Open Modular Interpreter

Manual de usuario

Fco. Javier Bohórquez Ogalla

Índice

1. Introducción

OMI, acrónimo de Open Modular Interpreter, es un lenguaje para la creación de 'scripts' que presenta un propósito general y es de código abierto. El objetivo del proyecto OMI es servir de guía en la aplicación práctica de la teoría de compiladores e intérpretes. Para ello presenta un caso práctico en el que se construye un lenguaje de programación moderno a partir de los conceptos teóricos de autómatas y lenguajes formales.

La plataforma OMI representa un sistema software desarrollado para la comunidad académica. Pretende ser una herramienta utilizada en el aprendizaje de los sistemas intérpretes y traductores modernos, además de cualquier otro basado en los conceptos sobre los que estos se construyen.

La plataforma OMI se constituye mediante los siguientes elementos:

- Documentación completa del proceso aplicado en el desarrollo de un lenguaje de programación.
- Intérprete de un lenguaje de programación denominado OMI, el cual da nombre al proyecto.
- Aplicación web que permite consultar la documentación e interactuar con el intérprete.

Este manual describe al lenguaje, las funciones que integra, y las características del intérprete. Además detalla cómo se estructura el contenido dentro de la aplicación web y las herramientas que la componen.

2. Características

OMI representa una plataforma constituida por una serie de herramientas. Estas ayudan al aprendizaje en la aplicación práctica de la teoría de autómatas y lenguajes formales para el desarrollo de un lenguaje de programación moderno. Sus características son las siguientes:

- Intérprete OMI
  - Abierto
  - Modular
  - Interactivo

    - Configurable
    - Paso de argumentos
- Lenguaje OMI
    - Propósito general
    - Interpretado
    - Sintaxis simple y cercana a los lenguajes modernos
    - Tipado dinámico y débil
    - Tipos de datos simples y compuestos
    - Referencia de datos
    - Funciones y operadores sobre los distintos tipos de datos
    - Sentencias de control
    - Variable de ámbito global y local
    - Definición de funciones
    - Paso de parámetros por valor y por referencia
    - Funciones de orden superior
    - Clausura de funciones
    - Funciones anónimas
    - Aplicación parcial de funciones
    - Decoradores
    - Definición de clases de objetos
    - Creación e instanciación de objetos
    - Tipos de datos como clases de objetos
    - Visibilidad de métodos y atributos
    - Definición estática de métodos y atributos
    - Polimorfismo
    - Duck typing
    - Herencia simple
    - Métodos mágicos
    - Dynamic binding
    - Excepciones
    - Evaluación por cortocircuito devolviendo último valor
    - Operadores condicionales
    - Funciones de fechas y tiempo

    - Funciones de creación y acceso a ficheros
    - Concurrente
    - Recolector de basura
- Web OMI project.

3. Requisitos previos

El intérprete OMI puede ser ejecutado sobre cualquier hardware actual. El software por si mismo solo necesita unos pocos kilobytes de memoria para funcionar, sin embargo el hardware necesario dependerá en gran medida del código fuente que será interpretado.

Para usar el intérprete OMI es necesario disponer de un sistema GNU/Linux. El intreprete depende de una serie de bibliotecas de programación. Dependiendo de la instalación que se realice será necesario la instalación previa de estas o no.

- readline
- boost-regex

Por otro lado para el correcto uso de la aplicación web se precisa de un navegador web con soporte HTML5 y JavaScript.

4. Uso del sistema

4.1. Obtener OMI

El intérprete OMI puede ser descargado desde la web OMI project. Se ofrecen varias alternativas para la descarga:

- Código fuente.
- Precompilado y empaquetado para algunas distribuciones GNU/Linux.

Se recomienda la descarga de la última versión estable del sistema software.

4.2. Instalación

El intérprete OMI puede ser instalado sobre un sistema GNU/Linux mediante varios procesos.

4.2.1. Compilación e instalación desde código fuente

Para instalar OMI mediante el código fuente de la aplicación previamente se deberá llevar a cabo la compilación de este. Es posible obtener el código fuente de la aplicación desde la sección de descargas de la web OMI project.

El proceso de compilación se vale de una serie de herramientas que lo simplifican y parametrizan, permitiendo la configuración del ejecutable obtenido.

- autoconf
- automake
- make
- g++

Además se mantiene dependencia con las siguientes bibliotecas de programación:

- Biblioteca estándar de C/C++
- Biblioteca GNU readline
- Biblioteca Boost regex

Instalación de dependencias en una distribución Debian:

```
1 $PS1 apt-get install autoconf automake build-essential \
2 $PS2  libreadline-dev libboost-regex-dev
```

Para la compilación es necesario abrir una sesión de terminal de comandos en el directorio sobre el que se desea llevar a cabo el proceso. Se supone que el usuario tiene permisos suficientes en el sistema para llevar a cabo la instalación.

