimación controlada a modo de ejemplo.
Serie residuals = SubSer(Gaussian(0, 0.1, Monthly), y1999m11, y2010);
Real phi1 = 0.5;
Real phi2 = -0.3;
Polyn MA = 1 - phi1*B - phi2*B^2;
Serie noise = MA:residuals;
Real beta = 1.2;
Serie input = SubSer(Rand(1, 2, Monthly), y2000, y2010);
Serie output = noise + beta * input;
// Estimamos el modelo: output = beta*input + noise
// con: noise = MA(2):residuals
Set estimation = Estimate(@ModelDef(output, 1, 0, 1, 1, // Output
  Polyn 1, [[Polyn 1]], [[Polyn 1-0.1*B-0.1*B^2]],      // ARIMA
  [[@InputDef(0.1, input)]], Empty)                     // Filtro
);
Real estimated_beta = estimation["ParameterInfo"][1]->Value;
Real estimated_phi1 = estimation["ParameterInfo"][2]->Value;
Real estimated_phi2 = estimation["ParameterInfo"][3]->Value;
```

2.5 Nociones avanzadas

2.5.1 Programación modular (`NameBlock`)

Un nuevo tipo de variable en *TOL* que también tiene naturaleza múltiple como los conjuntos (`Set`), es decir, está formada a su vez por otras variables, es el bloque o *nameblock* (`NameBlock`).

Aunque su naturaleza y la forma de definirlos se parezca a la de los conjuntos, su motivación es bastante distinta. Mientras que los conjuntos (`Set`) básicamente están destinados a contener elementos de distinto tipo, un bloque (`NameBlock`) está pensado para ofrecer un espacio de definición o ámbito local y permanente donde crear variables y funciones.

Una de las principales razones por la que se incorporaron los *nameblocks* a *TOL* fue la de favorecer cierta organización y modularidad en la programación, permitiendo así la definición de variables y funciones con un nivel de visibilidad local, que de otro modo pasarían a formar parte del grueso de variables y funciones globales.

Para definir un bloque (`NameBlock`) haremos uso de la siguiente sintaxis:

```
NameBlock <name> = [[
  <Grammar1> <name1> = ...;
  <Grammar2> <name2> = ...;
  ...
]];
```

Ejemplo:

```
NameBlock block = [[
  Real value = 3.1;
  Text country = "Spain";
  Date date = Today
]];
```

Nótese que al construir el *nameblock*, las variables definidas, a diferencia de lo que ocurre al crear conjuntos, sólo tienen visibilidad local. Compárense los siguientes códigos:

```
Set set = [[
  Text text1 = "hola"
]];
WriteLn(text1);
```

```
hola
```

```
NameBlock block = [[
  Text text2 = "hola"
]];
WriteLn(text2);
```

```
ERROR: [] text2 no es un objeto valido para el tipo Text.
```

Acceso a los miembros

Los elementos o miembros de un bloque se caracterizan por su nombre, siendo esta característica única para cada elemento e imprescindible. Las funciones (`Code`) definidas dentro de un bloque tienen acceso al resto de elementos del bloque por su nombre como si éstos fueran globales.

En general, para acceder a los miembros de un bloque usaremos el operador `::` (cuatro puntos) como si compusiésemos un identificador de dicho miembro anteponiéndole el nombre del bloque en el que se encuentra:

```
<Grammar> <blockName>::<memberName>;
```

Por ejemplo:

```
NameBlock supercomputer = [[
  Text name = "DeepThought";
  Real theAnswer = 42;
  Real GetTheAnswer(Real void) { theAnswer }
]];
Text supercomputer::name;             // -> "DeepThought"
Real supercomputer::GetTheAnswer(?); // -> 42
```

Nótese cómo se puede definir una función en el marco de un *nameblock* y cómo ésta puede llamarse en la misma sentencia que se utiliza para su acceso.

Encapsulamiento

Otra de las características que incorporan los *nameblocks* es la de favorecer el encapsulamiento de variables y funciones, dando visibilidad sólo a los miembros que tienen utilidad fuera del contexto local del bloque.

Para ello, la accesibilidad a los miembros de un bloque puede restringirse a través de la elección de su nombre, pudiendo distinguir tres tipos:

- Miembros públicos, cuyo acceso desde fuera del bloque está permitido, tanto para su consulta como su modificación. Sus nombres se caracterizan por comenzar con una letra.

- Miembros de sólo lectura, cuyo acceso desde fuera del bloque está orientado a sólo la consulta. Sus nombres se caracterizan por comenzar con la combinación guión bajo y punto (`_.`).
- Miembros privados, cuyo acceso desde fuere del bloque no está permitido. Sus nombres se caracterizan por comenzar por un guión bajo (`_`) seguido de cualquier otro carácter alfanumérico distinto del punto (`.`)

Ejemplo:

Como ejemplo de uso de los *nameblocks* como estructuras modulares creamos un módulo a modo de cronómetro con funciones para medir el paso del tiempo entre llamadas.

```
NameBlock Clock = [[
  Text _.description = "Reloj para usar como cronómetro";
  Real _initialTime = 0; // miembro privado
  Real _elapsedTime = 0; // miembro privado
  Real Start(Real void) {
    Real _elapsedTime := 0;
    Real _initialTime := Copy(Time);
  1};
  Real Stop(Real void) {
    If(_initialTime!=0, {
      Real _elapsedTime := Time - _initialTime;
      Real _initialTime := 0;
    1}, 0)
  };
  Real Reset(Real void) {
    Real _elapsedTime := 0;
    Real _initialTime := 0;
  1};
  Real GetElapsedTime(Real void) {
    Case(_elapsedTime!=0, {
      Copy(_elapsedTime)
    }, _initialTime!=0, {
      Time - _initialTime
    }, True, {
      WriteLn("El reloj no está en marcha.", "W");
      ?
    })
  }
]];
Real Clock::Start(?);
Real Sleep(1);
Real Clock::GetElapsedTime(?); // -> ~1
Real Sleep(1);
Real Clock::Stop(?);
Real Sleep(1);
Real Clock::GetElapsedTime(?); // -> ~2
```

2.5.2 Clases (`Class`)

La incorporación de la gramática `NameBlock` en *TOL*, abre un gran abanico de posibilidades en torno a la creación de bloques funcionales, que desemboca en el desarrollo de la programación orientada a objetos.

Hay mucho escrito sobre cómo introducir el paradigma de la programación orientada a objetos, pero quizá, para un usuario de *TOL* que conoce el papel que desempeñan las estructuras (`Struct`) en relación a los conjuntos (`Set`) la mejor manera de comenzar sea diciendo que las clases (`Class`) son a los bloques (`NameBlock`) lo que las estructuras a los conjuntos.

De manera similar a cómo introdujimos las estructuras (véase la sección 2.2.4), podemos presentar las clases como una especialización de la gramática `NameBlock` que nos permite representar entidades y crear unidades de un determinado concepto. Un *objeto*, se define así, como cada instancia de una clase, y que podríamos describir como *nameblock estructurado*.

Además de los miembros del bloque que albergan la información característica de cada unidad, las clases (a diferencia de las estructuras) nos permiten dotar a los objetos de una estructura funcional (común para todas las instancias de una misma clase) que nos permite interaccionar con ellos. A los primeros se los denominan *atributos* y a los segundos *métodos*.

Terminología

A continuación, para facilitar su aprendizaje, se indica el significado de algunos términos de la programación orientada a objetos:

- Clase: Es la estructura, el diseño o el patrón con el que se construyen los objetos.
- Objeto: Es cada una de las instancias (variables de tipo `NameBlock`) creadas con una clase.
- Miembro: Es cada uno de los elementos de un objeto o instancia. También puede denominarse componente.
- Atributo: Es cualquier miembro de tipo variable (no función) de una instancia, su valor es particular para cada instancia. También puede denominarse propiedad.
- Método: Es cualquier miembro de tipo función de una instancia. Es común para todas las instancias, aunque actúa y sólo tiene acceso (interno) al objeto sobre el que se hizo la llamada.

Definición de clases

La definición de una clase se realiza a través de la siguiente sintaxis:

```
Class @<ClassName> {
  // Atributos:
  <Grammar1> <attribute1>;
  <Grammar2> _.<attribute2>; // atributo de sólo lectura
  <Grammar3> _<attribute3>;  // atributo privado (de uso interno)
  <Grammar4> <attribute4> = <defaultValue4>;
  ...
  // Métodos:
  <GrammarM1> <Method1>(<arguments...>) {
    <code...>
  };
  <GrammarM2> _<Method2>(<arguments...>) {
    <code...>
  }; // método privado (de uso interno)
  ...
};
```

donde el código entre paréntesis angulares (<>) ha de sustituirse por su valor correspondiente.

Téngase en cuenta que los miembros de las instancias de una clase responden a los mismos criterios de visibilidad que los de cualquier otro *nameblock*, pudiéndose así declarar miembros públicos, privados o de sólo lectura.

<u>Ejemplo:</u>

```
// Para facilitar la comparación con las estructuras,
// redefinimos como clase, la estructura @Vector3D:
```

```
Class @Vector3D {
// Atributos:
  Real _.x;
  Real _.y;
  Real _.z;
// Métodos:
  Real GetX(Real void){ _.x };
  Real GetY(Real void){ _.y };
  Real GetZ(Real void){ _.z };
  Real SetX(Real x) { Real _.x := x; 1 };
  Real SetY(Real y) { Real _.y := y; 1 };
  Real SetZ(Real z) { Real _.z := z; 1 };
  // spherical and cylindrical additional coordinates:
  Real GetR(Real void) { Sqrt(_.x^2 + _.y^2 + _.z^2) }; // length, radius
  Real GetRho(Real void) { Sqrt(_.x^2 + _.y^2) };       // radial distance
  Real GetPhi(Real void) { _ATan2(_.y, _.x) };          // azimuth angle
  Real GetTheta(Real void) { ACos(_.z/GetR(?)) };       // polar angle
  // auxiliar (private) methods:
  Real _ATan2(Real y, Real x) { ATan(y/x) + If(x<0,Sign(y)*Pi,0) }
};
```

Nótese que al igual que los nombres de las estructuras, los nombres de las clases también han de comenzar con el carácter `@` (arroba).

Instanciación

Para la creación de los objetos o instancias de una clase utilizaremos la siguiente sintaxis:

```
@<ClassName> object = [[
  // Lista de valores para los atributos (salvo si acaso los opcionales)
  <Grammar1> <attribute1> = <value1>;
  <Grammar2> _.<attribute2> = <value2>;
  ...
]];
```

El acceso desde fuera de la clase a los atributos se hace, como a los de otro bloque (`NameBlock`) cualquiera, mediante el operador `::` (cuatro puntos):

```
<Grammar1> value1 = object::<attribute1>;
```

<u>Por ejemplo:</u>

Para crear una instancia de la clase `@Vector3D`, escribimos:

```
@Vector3D vector3D = [[
  Real _.x = 1;
  Real _.y = 2;
  Real _.z = 3
]];
```

Nótese que la principal diferencia que encontraremos con un bloque declarado de forma similar:

```
NameBlock block3D = [[
  Real _.x = 1;
  Real _.y = 2;
  Real _.z = 3
]];
```

es la existencia de los métodos que dispone el objeto `vector3D` por ser instancia de la clase `@Vector3D` y que nos permiten obtener información adicional:

```
Real vector3D_length = vector3D::GetR(?);                // -> 3.741657
Real vector3D_azimuth = vector3D::GetPhi(?)*180/Pi;      // -> 63.435°
Real vector3D_inclination = vector3D::GetTheta(?)*180/Pi;  // -> 36.70°
```

Por así decirlo, la clase no sólo se encarga de verificar la creación de los atributos (y asignar en su caso los valores por defecto para los atributos opcionales) sino que recubre al objeto creado del conjunto de funcionalidades de la clase, los métodos.

Miembros estáticos (`Static`)

En ocasiones queremos incorporar una determinada información o funcionalidad a una clase y facilitar su acceso sin la necesidad de crear una instancia.

TOL permite que le incorporar a las clases miembros como si ésta fuera un tipo especial de *nameblock* simplemente anteponiendo la palabra clave `Static` en la definición:

```
Class <@ClassName> {
  ...
  // Atributos estáticos:
  Static <GrammarC1> <classAttribute1> = <valueC1>;
  ...
  // Métodos estáticos:
  Static <GrammarCM1> <ClassMethod>(<arguments...>) {
    <code...>
  };
  ...
};
```

El acceso a estos miembros de la clase que podemos denominar *miembros estáticos*, puede hacerse a través del operador `::` (cuatro puntos) con la siguiente sintaxis:

```
Anything @<ClassName>::<classMember>;
```

Declaración anticipada

Para poder utilizar el nombre de una clase, aunque su uso se posponga, ésta ha de estar declarada. De lo contrario *TOL* devolverá un error sintáctico. Para facilitar estos problemas, TOL admite la declaración anticipada de una clase indicando simplemente su nombre:

```
Class @<ClassName>;
```

Un caso que pone claramente de manifiesto esta necesidad es aquel en el que definimos dos clases interrelacionadas, de modo que cada una hace uso de la otra en su definición:

```
// Predeclaramos @B para poder usarlo en la definición de @A
Class @B;

// Declaramos @A usando @B (pues ya está predeclarada)
Class @A {
  ... @B ...
};

// Declaramos @B usando @A (pues ya está declarada)
Class @B {
  ... @A ...
};
```

2.5.3 Diseño de clases

A continuación indicamos algunas características adicionales de la creación y uso de las clases:

Gestión de los atributos

El valor de un atributo podemos modificarlo simplemente como otra variable cualquiera mediante:

```
<Grammar1> object::<attribute1> := <newValue1>;
```

Sin embargo, en el diseño de muchas clases, esto no es conveniente, ya que es posible que la edición de un atributo deba desencadenar otras modificaciones internas. Para evitar esto, se declaran los atributos como de sólo lectura (precedidos por `_.`) y se acompañan de un método para su edición.

Una práctica recomendable en estos casos, es crear para cada atributo (`_.attribute`) un par de métodos `Get<Attribute>` y `Set<Attribute>` que permitan su acceso y edición respectivamente:

```
<Grammar1> value1 = object::Get<Attribute1>(?);
Real object::Set<Attribute1>(<newValue1>);
```

Constructores alternativos

A menudo resulta conveniente disponer de funciones capaces de construir instancias a partir de unos pocos argumentos. Estos constructores alternativos no son propiamente constructores sino funciones que de manera directa o indirecta hacen una llamada al constructor por defecto cuya sintaxis es:

```
@<ClassName> object = [[ ... ]];
```

Estas funciones constructoras están fuertemente relacionadas con la clase, por lo que es común (aunque no imprescindible) introducirlas en ella como métodos estáticos de la clase, de modo que se pueda crear una instancia con la sintaxis:

```
@<ClassName> object = @<ClassName>::<Constructor1>(<arguments...>);
```

La función constructora (que no es más que un método estático que devuelve una instancia) se define como cualquier otro método pero precediendo la declaración con la palabra `Static`:

```
Class @<ClassName> {
  ...
  // Constructores:
  Static @<ClassName> <Constructor1>(<arguments...>) {
    <definition...>
  };
  ...
};
```

Herencia (`:`)

Una de las características más importantes de la programación orientada a objetos es la herencia. Este mecanismo nos permite derivar de una clase más general otra más particular conservando la definición de la primera y pudiendo así definir una nueva clase definiendo sólo los nuevos miembros que se desean incorporar.

Para crear una clase heredando de otra (u otras) se amplía la sintaxis de la definición de una clase con el operador dos puntos (`:`) seguido del nombre (o nombres) de las clases de las que se desea heredar:

```
Class @<ClassName> : @<ClassName1>[, @<ClassName2>, ...] {
  // Miembros:
  ...
};
```

donde los corchetes (`[]`) indican opcionalidad.

Redefinición de métodos

En el caso de que se herede el mismo método de dos clases distintas siempre prevalecerá la definición de la última respecto a la primera. Aún más, los métodos que se incorporen en la definición de la nueva clase prevalecerán a cualquier miembro heredado.

Téngase en cuenta, sin embargo, que no se puede cambiar la declaración de los métodos (gramática de la salida y gramática y nombres de los argumentos de entrada).

Clases abstractas

Asociado al concepto de herencia y de la redefinición de métodos surge la posibilidad de crear clases parcialmente definidas. Denominamos así, métodos virtuales a aquellos métodos que simplemente se declaran (sin indicar el cuerpo de la función) en la definición de la clase, dejándose su definición para las clases derivadas.

Las clases virtuales o abstractas son, por tanto, aquellas clases que tienen al menos un método virtual. Al no estar completamente definidas, no se pueden crear instancias directamente de ellas, sino de alguna clase derivada.

Ejemplo:

```
// Creamos una clase abstracta para representar figuras planas
Class @Shape {
  // Declaramos dos métodos virtuales que han de definirse en clases derivadas
  Real GetPerimeter(Real void);
  Real GetArea(Real void);
  // Definimos otros métodos que heredarán las clases derivadas
  Real PrintPerimeter(Real void){ WriteLn("Perimeter: "<<GetPerimeter(?)); 1};
  Real PrintArea(Real void) { WriteLn("Area: "<<GetArea(?)); 1}
};
// Derivamos dos clases de @Shape para representar cículos y cuadrados:
Class @Circle : @Shape {
  Real _.radius;
  Real GetPerimeter(Real void) { 2 * Pi * _.radius };
  Real GetArea(Real void) { Pi * _.radius**2 }
};
Class @Square : @Shape {
  Real _.side;
  Real GetPerimeter(Real void) { 4 * _.side };
  Real GetArea(Real void) { _.side**2 }
};
// Creamos tres figuras instanciando las clases @Circle y @Square
Set shapes = [[
  @Circle c1 = [[ Real _.radius = 1 ]];
  @Circle c2 = [[ Real _.radius = 1.5 ]];
  @Square s1 = [[ Real _.side = 2.25 ]]
]];
// Llamamos al método común WriteArea para imprimir sus áreas
Set EvalSet(shapes, Real (@Shape shape) { shape::PrintArea(?) });
```

```
Area: 3.141592653589793
Area: 7.068583470577035
Area: 5.0625
```

Referencia interna (`_this`)

En ocasiones, al diseñar una clase, surge la necesidad de que el objeto disponga de una referencia a sí mismo. Toda clase dispone de esta referencia a modo de atributo privado con el nombre `_this`.

Método destructor (`__destroy`)

Aunque no es muy habitual su uso en la programación en *TOL*, ya que la gestión de la memoria no corre a cargo del usuario, existe la posibilidad de incorporar un método destructor que se llame cuando una instancia se decompila.

Este método se podría encargar de tareas como cerrar archivos o conexiones a bases de datos abiertos durante la creación o uso de la instancia.

La declaración reservada para este método es:

```
Real __destroy (Real void)
```

2.5.4 Otros elementos del lenguaje

Directivas (`#<Directive>`)

Esta sección introducirá las directivas en TOL como un elemento especial del lenguaje.

`#Embed`: Véase la sección 3.1.1.

`#Require`: Véase la sección 3.4.2.

2.5.5 Uso de la memoria

Esta sección recogerá algunas indicaciones para comprender mejor el uso que TOL hace de la memoria y cómo mejorar la eficiencia de los algoritmos que lo requieran.

Objetos en memoria (`NObject`)

Estructuras para el acceso por referencia (`@<Grammar>`)

3 Uso de *TOL*

3.1 Sistema de archivos

3.1.1 Archivos .tol

Esta sección incluirá información relativa al diseño y creación de programas *TOL* en archivos .tol. Se mencionarán también otros tipos de archivos para la construcción de proyectos de código como los archivos .prj.

Directiva `#Embed`

3.1.2 Lectura y escritura de archivos

Esta sección incluirá información relativa a las funciones de lectura y escritura de archivos.

Destacamos las funciones: `FOpen`, `ReadFile`, `ReadTable`, `MatReadFile`, `VMatReadFile` y los tipos de archivos: .bst, .bdt, .bbm, .bvm, etc.

3.1.3 Serialización en *TOL*. *OIS*

Esta sección incluirá información relativa a la serialización en *TOL*: *OIS* y la lectura y escritura de archivos .oza.

3.1.4 Integración con el sistema operativo

Esta sección incluirá información relativa a la integración de *TOL* con el sistema de archivos a través del sistema operativo.

Destacamos las funciones: `GetDir`, `MkDir`, `GetFileName`, `GetFileExtension`, `GetFilePath`, `FileExist`, `FileDelete` y el conjunto de funciones `OsDir*` y `OsFil*`.

3.2 Comunicación

3.2.1 Comunicación con el sistema operativo

Esta sección incluirá información relativa a la integración de *TOL* con el sistema operativo para la ejecución de programas a través de la línea de comandos, a través de funciones como: `System` o `WinSystem`.

3.2.2 Obtención de *urls*

TOL dispone de varias funciones para la descarga de archivos remotos mediante protocolos http o ftp. Todas las funciones descargan el contenido referenciado por el url dado como argumento. El valor retornado es el contenido de la descarga.

Las funciones disponibles para este propósito son:

- `Text GetUrlContents.tcl.uri(Text url)` implementa la descarga basada en el paquete de *uri* de *Tcl*.
- `Text GetUrlContents.tcl.curl(Text url_)` implementa la descarga basada en la biblioteca *cURL*.

- `Text GetUrlContents.sys.wget(Text url)` implementa la descarga basado en la herramienta de línea de comandos *wget*.
- `Text GetUrlContents.tcom.iexplore(Text url)` implementa la descarga basado en el *API COM* de *Internet Explorer*.

También hay disponible una variante de estas funciones que se redirige a una de ellas de manera pre-configurada:

- `Text GetUrlContents(Text url)` es una función de descarga general que utiliza una de las funciones anteriores en función de la variable de configuración global `TolConfigManager::Config::ExternalTools::UrlDownloader` (véase la sección 3.3.1), la cual puede tomar uno de los siguientes valores:
  - `"tcl:uri"` para invocar `GetUrlContents.tcl.uri`.
  - `"tcl:curl"` para invocar `GetUrlContents.tcl.curl`
  - `"sys:wget"` para invocar `GetUrlContents.sys.wget` y
  - `"tcom:iexplore"` para invocar `GetUrlContents.tcom.iexplore`.

Obtener la *url* a través de *cURL*

La función `GetUrlContents.tcl.curl` descansa en la función `GetUrl` implementada en el módulo `CurlApi` (véase la sección 4.1.2). Esta función está basada en la herramienta *cURL* (http://curl.haxx.se) y ofrece la posibilidad de controlar parámetros de la descarga. La función tiene el siguiente prototipo:

```
Set CurApi::GetUrl(NameBlock args)
```

Donde `NameBlock args` puede contener los siguientes miembros:

- `Text url`: contiene el *url* que referencia al contenido que se solicita descargar.
- `Text file`: contiene la ruta donde se descargará el contenido. Es un argumento opcional, si no define este argumento entonces el contenido se retorna como parte del resultado de la función.
- `Real timeout`: establece el tiempo máximo en segundos de descarga. Nótese que normalmente el tiempo de resolución de nombres y de establecimiento de conexión puede tomar un tiempo considerable y si se establece este argumento a menos de unos pocos minutos se corre el riesgo de abortar operaciones perfectamente normales. Es un parámetro opcional y su valor por omisión `0.0` implica ningún chequeo de tiempo.
- `Real connectiontimeout`: establece el tiempo máximo en segundos dedicados al establecimiento de la conexión con el servidor. Sólo se considera durante la fase de conexión, y una vez que ya se establece la conexión esta opción no se usa más. Es un parámetro opcional y su valor por omisión `0.0` implica ningún chequeo de tiempo.
- `Real verbose`: es un indicador que, por cuando toma el valor verdadero (`!=0`), provoca la salida de mensajes por la salida estándar sobre el estado de las operaciones involucradas en la descarga. Es muy útil para la comprensión y depuración del protocolo de descarga.

La función retorna un conjunto que contiene información acerca de la transferencia ejecutada. Si la transferencia fue exitosa el conjunto contiene los siguientes campos:

- `Real connectTime`: el tiempo, en segundos, que tomó desde el inicio hasta completar la conexión con el host remoto.
- `Real fileTime`: la fecha remota del documento descargado (contado como el número de segundos desde el 1 de enero de 1970, en la zona horaria *GMT/UTC*). El valor de este campo puede ser -1 debido a varias razones (tiempo desconocido, el servidor no soporta el comando que informa sobre la fecha de un documento, etc.).
- `Real totalTime`: el tiempo total de la transacción, en segundos, para la transferencia, incluyendo la resolución de nombres, conexión *TCP*, etc.
- `Real sizeDownload`: la cantidad total de bytes que fueron descargados.
- `Real speedDownload`: la velocidad media de descarga, medida en bytes/s, para la descarga completa.
- `Text nameData`: el nombre del campo, dentro del propio conjunto, que contiene los datos descargados. Puede tomar dos valores, `"file"` si se solicitó que la descarga fuera escrita en un archivo o `"bodyData"` si en la llamada no se especificó el archivo destino.
- `Text file`: el archive donde debe almacenarse el contenido descargado.
- `Text bodyData`: el contenido descargado si no se ha especificado un archive destino.

Si la descarga no fue exitosa, el conjunto resultado contiene los siguientes campos:

- `Real failStatus`: un código de error.
- `Text explain`: una cadena con la descripción del error.

Ilustramos a continuación algunos ejemplos de uso de la función `CurApi::GetUrl`:

Ejemplo 1: Descargar un contenido http almacenarlo en una variables.

```
Set result1 = CurlApi::GetUrl([[
  Text url = "http://packages.tol-project.org/OfficialTolArchiveNetwork/"
    "repository.php?tol_package_version=1.1"
    "&tol_version=v2.0.1%20b.4&action=ping&key=658184943";
  Real verbose = 1;
  Real connecttimeout = 3
]]);
If(result1::failStatus, {
  WriteLn("CurlApi::GetUrl error: "<<result1::explain )
}, {
  WriteLn("CurlApi::GetUrl ok, the result is: "<<result1::bodyData)
} );
```

Ejemplo 2: Descargar un contenido http hacia un archivo.

```
Set result2 = CurlApi::GetUrl([[
  Text url = "http://packages.tol-project.org/OfficialTolArchiveNetwork/"
    "repository.php?action=download&format=attachment&package=BysMcmc.4.4";
  Text file = "/tmp/BysMcmc.4.4.zip",
  Real verbose = 1;
  Real connecttimeout = 3
]]);
If(result2::failStatus, {
  WriteLn("CurlApi::GetUrl error: "<<result2::explain )
}, {
  WriteLn("CurlApi::GetUrl ok, the result was downloaded to: "
    <<result2::file)
});
```

Ejemplo 3: Descargar un contenido ftp hacia un archivo.

```
Set result3 = CurlApi::GetUrl([[
```

```
  Text url = "ftp://ftp.rediris.es//pub/OpenBSD/README";
  Text file = "/tmp/README";
  Real verbose = 1;
  Real connecttimeout = 30
]]);
```

3.2.3 Acceso a base de datos (`DB<Function>`)

En las secciones 2.3.3 y 2.4.3 veíamos cómo podemos generar series temporales o matrices utilizando funciones elementales, datos introducidos manualmente o números aleatorios. A continuación describimos algunas de las funciones que dispone *TOL* para el acceso a bases de datos y que nos permiten obtener la información y construir variables.

TOL incorpora algunas funciones elementales para la comunicación con las bases de datos a través de *ODBC*:

- La función `DBOpen` abre una conexión con una base de datos.
- La función `DBClose` cierra una conexión con una base de datos.
- La función `DBTable` devuelve el resultado en forma de conjunto (`Set`) de una consulta `SELECT` de *SQL*.
- Las funciones `DBSeries`, `DBSeriesColumn` y `DBSeriesTable`, nos facilitan la creación de series temporales a partir de consultas `SELECT` de *SQL*.
- La función `DBMatrix` nos permite la creación de matrices a partir de consultas `SELECT` de *SQL*.
- La función `DBExecQuery` nos permite ejecutar en general todo tipo de sentencias *SQL* para la interacción con la base de datos.

Ejemplo. Lectura de archivos

A continuación describimos un ejemplo para usar las funciones de acceso a base de datos con un *ODBC* a una carpeta con archivos de datos planos:

Creación de los datos en archivo

Creamos una carpeta que hará las veces de base de datos. En el ejemplo usaremos: "C:/Data".

Allí ubicaremos los archivos de tipo .txt con los datos. Para ello creamos dos archivos con los datos siguientes. Podemos utilizar el bloc de notas de notas para crearlos.

data.txt

```
date,value1,value2
2000/01/01,210,162
2001/01/01,181,142
2002/01/01,192,158
2003/01/01,191,182
2004/01/01,172,144
2005/01/01,207,132
2006/01/01,205,159
2007/01/01,199,162
2008/01/01,205,158
2009/01/01,219,122
2010/01/01,186,125
```

multiple.txt

```
name,date,value1,value2
"A100",2000/01/01,102,312
```

```
"A100",2001/01/01,111,282
"A100",2002/01/01,128,315
"A100",2003/01/01,107,308
"A100",2004/01/01,131,322
"A100",2005/01/01,142,295
"A101",2000/01/01,52,112
"A101",2001/01/01,61,82
"A101",2002/01/01,28,85
"A101",2003/01/01,67,108
"A101",2004/01/01,81,122
"A101",2005/01/01,42,95
```

Nótese que estos datos sólo tienen un propósito educativo.

ODBC

A continuación creamos una conexión *ODBC* utilizando el driver de *Microsoft* para archivos de texto y le ponemos, por ejemplo, el nombre "TOLTest".

Esto puede hacerse manualmente utilizando las herramientas administrativas de *Windows* o ejecutando el siguiente código en la consola de comandos:

```
odbcconf configsysdsn "Microsoft Text Driver (*.txt; *.csv)"
  "DSN=TOLTest|defaultdir=C:\Data"
```

Obtención de datos

```
Real GlobalizeSeries := 0;
Real DBOpen("TolTest", "", "");
Set DBTable("SELECT * FROM data.txt");
Set DBSeries("SELECT date, value1 FROM data.txt", Yearly, [["A000"]]);
Set DBSeriesColumn("SELECT name, date, value1 FROM multiple.txt", Yearly);
Set DBSeriesTable("SELECT name, date, value1, value2 FROM multiple.txt",
  Yearly, [["_A", "_B"]]);
Real DBClose("TolTest");
```

3.3 Configuración de *TOL*

3.3.1 Módulo de configuración (`TolConfigManager`)

La configuración de *TOL* se gestiona desde el módulo `TolConfigManager`. Este *nameblock* alberga todo el conjunto de valores de configuración así como de los métodos para su uso y edición.

El `TolConfigManager` permite al usuario modificar sus preferencias en el uso de *TOL* utilizando el propio lenguaje y almacenarlo en el disco de una forma fácil y sencilla.

La configuración se conserva de una sesión de *TOL* para otra en un archivo de configuración en la carpeta de datos de la aplicación:

```
Text TolConfigManager::_.path
// => Text TolAppDataPath+".tolConfig."+Tokenizer(Version," ")[1]+".tol";
```

y se lee cada vez que se inicia *TOL* construyéndose el bloque: `TolConfigManager::Config`.

El bloque de configuración puede editarse (en caliente) como cualquier otra variable de tipo `NameBlock`, aunque es posible que las consecuencias de aplicar esa configuración no tengan efecto hasta que se abra una nueva sesión. Para guardar los cambios en la configuración y que tengan efecto en la siguiente sesión basta con llamar al método `Save` del `TolConfigManager`.

<u>Ejemplo:</u>

Por ejemplo, si tenemos una máquina sin conexión a internet es recomendable cambiar la configuración local de *TOL*, para evitar que el sistema pierda tiempo intentando conectarse. Esto puede hacerse manualmente mediante la evaluación del siguiente código:

```
Real TolConfigManager::Config::Upgrading::TolVersion::CheckAllowed := False;
Real TolConfigManager::Config::Upgrading::TolPackage::LocalOnly := True;
Real TolConfigManager::Save(?);
```

Si más adelante se tiene conexión puede deshacerse la configuración anterior mediante:

```
Real TolConfigManager::Config::Upgrading::TolVersion::CheckAllowed := True;
Real TolConfigManager::Config::Upgrading::TolPackage::LocalOnly := False;
Real TolConfigManager::Save(?);
```

3.3.2 Otros mecanismos de configuración

Las configuraciones de *TOL* más antiguas permanecen en variables globales o variables internas que se pueden editar a través de otros medios.

La configuración relacionada con la interfaz de *TOLBase* se almacena en el archivo `tolcon.ini` en el directorio *home* del usuario.

3.4 Sistema de paquetes

A partir de la versión 2 de *TOL* se ha implementado un mecanismo para la construcción y distribución automática de componentes de software reutilizable. En esta sección explicaremos qué es un paquete en *TOL* y cómo usarlo, así como otros aspectos relacionados con la gestión de los paquetes instalados y los repositorios de paquetes.

3.4.1 Paquetes

En la sección 2.5.1 presentábamos los bloques o *nameblocks* (`NameBlock`) como la gramática que permitía a *TOL* la posibilidad de orientarse a la modularidad, creando así espacios de definición independientes, capaces de albergar conjuntos de variables y funciones relacionados entre sí.

La idea de paquete surge de este concepto de modularidad y de la posibilidad de cargar y distribuir todo un módulo como una unidad. Esencialmente un paquete no es más que un *nameblock* distribuido en un archivo .oza. Véase la sección 3.1.3 para más detalles sobre este tipo de almacenamiento.

Empaquetamiento

En el marco de la programación en *TOL*, identificamos al paquete precisamente con este bloque (`NameBlock`) que encapsula el conjunto de variables, funciones y definiciones (de clases o estructuras) y que es distribuido y cargado como una unidad del sistema de paquetes de *TOL*.

Desde un punto de vista más cercano a su distribución, un paquete es un archivo comprimido (concretamente un archivo .zip) que contiene la serialización del *nameblock* (un archivo .oza) acompañado de otros recursos complementarios como pueden ser: archivos de iconos o imágenes, documentación, bibliotecas de funciones compiladas (escritas en *C++* y distribuidas en archivos .dll o .so) e incluso código *TOL* no compilado (archivos .tol) a modo de ejemplos.

Nombre y versión

Un paquete en *TOL* se identifica por su nombre y versión.

El nombre del paquete coincide con el nombre del *nameblock* que lo representa y por lo común sigue una estructura *CamelCase*.

La versión está formada por dos números enteros. El primero (el número de *versión alta*) indica si ha habido un cambio importante en la estructura y funcionalidad del paquete que puede romper compatibilidad con el código de usuario que lo usa, mientras que el segundo (el número de *versión baja*) indica que se han realizado cambios en el paquete, ya sean correcciones o la incorporación de nuevas funcionalidades, que mantienen la compatibilidad en el uso del paquete.

Identificador

El identificador del paquete se crea concatenando con un punto (`.`) el nombre y los dos números de versión: `<name>.<version.high>.<version.low>`.

Por ejemplo, el identificador `GuiTools.3.5` hace referencia a la versión 3.5 del paquete *GuiTools* que incorpora utilidades para la integración gráfica con *TOLBase*.

3.4.2 Instalación y uso de paquetes (`#Require`)

La manera más sencilla de cargar e instalar un paquete (si aún no estuviese instalado) es utilizar la directiva *TOL* `#Require` acompañada del nombre o identificador del paquete que se desea utilizar.

Por ejemplo:

```
#Require GuiTools;
```

El uso de directivas en *TOL* es algo relativamente excepcional y su comportamiento varía ligeramente del resto de sentencias, ya que actúan antes de comenzar la compilación o interpretación del resto de líneas de código.

Por ejemplo:

```
WriteLn("Line 1");
#Require GuiTools;
WriteLn("Line 2");
```

```
Ha sido cargado el paquete GuiTools.3.5
Line 1
Line 2
```

Concretamente la directiva `#Require` se encarga de cargar un paquete (si no está cargado) antes de compilar el resto del código, evitando así posibles errores sintácticos por el uso de clases o estructuras declaradas en el paquete.

Téngase en cuenta que la llamada a `#Require` intenta cargar el paquete por todos los medios, probando incluso a localizarlo en un repositorio de paquetes e instalarlo.

Carga de versiones

La directiva `#Require` nos permite especificar la versión del paquete que deseamos cargar, pudiendo así indicar, según nos interese: los dos números de versión, sólo el primero o únicamente el nombre del paquete:

- Si sólo indicamos el nombre del paquete, se cargará la última versión entre los que se encuentren instalados. Concretamente se cargará aquel paquete cuya *versión baja* sea la mayor entre aquellos que tienen la mayor *versión alta*.

```
#Require <PackageName>;
```

- Si indicamos el primer número de versión, se cargará la última versión del paquete cuyo número de *versión alta* coincida con el indicado.

```
#Require <PackageName>.<VersionHigh>;
```

- Si indicamos el identificador completo del paquete (el nombre y los dos números de versión) se cargará únicamente dicha versión del paquete.

```
#Require <PackageName>.<VersionHigh>.<VersionLow>;
```

Nótese que la directiva no intentará cargar el paquete si éste ya está cargado aún cuando la versión indicada no coincida con la del paquete cargado, ya que no pueden cargarse dos versiones distintas de un mismo paquete.

Compatibilidad con la versión de *TOL*

En ocasiones, los paquetes se construyen apoyándose en algunas modificaciones o mejoras del lenguaje que impiden que puedan funcionar en versiones anteriores de *TOL*. Por ello, algunos paquetes disponen de una versión mínima a partir de la cual se garantiza la compatibilidad.

Por ejemplo, la versión 3.5 de *GuiTools* necesita al menos la versión de 3.1 de *TOL*.

Téngase en cuenta que si una versión de un paquete no es compatible con la versión de *TOL* que estamos utilizando, permanecerá invisible, como si no existiera, para todos los mecanismos de instalación, carga o actualización de paquetes.

Auto-documentación del paquete

Un paquete (el `NameBlock`) incorpora entre sus miembros atributos cuya función es la auto-documentación del paquete. Entre ellos podemos destacar:

- `Text _.autodoc.name`: el nombre del paquete.
- `Real _.autodoc.version.high`: el primer número de la versión o versión alta.
- `Real _.autodoc.version.low`: el segundo número de la versión o versión baja.
- `Text _.autodoc.brief`: una breve descripción del paquete.
- `Text _.autodoc.description`: una descripción algo más detallada del paquete.
- `Set _.autodoc.keys`: un conjunto de palabras clave.
- `Set _.autodoc.authors`: el conjunto de autores del paquete.
- `Set _.autodoc.dependencies`: el conjunto de dependencias del paquete.
- `Text _.autodoc.minTolVersion`: versión mínima de *TOL* compatible.

3.4.3 Gestión de paquetes (TolPackage)

El módulo para la gestión de paquetes en *TOL* es TolPackage. Se trata de un *nameblock* estructurado a su vez en sub-módulos encargados de las distintas funcionalidades, como son: la gestión de los paquetes instalados, la interacción con los repositorios de paquetes, la construcción de paquetes, etc.

El módulo TolPackage dispone de números de versión (como si de un paquete se tratara) para facilitar la gestión de cambios y mejoras.

A continuación presentamos algunas nociones básicas en el uso de TolPackage, válidas desde su versión 2 (2.X) que se distribuye con *TOL* desde la versión 3.1 (concretamente v3.1 p011).

Instalación de paquetes

Un paquete se instala automáticamente la primera vez que es requerido (#Require), sin embargo, la instalación puede realizarse manualmente a través de la función InstallPackage indicando el identificador (nombre y versión) del paquete que se desea instalar:

```
Real TolPackage:InstallPackage(<packageIdentifier>);
```

SI no conocemos la versión última podemos usar la función InstallLastCompatible e indicar simplemente el nombre del paquete:

```
Real TolPackage:InstallLastCompatible(<packageName>);
```

Actualización y mejora (*update & upgrade*)

Cada cierto tiempo, se crean nuevas versiones de un paquete, de modo que para mantenernos al día hemos de actualizarnos.

El sistema de paquetes identifica dos mecanismos de puesta al día de los paquetes:

- La actualización (*update*) de un paquete, que consiste en la sustitución del paquete por otro con los mismos números de versión, pero que es más reciente.
- La mejora (*upgrade*) de un paquete, que consiste en la instalación (sin desinstalar la anterior) de una nueva versión del paquete.

La actualización de un paquete suele incorporar correcciones importantes, ya que la reconstrucción de un nuevo paquete con la misma versión descarta la anterior, por lo tanto es importante siempre estar actualizado.

Para actualizar un paquete podemos utilizar la función Update indicando el nombre del paquete:

```
Real TolPackage::Update(<packageName>);
```

También podemos actualizar todos los paquetes que tengamos instalados con la función UpdateAll:

```
Real TolPackage::UpdateAll(?);
```

La mejora de un paquete, es igualmente recomendable, ya que en muchos casos incorporan importantes avances en los procedimientos y algoritmos implementados.

Para mejorar un único paquete disponemos de la función Upgrade:

```
Real TolPackage::Upgrade(<packageName>);
```

Y de un modo similar, la función `UpgradeAll`, para mejorar todos los paquetes instalados:

```
Real TolPackage::UpgradeAll(?);
```

Mejora de la versión baja (*low-upgrade*)

En alguna ocasión ocurre, que queremos mejorar la *versión baja* (segundo número de versión) de un paquete manteniendo la alta. Para ello disponemos de la función `LowUpgrade`, a la que podemos llamar indicando el nombre del paquete y el primer número de versión (*versión alta*):

```
Real TolPackage::LowUpgrade(<packageName.packageVersion>);
```

Téngase en cuenta que si indicásemos sólo el nombre del paquete se intentaría mejorar la versión baja de todas las diferentes versiones altas del paquete que se encuentren instaladas.

Si deseamos incorporar todas las mejoras de un paquete, disponemos de la función `FullUpgrade` que aúna las llamadas a `Upgrade` y `LowUpgrade`:

```
Real TolPackage::FullUpgrade(<packageName>);
```

También están disponibles las versiones para todos los paquetes instalados: `LowUpgradeAll` y `FullUpgradeAll`.

3.4.4 Repositorios de paquetes

Hasta ahora hemos mencionado los repositorios de paquetes sin decir mucho sobre ellos. Los repositorios de paquetes no son más que los sitios donde se almacenan y desde los que se distribuyen los paquetes para su instalación y uso.

La arquitectura interna de los repositorios, no es de mucho interés para su uso, nos basta saber que disponen de una interfaz *PHP* indispensable para la comunicación y obtención de paquetes.

Por ejemplo, la *URL* correspondiente al repositorio oficial de paquetes de *TOL* es: http://packages.tol-project.org/OfficialTolArchiveNetwork/repository.php (ruta *php*).

Repositorios conocidos

El módulo `TolPackage` nos facilita esta comunicación con los repositorios, dirigiéndose a ellos para la instalación, actualización o mejora de los paquetes.

La lista de repositorios conocidos a los que consulta `TolPackage` se encuentra en la variable de configuración:

```
Set TolConfigManager::Config::Upgrading::TolPackage::Repositories;
```

Por defecto, esta variable sólo contiene la URL del repositorio oficial de paquetes en *TOL*, aunque puede editarse a través de los mecanismos habituales de `TolConfigManager` (véase la sección 3.3.1) o usando la función `AddRepository` de `TolPackage`:

```
Real TolPackage::AddRepository(<newRepositoryURL>);
```

Página web del repositorio

Habitualmente, los repositorios también disponen de una página web que permite a los usuarios conocer la lista completa de paquetes disponibles, inspeccionar todas sus características e incluso descargarlos.

La página web del repositorio oficial de paquetes de *TOL* conocido como *OTAN* (*Official Tol Archive Network*) se encuentra en la siguiente página de la wiki de su *trac*: https://www.tol-project.org/wiki/OfficialTolArchiveNetwork (ruta *wiki*).

3.4.5 Interfaz gráfica del gestor de paquetes

Desde la versión 3, *TOLBase* incorpora una interfaz gráfica específica para facilitar la gestión de paquetes, accesible desde el menú Herramientas en la opción Gestiona Paquetes....

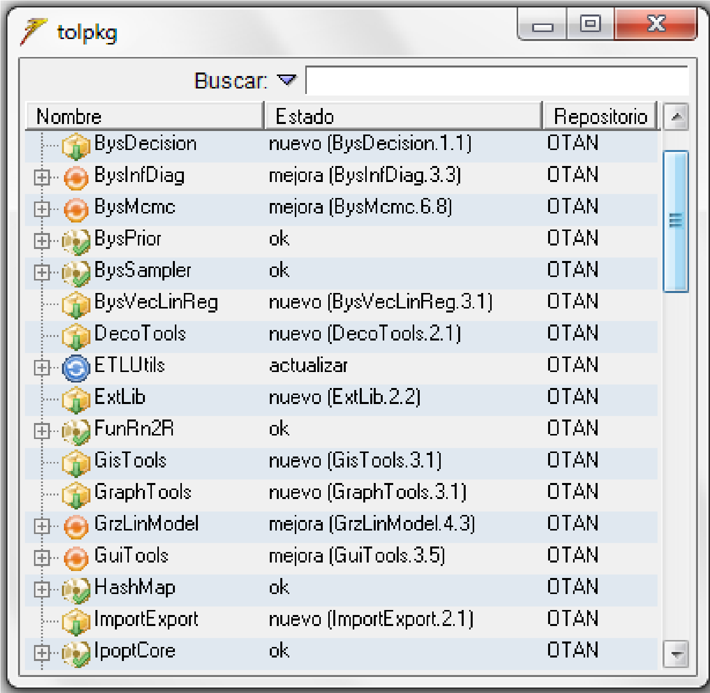

Figura 3.4.1: Interfaz del gestor de paquetes de *TOLBase*.

La interfaz ofrece una lista con todos los paquetes disponibles de todos los repositorios a los que se tiene acceso para instalar, actualizar o mejorar.

A través de los distintos iconos se caracterizan los distintos estados de un paquete:

- : Es nuevo, no está instalado.
- : Necesita actualizar, hay un paquete con idéntica versión pero más reciente.
- : Necesita mejorar, hay disponible un paquete con una versión superior.
- : Necesita mejorar y actualizar, si a un tiempo necesita tanto actualización como mejora.
- : Está OK, está instalado y no necesita ni actualización ni mejora.

Las distintas acciones que pueden realizarse sobre los paquetes se ofrecen en el menú contextual.

3.4.6 Código fuente de los paquetes

El código fuente de los paquetes de *TOL* se gestiona a través de un sistema de control de versiones denominado *SVN* (véase: http://subversion.apache.org) vinculado al *trac* del proyecto.

Servidores de código públicos

El desarrollo de *TOL* dispone hoy en día de dos *tracs* públicos de desarrollo diferentes:

- El *trac* principal de *TOL-Project*: https://www.tol-project.org.
- El *trac* de *MMS* (*Model Management System*): https://mms.tol-project.org.

Cada uno de estos *tracs* tiene su servidor de código *SVN* correspondiente, aunque comparten repositorio de paquetes. Véase la sección 3.4.4.

Para más información sobre *MMS* consúltese su manual correspondiente.

Descargar el código (ruta *svn*)

El código de los paquetes se encuentra organizado en carpetas con el nombre de cada paquete en la ruta: https://www.tol-project.org/svn/tolp/OfficialTolArchiveNetwork (ruta *svn*).

Para descargar el código necesitamos utilizar un programa cliente del *SVN*.

Por ejemplo, para descargar todo el conjunto de paquetes de *OTAN* desde la línea de comandos a un directorio local escribiremos:

```
svn co https://www.tol-project.org/svn/tolp/OfficialTolArchiveNetwork/
  c:/users/<user>/svn/OTAN
```

En lugar de utilizar instrucciones en línea de comandos, también podemos utilizar alguna aplicación con interfaz gráfica como *TortoiseSVN* (http://tortoisesvn.net).

Consultar del código (ruta *browser*)

Si simplemente queremos consultar algún detalle, o nos interesa ver los cambios de algunas funcionalidades en relación a los tiques, hitos (*milestones*) o componentes del *trac*, podemos consultar el código desde el explorador web que dispone el *trac* en: https://www.tol-project.org/browser/tolp/OfficialTolArchiveNetwork (ruta *browser*).

4 Paquetes de *TOL*

En la sección 3.4 se ha presentado todo lo relativo al sistema de paquetes de *TOL*, su concepción y su uso. En este capítulo describiremos las funcionalidades implementadas en algunos de los paquetes vinculados al desarrollo de *TOL* y a su trac principal *TOL-Project*.

No describiremos todos los paquetes disponibles en la *OTAN*, sino sólo aquellos que se consideran de mayor utilidad. En la última sección del capítulo hacemos referencia a un conjunto mayor de paquetes indicando su propósito e indicando referencias a sitios de interés relacionados con cada paquete.

Recuérdese que la lista completa de paquetes disponibles en el repositorio de paquetes públicos de *TOL* conocido como *OTAN* (*Official Tol Archive Network*) puede consultarse en: https://www.tol-project.org/wiki/OfficialTolArchiveNetwork. Véase la sección 3.4.4.

El código de estos paquetes puede descargarse del *SVN* de *TOL-Project*:
https://www.tol-project.org/svn/tolp/OfficialTolArchiveNetwork (ruta *svn*)
o consultarse desde la web del *trac* en:
https://www.tol-project.org/browser/tolp/OfficialTolArchiveNetwork (ruta *browser*).
Para más detalles, consúltese se la sección 3.4.6.

Todos los paquetes se encuentran en la ruta del servidor de código organizados en directorios con el nombre del paquete. Por ejemplo, el paquete `GuiTools` puede localizarse en:
https://www.tol-project.org/svn/tolp/OfficialTolArchiveNetwork/GuiTools (ruta *svn*) o en:
https://www.tol-project.org/browser/tolp/OfficialTolArchiveNetwork/GuiTools (ruta *browser*).
En adelante indicaremos simplemente: OTAN/GuiTools.

4.1 `StdLib`

En los inicios de *TOL* la unidad básica de encapsulamiento era el archivo, y para hacer uso de una funcionalidad concreta se incluían las funciones relacionadas con esa funcionalidad a partir de un conjunto de archivos dados.

Para facilitar el acceso a las funciones más comunes ya estandarizadas por parte de los usuarios se distribuía junto con la instalación de *TOL* un directorio que contenía un conjunto de archivos con la implementación de dichas funciones. A este conjunto de funciones se le denominó *StdLib*, debido al término en inglés *Standard Library*.

Paquete `StdLib`

Este esquema de organización basado en archivos es muy inflexible y con la aparición de la modularización basada en paquetes de *TOL* se pasó a una distribución de la *StdLib* como un paquete: `StdLib`.

Actualmente `StdLib` es un gran módulo que contiene un conjunto de funciones para usos muy diversos, de ahí que en un futuro y en aras de ganar más en modularidad vayamos dividiéndola en paquetes más pequeños y atómicos dedicados a funcionalidades más concretas.

Bloque `TolCore`

Algunas de las funciones, estructuras de datos y constantes no pudieron ser extraídas de la distribución de *TOL*, bien porque estaban muy ligadas a algunas funciones internas de *TOL*, bien porque eran esenciales para la carga de la propia de `StdLib`, como es el caso del módulo interno `TolPackage`. Este conjunto de funciones han permanecido agrupadas en un *nameblock* interno denominado `TolCore`.

Todas las funciones miembros de estos *nameblocks*, tanto las de la `StdLib` como las de `TolCore` son exportadas automáticamente al ámbito global para facilitar la compatibilidad con los códigos más antiguos.

4.1.2 Bloque `TolCore`

El código correspondiente al bloque `TolCore` sigue ubicado en un directorio denominado stdlib junto al código del propio lenguaje. Consúltese la ruta del servidor de código:
https://www.tol-project.org/svn/tolp/trunk/tol/stdlib (ruta *svn*) o
https://www.tol-project.org/browser/tolp/trunk/tol/stdlib (ruta *browser*),
en adelante simplemente stdlib.

Entre las funcionalidades que han permanecido en `TolCore` encontramos:

- **Funciones para la gestión de paquetes:** implementadas en el módulo `TolPackage`. Véase la sección 3.4.3. El código fuente se encuentra en: stdlib/TolPackage.
- **Estructuras internas:** estructuras (`Struct`) referenciadas en la función interna `Estimate` usada en la estimación de modelos `ARIMA`. Véase la sección 2.4.6. El código fuente se encuentra en: stdlib/arima.
- **Funciones de configuración de *TOL*:** implementadas en el módulo `TolConfigManager`. Véase la sección 3.3.1. El código fuente se encuentra en: stdlib/TolConfigManager.
- **Funciones para la descarga de archivos remotos:** implementadas en el módulo `CurlApi`. Véase la sección 3.2.2. El código fuente se encuentra en: stdlib/system/CurlApi.
- **Funciones para la lectura y escrituras de archivos y directorios comprimidos:** implementadas en el módulo `PackArchive`. El código fuente se encuentra en: stdlib/system/PackArchive.

4.1.3 Paquete `StdLib`

Por otra parte, como se mencionó anteriormente el paquete `StdLib` contiene funciones para usos muy diversos y está siendo revisado en busca de una mayor modularización. En futuras versiones su contenido será particionado en varios paquetes.

A continuación enumeramos algunas de las funcionalidades disponibles actualmente:

- **Acceso a bases de datos**: implementado en el módulo `DBConnect`. Se ofrecen mecanismos unificados para establecer conexiones a diferentes bases de datos. El código fuente puede encontrarse en: OTAN/StdLib/data/db.
- **Funciones de interpolación basados en el *API* de *GSL* y *AlgLib*.** El código fuente puede explorarse en: OTAN/StdLib/math/interpolate.

- ***Solvers* lineales precondicionados**: contiene rutinas para la solución de sistemas lineales simétricos y no simétricos mal condicionados. Podemos revisar el código fuente en OTAN/StdLib/math/linalg. Explórense también los objetos:
    - `Code StdLib::SolvePrecondSym`
    - `Code StdLib::SolvePrecondUnsym`
- **Funciones de optimización**: usando *Marquardt*, programación lineal (basado en el paquete *Rglpk* de *R*, `StdLib::Rglpk`) y programación cuadrática (basado en *Rquadprog* de *R*, `StdLib::Rquadprog`). Podemos encontrar el código fuente en: OTAN/StdLib/math/optim.
- **Funciones para la invocación de *R* desde *TOL***: implementa funciones que facilitan la invocación de funciones escritas en *R*. Podemos ver dos ejemplo de uso en la implementación de las funciones `StdLib::Rglp::solveLP` y `StdLib::Rquadprog::solveQP`. El código fuente puede explorarse en OTAN/StdLib/math/Rapi.
- **Funciones para la estimación de densidad univariada**: basado en la función *density* de *R*. Explórese el módulo `StdLib::Rkde`. El código fuente podemos encontrarlo en: OTAN/StdLib/math/stat/kde.

4.2 `GuiTools`

`GuiTools` es un paquete que permite utilizar y extender las funcionalidades propias de la interfaz gráfica (*GUI*: *Graphical User Interface*) usando lenguaje *TOL*.

Aunque se concibe independiente de la plataforma gráfica, la actual implementación funciona sólo completamente para el interfaz de gráfico *TOLBase* que está implementado en *Tcl-Tk*.

Entre las funcionalidades más utilizadas de este paquete están los módulos dedicados a:

- La gestión y personalización de imágenes: `GuiTools::ImageManager` y
- La gestión y personalización de menús contextuales: `GuiTools::MenuManager`.

4.2.1 Gestión de imágenes (`ImageManager`)

Registro de imágenes

El gestor de imágenes implementa dos funciones, dentro del módulo `ImageManager`, que permiten registrar imágenes a partir de un archivo o de un texto con la imagen codificada usando *base64*:

- `Real defineImageFromFile(Text imageName, Text imagePath)` registra una imagen contenida en archivo. La imagen puede referenciarse posteriormente por el identificador dado en `imageName`.
- `Real defineImageFromData(Text imageName, Text imageData)` registra una imagen a partir de un texto que contiene la codificación en *base64* de los datos de la imagen. La imagen puede referenciarse posteriormente por el identificador dado en `imageName`.

<u>Ejemplo:</u>

Para obtener una imagen codificada en *base64*, *TOL* dispone de una función que permite obtener dicha codificación desde un archivo: `EncodeBase64FromFile`. Por ejemplo:

```
Text image_base64 = EncodeBase64FromFile(".../image.gif");
```

El contenido de la variable resultante ha de pasarse como argumento a la función `defineImageFromData` para registrar la imagen, por ejemplo:

```
Real GuiTools::ImageManager::defineImageFromData("my_image", image_base64);
```

Asignación de iconos a clases

Los objetos que son instancias de clase se muestran en el inspector de objetos de con un icono que es común a todos los objetos de tipo `NameBlock`, esto, a menudo, dificulta la diferenciación visual entre objetos de instancias distintas. `ImageManager` contiene una función para poder asociar un icono las instancias de una clase:

- `Real setIconForClass(Text className, Anything imageName)` define la imagen que se mostrará en el inspector de objetos para las instancias de la clase con nombre `className`. El argumento `imageName` puede ser de tipo `Text` o `Code`, cuando es de tipo `Text` se interpreta como el nombre de una imagen registrada, cuando es de tipo `Code` se interpreta como un objeto función que retorna el nombre de la imagen registrada, en este caso la función recibe como argumento la instancia seleccionada y puede decidir la imagen apropiada en función del estado del objeto.

<u>Ejemplo:</u>

A continuación ilustramos mediante un ejemplo cómo personalizar una clase asignándole un icono.

```
#Require GuiTools;

// Se crea una clase:
Class @TestA {
  Real value
};

// Se registra un icono a través de su codificación base64:
Real GuiTools::ImageManager::defineImageFromData("checkedBox",
  "R0lGODdhCwALAJEAAH9/f////wAAAP///ywAAAAACwALAAACLISPRvEPAE8oAMUXCYAg"
  "JSEiAYRIQkSCAgTJjgiAoEgSEQGEJIRiA9wdwUcrADs=");

// Se asigna el icono a la clase creada
Real GuiTools::ImageManager::setIconForClass("@TestA", "checkedBox");

// Se crea una instancia de la clase
@TestA inst1 = [[ Real value = 1.0 ]];
```

4.2.2 Gestión de menús contextuales (`MenuManager`)

Al presionar el botón derecho del ratón sobre el panel de variables del inspector de objetos de *TOLBase* se despliega un menú contextual con opciones relativas a la selección activa, véase la figura 4.2.1. La selección activa puede contener uno o varios objetos *TOL* y las entradas de este menú contextual dependen de la composición de esa selección.

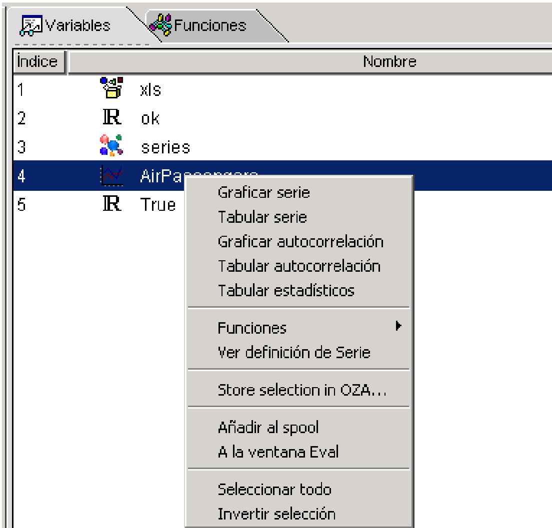

Figura 4.2.1: Menú contextual desplegado sobre un objeto de tipo `Serie`.

Cuando hay una selección múltiple que involucra a objetos de diferentes tipos el menú contextual se organiza en submenús, uno por cada tipo de dato.

Con el módulo `MenuManager` es posible extender el conjunto de opciones de menú contextual disponibles por omisión en *TOLBase*.

Argumentos para definir un menú contextual

Cuando definimos una opción de menú debemos especificar un conjunto de argumentos que configuran su apariencia y funcionalidad. El constructor de opciones de menú recibe un *NameBlock* que contiene los argumentos de la opción de menú que se define. El único argumento que es requerido a la hora definir una opción de menú es su nombre (`Text name`) que además debe ser único para dicha opción de menú. Por ejemplo:

```
NameBlock menuOption_arguments = [[
  Text name = "Entrada1"
]];
```

Los atributos que configuran la apariencia visual de una entrada de menú son:

- `Text label`: contiene la etiqueta que se muestra en el menú para la entrada definida. Si no se define se asume igual al nombre.
- `Text image`: contiene el nombre de una imagen registrada en `ImageManager` (véase la sección 4.2.1), por omisión toma el valor `""` y no se muestra imagen en la entrada de menú.
- `Real rank`: contiene un valor numérico que es usado para ordenar las entradas dentro un mismo submenú. Por omisión asume el valor `0`. Las entradas con idéntico valor de `rank` se ordenan según el orden de definición.

Los atributos que configuran la funcionalidad de una entrada de menú son:

- `Real flagGroup`: valor booleano que especifica si la acción se aplica sobre una instancia o un conjunto de instancias. Si es `0` (falso) la opción aparece disponible en el menú cuando la selección activa contiene sólo un objeto del tipo asociado; en caso

contrario, si es: `!=0` (verdadero), aparece disponible en el menú cuando la selección activa contiene más de un objeto del tipo asociado.

- `Code|Set CmdInvoke`: puede ser un objeto `Code` o un `Set` que contiene un objeto `Code`. El objeto `Code` es una función de usuario que debe retornar un `Real` (el cual es ignorado) y se utiliza como acción a invocar cuando la opción de menú es seleccionada. La función invocada recibe como primer argumento el objeto (si `flagGroup` es falso) o conjunto de objetos (si `flagGroup` es verdadero) contenidos en la selección activa. Además recibe como segundo argumento un conjunto (`Set`) de parámetros extras propios de la opción. Este conjunto de parámetros extras se especifica a la hora de definir la opción.
- `Code|Set CmdCheck`: puede ser un objeto `Code` o un `Set` que contiene un objeto `Code`. El objeto `Code` es una función de usuario que es invocada por el gestor de menú para decidir el estado con el que se muestra la opción de menú. La función debe retornar `!=0` (verdadero) para indicar que la opción debe aparecer habilitada ó `0` (falso) para indicar que la opción debe aparecer deshabilitada. Recibe los mismos argumentos que la función referenciada en el atributo `CmdInvoke`.
- `Text delegateOn`: el valor del atributo `delegateOn` es una expresión que al ser evaluada sobre el objeto objetivo retorna como resultado el objeto sobre el cual se invocará la acción (`Check` o `Invoke`).Con este atributo podemos definir opciones para un tipo de objeto y ejecutar la acción sobre un miembro del objeto.

A continuación describimos las funciones disponibles dentro del módulo `MenuManager` para la definición de opciones de menú contextual asociadas a los tipos de datos.

- `Real defineMenuCommand(Text typeName, NameBlock args)`: define una opción de menú disponible para objetos del tipo de dato dado en `typeName`. No es necesario que exista el tipo de dato en el momento de definir la opción. El argumento `NameBlock args` contiene los atributos que especifican la apariencia visual y la acción asociada a la opción.
- `Real replaceMenuCommand(Text typeName, NameBlock args)`: similar a `defineMenuCommand` pero si la opción ya está definida reemplaza su definición con los nuevos atributos dados en `args`.
- `Real defineOptionLabel(NameBlock args)` : define la apariencia visual de una etiqueta de menú, como puede ser la etiqueta usada para los submenús de los tipos de datos cuando la selección activa contiene varios tipos de datos para los cuales existen opciones de menú definidas o para las etiquetas de submenús inferidos del nombre de una opción de menú (ver más adelante la organización en submenús).
- `Real replaceOptionLabel(NameBlock args)` : similar a `defineOptionLabel` pero si la opción ya está definida reemplaza su definición con los nuevos atributos dados en `args`.

Las opciones de menú pueden ser compartidas por más de un tipo de dato. Para ello sólo debemos definir la opción de menú una vez para el primer tipo de dato. Para el resto de tipos de datos simplemente indicaremos como argumento el nombre de la opción de menú ya creada anteriormente.

Submenús contextuales

Las entradas del menú contextual podemos organizarlas en submenús. Para ello especificamos en el nombre la estructura jerárquica de submenús usando el separador /.

Por ejemplo, el nombre de entrada "Padre1/Padre2/Entrada" define a "Padre1" como una entrada de menú de tipo submenú en el nivel superior, "Padre1/Padre2" como un entrada de menú de tipo submenú dentro del submenú "Padre1" y "Padre1/Padre2/Entrada" como una entrada de menú dentro del submenú "Padre1/Padre2". La apariencia visual de las entradas asociadas a los submenús "Padre1" y "Padre1/Padre2" puede configurarse con una de las funciones defineOptionLabel o replaceOptionLabel.

Ejemplo:

Finalizamos esta sección con un ejemplo sencillo que ilustra esta funcionalidad de MenuManager.