Descarga del código fuente de la versión $VERSION:

```
1 $PS1 wget http://wwww.omi-project.com/download/code/omi_$VERSION.tar.gz
```

Descompresión y desempaquetado del fichero:

```
$PS1 tar -xzvf omi_$VERSION.tar.gz
$PS1 cd omi_$VERSION
```

Es posible establecer determinados aspectos del intérprete durante el proceso de compilación. Las opciones más significativas son las siguientes:

numType: Tipo de dato utilizado para la representación interna de los datos numéricos. El valor por defecto es "double". Su valor puede ser cualquier tipo de dato estándar de C/C++ o algún tipo específico facilitado por el usuario. También admite el valor en número de bytes, siendo los posibles valores "4bytes", "8bytes" y "16bytes".

numPrecision: Precisión numérica utilizada en al escribir en la salida estándar. Su valor por defecto es 15, y puede ser cualquier número entero.

refCType: Tipo de dato utilizado para contar el número de referencias de los nodos. Determina el número de referencias máximas que puede llegar a tener un dato. Su valor por defecto "unsigned int". Puede ser cualquier tipo de dato estándar de C/C++ o algún tipo específico facilitado por el usuario.

sizeNode: Esta opción permite configurar el intérprete dando valores óptimos a las opciones anteriores. Puede presentar uno de los siguientes valores:

small: numType="float" refCType="unsigned int" numPrecision=6

normal: numType="double" refCType="unsigned int" numPrecision=15

big: numType="long double" refCType="unsigned int" numPrecision=18

Para ver un listado completo de las opciones de configuración para la compilación:

```
$PS1 ./configure --help
```

Configurar el proceso de compilación con las opciones por defecto:

```
$PS1 ./configure
```

Durante la ejecución de este script se comprobará si se satisfacen las dependencias, y se creará los scripts necesarios para la compilación.

Para llevar a cabo la compilación:

```
$PS1 make
```

Una vez compilado con éxito es posible proceder con la instalación:

```
$PS1 make install
```

Para comprobar la integridad de la instalación:

```
$PS1 make check
```

Para limpiar el entorno de compilación:

```
1 $PS1 make clean
```

Para desinstalar el software:

```
1 $PS1 make unistall
```

4.2.2. Instalación mediante paquete .deb

Cada versión del intérprete es precompilada y empaquetada. Las bibliotecas externas se compilan de forma estática por lo que se elimina la dependencia con otros paquetes.

Con una sesión de terminal y los permisos en el sistema suficientes es posible descargar e instalar el paquete .deb

Para descargar el paquete:

```
1 $PS1 wget http://wwww.omi-project.com/download/deb/omi_$VERSION.deb
```

Para instalar el paquete:

```
1 $PS1 dpkg -i omi_$VERSION.deb
```

Para desinstalar el paquete:

```
1 $PS1 dpkg -r omi
```

4.3. Intérprete

labelsec:intérpreter El intérprete es el sistema software sobre el que gira el proyecto OMI. Se trata del software encargado de leer y ejecutar el código fuente escrito en OMI.

Para ejecutar el intérprete se utiliza el comando con el mismo nombre.

```
1 $PS1 omi
```

Ejecutar el comando sin argumento hará que el intérprete lea y procese la entrada estándar.

La sintaxis del comando es la siguiente:

$$omi\ [options]\ [file]\ [args...]$$

El fichero será un documento escrito en el lenguaje OMI. La lista de argumentos serán datos accesibles desde el código fuente contenido en el fichero.

Las posibles opciones son:

-c cmd: Interpreta la cadena cmd.

-i: Modo interactivo. Las expresiones y sentencias son interpretadas en tiempo real y solicitadas mediante un prompt.

-h: Muestra la ayuda del comando.

-V: Muestra la versión del software.

Los argumentos facilitados serán accesible desde el código fuente mediante el array "args", donde la primera posición del array es el nombre del fichero y las siguientes los argumentos facilitados.

Se pone a disposición del usuario una hoja de manual que detalla el uso del comando.

```
$PS1 man omi
```

Con la instalación por defecto el intérprete es ubicado en la siguiente ruta:

/usr/local/bin/omi

Además los recursos, como por ejemplo las extensiones, serán ubicados en el siguiente directorio:

/usr/local/share/omi

4.4. Referencias del lenguaje

4.4.1. Sentencias

Como muchos otros lenguajes de programación toda sentencia OMI debe acabar en punto y coma ";".

Es posible delimitar un bloque de sentencias entre llaves. Muchos recursos del lenguaje operan o trabajan sobre un bloque de sentencias. Estas no necesitan acabar en ";".

Los cierre de bloque o fin de la entrada automáticamente implica un punto y coma. En el modo interactivo no es necesario si la línea introducida sólo se compone de una sentencia.

```
1  << "Hola mundo"; // Sentencia acabada
2  if (true) { << "Y un ejemplo"; } // Bloque de sentencias
3  << "Soy un programa OMI"
```

4.4.2. Comentarios

Los comentarios son fragmentos de texto que son incluidos en el código fuente y que serán omitidos por el intérprete. Se utilizan normalmente para aclarar y/o documentar las piezas de código.

En OMI es posible realizar un comentario de línea al estilo de C++ o de consola de comando UNIX.

```
1   x = 1; //x vale 1
2   y = 2; #y vale 2
```

Por otro lado, es posible escribir un bloque de una o varias líneas de comentarios delimitándolo mediante "/*" y "*/".

```
1   /*
2       x vale 1
3       y vale 2
4   */
5   x = 1;
6   y = 2;
```

4.4.3. Ejecución

La ejecución de un programa escrito en OMI comenzará desde un fichero, y desde la primera sentencia contenida en este. Las sentencias en este fichero que no esten contenidas en ninguna función o método son consideradas el flujo principal del programa.

La ejecución del programa normalmente se realiza sentencia a sentencia de forma secuencial. Algunas sentencias y expresiones del lenguaje podrán cambiar este flujo.

La ejecución finaliza cuando se llega al final de este fichero o se resuelve una sentencia que produzca la salida. También es posible que se de algún error sintáctico o semántico

4.4.4. Identificadores

Los identificadores en OMI sirven para nombrar elementos del script definidos por el programador, tales como variables, funciones, etiquetas, clases...

Un identificador debe comenzar por una letra (mayúscula o minúscula) o un subguión (_), seguidos de tantas letras, números o subguiones como se desee.

```
1   foo = 1;  // foo identifica una variable que vale uno
```

```
2   ~ bar () { // bar identifica una función que imprime "Hola mundo".
3       << "Hola mundo";
4   }
```

OMI es sensible a mayúsculas, por lo que considera dos identificadores distintos si no mantienen la misma capitalización.

Un identificador no puede coincidir con una palabra reservada del lenguaje o se producirá un error sintáctico.

4.4.4.1 Palabras reservadas

- null
- NULL
- true
- false
- global
- getenv
- function
- P
- return
- class
- new
- private
- extends
- this
- extends
- parent
- static
- include
- if
- else
- elseif
- elif
- switch
- case
- default
- with
- do
- while
- for
- as
- in
- break
- continue
- exit
- goto
- throw
- try
- catch
- print
- echo
- input
- inputline
- typeof
- typeOf
- datinfo
- datInfo
- empty
- isnull
- isbool
- isnum
- isstring
- isarray
- isobject
- AND
- and
- OR
- or
- size
- str_explode
- str_implode

- array_implode
- str_implode
- str_find
- str_pos
- str_replace
- str_replace_sub
- str_upper
- str_lower
- str_search
- str_match
- sprintf
- regexp
- reduce
- array_chunk
- array_search
- int
- float
- bool
- string
- date
- time
- sleep
- file
- fopen
- fput
- fget
- fread
- fwrite
- fappend
- fapp
- fclose
- ftell
- fseek
- fSET
- fCUR
- fEND
- exec
- eval
- fork
- wait
- getpid
- getppid
- process
- signal
- kill
- shandler
- exit_process
- load

En OMI se nombran de igual forma las variables, funciones, clases... Es el contexto en el que se da un identificador el que determina el recurso que este nombra. Por regla general todos los operadores, funciones y demás construcciones del lenguaje toman, a no ser que se indique lo contrario, los identificadores como variables. Las excepciones a esta regla son:

- Operador llamada a función
- Operador aplicación parcial
- Operador *new*

OMI dispone de mecanismos para indicar explícitamente el tipo de recurso que es nombrado.

4.4.5. Tipos de datos

Haciendo uso del lenguaje OMI se puede operar sobre varios tipos de datos simples y compuestos.

OMI es un lenguaje de programación que presenta un tipado dinámico, es decir, las variables se consideran del mismo tipo que al dato al que referencian. Así no es necesario declarar el tipo de dato que contendrán.