```
#Require GuiTools;

// Queremos asociar una nueva opción de menú a los reales

// Creamos una función que recibirá el número real
// y un conjunto de argumentos extra que no usaremos:
Real RealSquare(Real x, Set args) {
  Real sqX = x*x;
  WriteLn( "El cuadrado de " << x << " es " << sqX );
  sqX
};

// Creamos la opción de menú para el tipo de datos "Real"
Real GuiTools::MenuManager::defineMenuCommand("Real", [[
  Text name = "Real_SQ";
  Text label = "Cuadrado del Real";
  Real flagGroup = 0;
  Set CmdInvoke = [[ RealSquare ]]
]]);
```

4.2.3 Edición de un contenedor

Otra de las funcionalidades incorporadas a GuiTools es aquella que permite editar interactivamente los datos miembros de un Set o NameBlock, que aquí denominaremos *contenedor*. Cuando se invoca la función de edición esta abre una ventana de diálogo donde podemos editar los valores asociados a los datos miembros del contenedor.

La edición puede operar en dos estados: modal o no modal. En el estado no modal, la función de edición abre la ventana y retorna inmediatamente continuando la evaluación con las instrucciones siguientes, mientras la ventana de edición permanece activa. Cuando se ejecuta en estado modal la función no retorna hasta que no cerremos la ventana de edición.

Por ejemplo, en la figura 4.2.2 se muestra la ventana que resultaría de invocar la edición de los elementos de un Set o un NameBlock con dos datos miembros.

Figura 4.2.2: Ejemplo de la ventana de edición de un objeto contenedor: `Set` o `NameBlock`.

Las funciones que permiten la edición de los datos miembros de un contenedor son:

- `NameBlock TkNameBlockEditor(NameBlock options, Set args)` abre una ventana que contiene una tabla donde se pueden editar los valores de los datos miembros del `NameBlock options`.
- `Set TkSetEditor(Set options, Set args)` abre una ventana que contiene una tabla donde se pueden editar los valores de los datos miembros del `Set options`.

El argumento `Set args` es un conjunto con cardinalidad par. Los elementos en posición impar se interpretan como el nombre de una opción y el elemento siguiente como su valor, por ejemplo `[["-modal", "yes", ... ]]` especifica la opción con nombre `"-modal"` y valor `"yes"`. En ambas funciones el argumento `Set args` define el estado en el que se abre la ventana:

- **modal**: cuando el argumento `args` es el conjunto vacío entonces se realiza una edición modal. También se puede explicitar con `Set args = [["-modal", "yes"]]`.
- **no modal**: se especifica incluyendo en los argumentos la opción `"-modal"` con valor `"no"`. Además deben incluirse los las opciones:
  - `"-address"`: con valor igual a la dirección física del objeto que se va a editar. La dirección física de un objeto se obtiene mediante la función `GetAddressFromObject`.
  - `"-tolcommit"`: con valor igual al nombre de una función que se invocará al aceptar los datos editados, esta función recibe como primer argumento el objeto original que se está editando y una copia con los datos editados.

Además de las opciones anteriores se pueden usar las siguientes:

- `"-showbuttons"`: determina si se muestran los botones Aceptar/Cancelar de la ventana. Puede tomar valores booleanos `0` ó `1` de tipo `Real` o `"yes"` o `"no"` de tipo `Text`. Por omisión toma el valor `1`, es decir muestra los botones.
- `"-title"`: toma un valor de tipo `Text` que es usado como título de la ventana. Por omisión se usa uno de los valores `"Edit Set"` o `"Edit NameBlock"` dependiendo si el objeto editado es un `Set` o `NameBlock` respectivamente.
- `"-checkchanges"`: determina si al cerrar la ventana se advierte de que hay cambios pendiente que no han sido guardados en el objeto original. Puede tomar valores booleanos `0` ó `1` de tipo `Real` o `"yes"` o `"no"` de tipo `Text`. Por omisión toma el valor `1`, es decir si al cerrar la ventana hay cambios pendientes se avisa al usuario y se le da la oportunidad de guardar los cambios antes de cerrar la ventana.

Los siguientes ejemplos ilustran el uso de las funciones de edición descritas:

Ejemplo de editor en forma modal:

```
#Require GuiTools;
Set s1 = { [[ Real a = 1; Real b = 2  ]] };
Set s2 = GuiTools::TkSetEditor(s1, Empty);
```

Ejemplo de editor en forma no modal:

```
#Require GuiTools;
Set s1 = { [[ Real a = 1; Real b = 2  ]] };

// Función invocada en el evento aceptar de la ventana de edición:
Real ApplyChanges(Set to, Set from) {
  to::a := from::a;
  to::b := from::b;
  Real 1
};

Set s2 = GuiTools::TkSetEditor(s1, [[
  "-title", "Título",
  "-modal", "no",
  "-tolcommit", "ApplyChanges",
  "-address", GetAddressFromObject(s1)
]]);

WriteLn("Sigue la ejecuación...");
```

La ventana que se abre en los dos ejemplos anteriores son similares, la única diferencia está en que en el primero la evaluación se detiene hasta que se cierre la ventana de edición mientras que en el segundo la evaluación del código continúa.

4.3 TolExcel

El paquete `TolExcel` implementa funciones para la entrada y salida desde archivos en formato binario *Excel 97-2003* (.xls). A continuación veremos los elementos básicos en el uso de `TolExcel`.

`TolExcel` define una clase principal denominada `@WorkBook` que nos sirve para instanciar objetos asociados a archivos *Excel*. Los métodos de clase que permiten crear instancias de `@WorkBook` son `Open` y `New`:

- `@WorkBook xls = @WorkBook::Open( Text path );` crea una instancia a partir de un archivo existente, con la ruta indicada en el argumento `path`. El archivo no puede estar abierto simultáneamente por *Excel* pues `TolExcel` solicita un acceso del archivo exclusivo.
- `@WorkBook xls = @WorkBook::New( Text path, Anything wsInfo );` crea una instancia de un nuevo archivo, el archivo nuevo se creará en la ruta indicada en el argumento `path`. El argumento `wsInfo` especifica las hojas de trabajo que contendrá el archivo. Si `wsInfo` es un `Real` se creará un objeto con esa cantidad de hojas, si es un conjunto (`Set`) entonces se crearan tantas hojas como elementos tiene el conjunto. Si el i-ésimo elemento del conjunto es un texto (`Text`) entonces la i-ésima hoja tendrá por nombre el valor de ese elemento, en caso contrario se asumirá el nombre por omisión que asigna *Excel*.

Los cambios hechos en una instancia de `@WorkBook` pueden ser almacenados en disco usando los siguientes métodos:

- `Real xls::Save(Real void)`: guarda los cambios hechos a la instancia `xls` en el archivo asociado a la instancia.
- `Real xls::SaveAs(Text path)`: guarda los cambios hechos a la instancia `xls` en el archivo indicado en la ruta `path`. A partir de ese momento el archivo especificado en el argumento `path` se asocia a la instancia de forma tal que futuras invocaciones del método `Save` tienen efecto sobre ese archivo.

La mayorías de los métodos de una instancia `xls` de `@WorkBook` operan sobre celdas o rangos de celdas relativos a la hoja de trabajo activa. Por omisión una instancia de `@WorkBook` estable la primera hoja de trabajo como la activa. Si queremos cambiar de hoja activa podemos invocar los métodos `ActivateWS` y `ActivateNamedWS`:

- `Real xls::ActivateWS(Real workSheetNumber)` : cambia la hoja activa, la nueva hoja activa es aquella que aparece en el índice igual al valor del argumento `workSheetNumber`. Los índices válidos van desde 1 hasta el número de hojas de la instancia. El método retorna `True` si se ha podido cambiar la hoja activa, o `False` en caso contrario.
- `Real ActivateNamedWS(Text workSheetName)` : establece como hoja activa aquella cuyo nombre coincida con el valor del argumento `workSheetName`. El método retorna `True` si se ha podido cambiar la hoja activa, o `False` en caso contrario.

Después de trabajar con la instancia, debemos cerrar la conexión con el archivo mediante el método `Close`:

- `Real xls::Close(Real void)` : cierra la conexión con el archivo físico liberando todos los recursos asociados. A partir de la invocación a `Close`, la instancia permanece inválida. Es responsabilidad del programador invocar previamente a uno de los métodos `Save` o `SaveAs` para actualizar el contenido del archivo con los cambios hechos en memoria.

Hay algunas funciones de lectura de celdas y rangos de celdas que permiten leer los valores almacenados sin especificar un tipo de dato para el resultado en TOL. En estos casos se intenta inferir lo mejor que puede el tipo de dato que le corresponde en TOL. Los tipos de datos posibles que podemos obtener son `Real`, `Text` y `Date`. Si no se reconocer un tipo de dato de los anteriores entonces se retorna el valor `Text ""`.

Las funciones de lectura y escritura reciben como argumentos una referencia a la celda o rango de celda siguiendo el siguiente convenio:

- **celda**: dado por un par de índices enteros `(row,col)` donde `row` indica la fila y `col` el índice numérico de la columna, considerando el valor 1 como el índice de la columna `A` y así sucesivamente. Por ejemplo, la celda `B3` tiene como coordenadas `(3,2)`.
- **rango**: dado por dos pares, esquina y extensión. La esquina del rango es `(row_ini,col_ini)` que indica las coordenadas de una celda y la extensión dada por `(row_num,col_num)` que indica que el rango se extiende desde la celda inicial `row_num` filas hacia abajo y `col_num` hacia la derecha. Por ejemplo el rango `A1:D1` se especificaría como `(row_ini,col_ini)=(1,1)` y `(row_num,col_num)=(1,4)`.

Las siguientes son algunas de las funciones usadas para leer valores desde las celdas de la hoja de trabajo activa:

- `Set xls::ReadRange(Real row_ini, Real col_ini, Real row_num, Real col_num, Set colDef)` : lee el rango de celdas especificado por la celda de inicio en coordenadas `(row_ini, col_ini)` y extendido `row_num` filas hacia abajo y `col_num` hacia la derecha. El resultado es un conjunto donde cada elemento es a la vez un conjunto que se corresponde con una fila del rango leído.
- `VMatrix ReadVMatrix(Real row_ini, Real col_ini, Real row_num, Real col_num)` : lee la matriz contenida en el rango de celdas especificado por la celda de inicio en coordenadas `(row_ini, col_ini)` y extendido `row_num` filas hacia abajo y `col_num` hacia la derecha. El resultado es un objeto `VMatrix`. Las celdas vacías o con contenido no numérico se leen como el valor omitido.
- `Matrix ReadMatrix(Real row_ini, Real col_ini, Real row_num, Real col_num)` : similar a `ReadVMatrix`, excepto que el objeto resultante es de tipo `Matrix`.
- `Set ReadSeriesByCol(Real row_ini, Real col_ini, Real row_num, Real col_num, TimeSet dating, Text dateFormat)`: lee un conjunto de series contenidos en el rango especificado por los argumentos (`row_ini,col_ini, row_num, col_num)`. La primera fila del rango es el encabezamiento de los datos. La primera columna contiene las fechas de las series, las cuales son especificadas en el fechado dado en el argumento `dating`, si `dating` es `W` entonces el nombre de la fecha es leído desde la celda encabezamiento de la primera columna. Por omisión se asume el fechado `C`. Las columnas a partir de la segunda contienen los datos de las series a las cuales se les asigna el contenido de la correspondiente celda encabezamiento como nombre. Las fechas son interpretadas según el formato especificado en el argumento `dateFormat`, si el valor de este argumento es “”, entonces se interpretan según el formato por omisión de TOL haciendo uso de la función `TextToDate`.
- `Set GetFullSeriesByCol(TimeSet dating, Text dateFormat)`: similar a la función `ReadSeriesByCol` tomando como rango el rango mínimo que contiene datos en la hoja de trabajo activa.
- Además de los métodos de lectura anteriores tenemos los siguientes métodos para la escritura en rangos de celdas de la hoja de trabajo activa:
- `Anything WriteCell(Real row, Real col, Anything value)`: escribe un valor en la celda con coordenadas `(row,col)`.
- `Real WriteRange(Real row, Real col, Set cellValues)` : escribe un conjunto de conjuntos a partir la celda con coordenadas `(row,col)`. El rango resultante tendrá tantas filas como subconjuntos tenga `cellValues` y columnas como la longitud del subconjunto más grande.
- `Real WriteMatrix(Real row, Real col, Matrix values)`: escribe la matriz `values` a partir de la celda con coordenadas `(row,col)`. El rango resultante tendrá la misma dimensión en filas columnas que la matriz de entrada.
- `Real WriteVMatrix(Real row, Real col, VMatrix values)`: similar a `WriteMatrix`.

- `Real WriteSeries(Real row, Real col, Set series, Text dateFormat)`: escribe un conjunto de series a partir de la celda especificada por las coordenadas `(row,col)`. La columna de fechas se escribe según el formato especificado en el argumento `dateFormat`.

Ejemplo:

Por último ilustramos el uso de `TolExcel` con un ejemplo. Leeremos la serie temporal de pasajeros aéreos internacionales inicialmente referenciada en el libro *«Time Series: Forecast and Control by Box, Jenkins and Reinsel»* (ISBN: 978-0470272848). El archivo de datos *Excel* podemos descargarlo desde el *url*: http://packages.tol-project.org/data/AirPassangers.xls. Este archivo contiene el total mensual de pasajeros internacionales para el período comprendido entre enero de 1949 hasta diciembre de 1960.

A continuación mostramos el código *TOL* que podemos utilizar para cargar los datos, junto a una imagen de la gráfica de los datos cargados (véase la figura 4.3.1).

```
#Require TolExcel;
// abro el excel
TolExcel::@WorkBook xls =
    TolExcel::@WorkBook::Open("AirPassangers.xls");
Real xls::ActivateWS(1); // activo la hoja 1
// leo las series de datos contenidas en la hoja activa
Set series = xls::GetFullSeriesByCol( TimeSet W, Text "" );
Serie AirPassangers = series[1];
// cierro la conexión con el archivo Excel.
Real xls::Close(?);
```

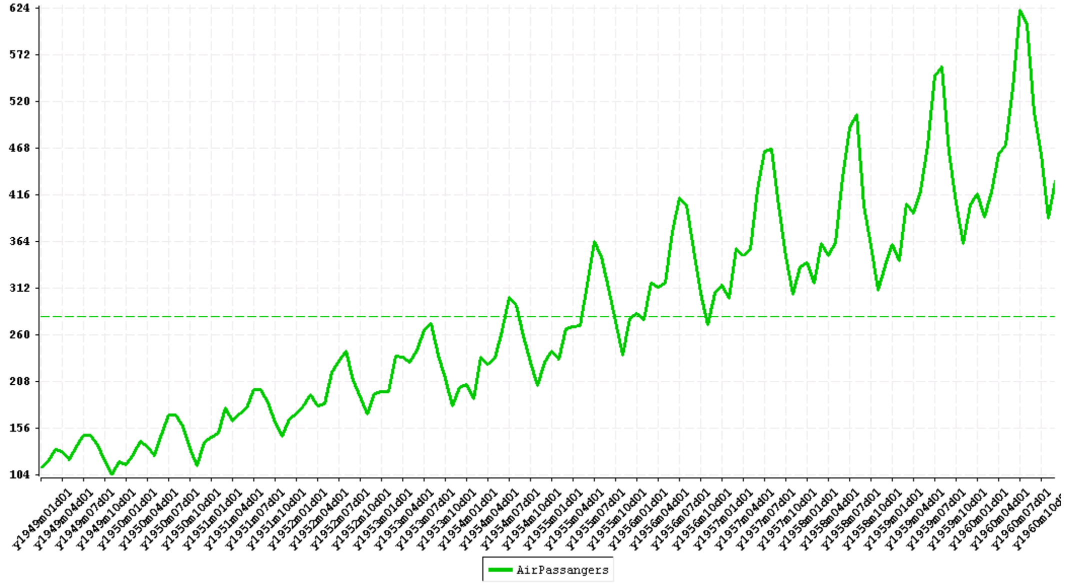

Figura 4.3.1: Gráfica generada en *TOLBase* con los datos del total de pasajeros del ejemplo.

4.4 Otros paquetes

El repositorio oficial de paquetes de *TOL* (*OTAN*) ofrece otros paquetes relacionados con temas como optimización, *gis*, modelación, álgebra lineal, etc. En esta sección destacaremos, sin entrar en detalles, algunos de los que consideramos más interesantes.

4.4.1 Paquetes de *MMS*

Como se menciona en la sección 3.4.6, algunos de los paquetes públicos disponibles en el repositorio *OTAN* han sido desarrollados en el marco de un *trac* diferente orientado a la creación de un sistema para la especificación de modelos estadísticos conocido como *MMS* (*Model Management System*).

El propio paquete `MMS`, así como otros paquetes como `RandVar` o `DecoTools`, son gestionados desde este otro *trac*. Para más información sobre estos paquetes consúltese la web del proyecto: https://mms.tol-project.org o el correspondiente manual de *MMS*.

4.4.2 `TolGlpk`

`TolGlpk` implementa un interfaz nativa con el paquete *GLPK* (*GNU Linear Programming Kit*, http://www.gnu.org/software/glpk). Aunque similar en funcionalidad al módulo `Rglpk`, contenido en *StdLib*, es más eficiente ya que se establece un vínculo directo con la funcionalidad implementada en *C* sin necesidad de pasar por *R*. Por eso se recomienda su uso en lugar de *Rglpk* cuando se necesite resolver un problema de optimización lineal. Además implementa otras opciones no implementadas en *Rglpk*. El código fuente del paquete puede explorarse en: OTAN/TolGlpk.

4.4.3 `NonLinGloOpt`

`NonLinGloOpt` implementa un interfaz nativa con el paquete *NLopt* (http://ab-initio.mit.edu/nlopt) el cual ofrece funcionalidades para la optimización de funciones no lineales con restricciones de igualdad y desigualdad no lineales. Este paquete cuenta con una página de documentación en: https://www.tol-project.org/wiki/OfficialTolArchiveNetworkNonLinGloOpt.

4.4.4 `TolIpopt`

`TolIpopt` es otro paquete de optimización no lineal basado en el método de punto interior. Descansa en el paquete, implementado en *C++*, *IPOPT* (*Interior Point OPTimizer*, https://projects.coin-or.org/Ipopt). El código fuente del paquete y ejemplos pueden explorarse desde el enlace: OTAN/TolIpopt.

4.4.5 `MatQuery`

`MatQuery` implementa funciones para la consulta, selección y clasificación sobre matrices. A veces tendemos a implementar operaciones de consultas sobre matrices ayudándonos de recorridos explícitos sobre las celdas de la matriz mediante el uso de operadores procedurales como el `For` o el `EvalSet`. Este paquete es un buen ejemplo sobre el uso eficiente de operadores matriciales. El código fuente puede explorarse en: OTAN/MatQuery.

4.4.6 `BysMcmc`

`BysMcmc` permite la generación de *cadenas de markov* (*MCMC*) de los parámetros de modelos lineales jerárquicos con errores normales. Entre las características de los modelos destacamos la posibilidad de especificar estructura *ARIMA* de los errores, información a priori de los parámetros lineales, efectos no lineales, restricciones lineales, etc.

Para ampliar sobre la teoría aplicada en este paquete y sobre el uso del paquete recomendamos la lectura del documento: OTAN/BysMcmc/bsr/doc/BSR_Bayesian_Sparse_Regression.pdf.

La forma más cómoda de usar este paquete es mediante el paquete *MMS*. El código fuente del paquete puede explorarse desde OTAN/BysMcmc.

Índices

Índice de figuras

Figura 1.1.1: Sitio web de *TOL-Project*. 5
Figura 1.2.1: Formulario de creación de un nuevo tique. 5
Figura 1.3.1: Explorador web del código fuente de *TOL* en el *trac* de *TOL-Project*. 8
Figura 1.3.2: Explorador de revisiones del código de *TOL* en el *trac* de *TOL-Project*. 8
Figura 1.4.1: Tabla de versiones disponible para descargar en *Windows*. 9
Figura 1.5.1: Ventana principal de *TOLBase*: inspector de objetos. 12
Figura 1.5.2: *TOLBase* mostrando algunas de sus funcionalidades: consola de evaluación de código *TOL*, gráficos de series temporales y edición de archivos. 13
Figura 3.4.1: Interfaz del gestor de paquetes de *TOLBase*. 72
Figura 4.2.1: Menú contextual desplegado sobre un objeto de tipo `Serie`. 78
Figura 4.2.2: Ejemplo de la ventana de edición de un objeto contenedor: `Set` o `NameBlock`. 81
Figura 4.3.1: Gráfica generada en *TOLBase* con los datos del total de pasajeros del ejemplo. 85

Índice de tablas

Tabla 2.1.1: Clasificación de los caracteres ASCII en *TOL*. 17
Tabla 2.1.2: Clasificación de los caracteres extra de LATIN1 (ISO 8859-1) en *TOL*. 17
Tabla 2.1.3: Caracteres escapados en la cadenas de texto en *TOL*. 20
Tabla 2.2.1: Proceder interno de *TOL* en la creación de conjuntos. 24
Tabla 2.4.1: Ejemplo en forma de tabla del resultado de aplicar un retardo y una diferencia a una serie temporal. 49

Manual de *TOL*
Edición 1. Fecha: 04/02/2013

Autores: Pedro Gea y Jorge S. Pérez.
Colaboradores: Miguel Á. Fernández y Claudia Escalonilla.
Editor: Pedro Gea.

Correspondencia: pgea@bayesinf.com.

www.bayesforecast.com
www.tol-project.org