Los tipos de datos sobre los que opera OMI son:

- Simples:
  - Booleanos.
  - Numéricos.
  - Cadenas de caracteres.
- Compuestos:
  - Arrays.
  - Objetos.
- Especiales:
  - Nulo.
  - Función.
  - Expresión regular.

Algunas funciones y operadores del lenguaje tratan tipos de datos concretos. A pesar de ello OMI es un lenguaje débilmente tipado, por lo que es posible usar valores de un tipo de dato en expresiones que requieran otro tipo de dato. Para ello se lleva a cabo una conversión de tipo de forma implícita.

4.4.5.1 Booleano

Este tipo de dato puede contener dos únicos valores: verdadero o falso. Es el tipo de dato más simple. El valor verdadero se expresa mediante el literal "true", mientras que el falso se representa mediante "false".

```
1   a = true; // a presenta el valor verdadero
2   b = false; // b presenta el valor falso
```

Para más información:

- Operadores lógicos (Párrafo 4.4.6.1).
- Operadores de comparación (Párrafo 4.4.6.2).
- Conversión implícita entre tipos de datos (Párrafo 4.4.5.9)

4.4.5.2 Numérico

En OMI únicamente existe un tipo de dato numérico. Mediante este tipo de dato es posible representar números enteros y aquellos decimales que puedan ser expresados en punto flotante.

A pesar de que solo presenta un tipo de dato numérico ofrece mecanismos para operar con enteros y decimales, por ejemplo es posible obtener la parte entera de un número decimal.

Los números son expresados en OMI mediante cualquier conbinación de dígitos, separando la parte decimal mediante un punto si existiera. Además es posible preceder la expresión de un signo "+" o "-".

```
1    a = 1; // a presenta el valor 1
2    b = -1; // b presenta el valor -1
3    c = +1; // c presenta el valor +1
4    d = 0.001; // d presenta el valor 0.001
5    e = -0.001; // e presenta el valor -0.001
```

En OMI es posible configurar cómo los tipos de datos numéricos son almacenados y tratados de forma interna. Así es posible alterar el valor numérico máximo que se puede representar y la precisión. No obstante, esto sólo es posible en tiempo de compilación, por lo que se deberá usar la misma configuración para todos las ejecuciones.

Para más información:

- Operadores aritméticos (Párrafo 4.4.6.3).
- Compilación e instalación desde código fuente (Subsubsección 4.2.1).
- Conversión implícita entre tipos de datos (Párrafo 4.4.5.9)

4.4.5.3 Cadena de caracteres

En general, un carácter se refiere a una letra, número u otro símbolo o signo. Las cadenas de caracteres son una sucesión de estos. Normalmente representan un contenido textual.

En OMI las cadenas de caracteres son consideradas como un tipo de dato simple. Esto es debido a que no existe el tipo de dato carácter, y por tanto los datos de este tipo no puede ser descompuesto en datos de tipos más simples. De esta forma se considera al carácter simple como una cadena de un solo elemento.

Una cadena de caracteres queda delimitada entre comillas simples o comillas dobles.

```
1   a = "Una cadena de caracteres"; // a es una cadena definida entre "
2   b = 'Y otra'; // b es una cadena definida entre '
```

> Para más información:
>
> - Operadores sobre cadenas de caracteres(Párrafo 4.4.6.4).
> - Funciones sobre cadenas de caracteres (??).
> - Conversión implícita entre tipos de datos (Párrafo 4.4.5.9)

4.4.5.4 Array

En OMI un array es una estructura de datos que almacena valores de forma contigua. Los datos que guarda un array pueden ser de diferente tipo.

Se pueden dar dos tipos de arrays:

- Secuenciales.
- Asociativos.

Los arrays secuenciales son accesibles mediante índices numéricos y de una forma posicional. Se definen mediante una secuencia de expresiones que representan un valor. Estas serán separadas por el carácter “,” y delimitadas entre llaves “” y “”. El primer elemento del array se corresponde con el índice numérico 0.

```
1   a = {0,1,2}; // a es un array que contiene los números 0, 1 y 2
2   /*
3      b es un array que contiene el numero 0, la
4      cadena "str" y otro array con los numeros 0 y 1
5   */
6   b = {0,"str", { 0, 1 } };
```

En arrays asociativos cada elemento está referenciado mediante una clave, de forma que se puede utilizar esta para acceder al elemento dentro del array. En OMI las claves de los arrays asociativos pueden ser cualquier expresión válida que represente un valor de tipo simple. Un array asociativo se define igual que uno secuencial salvo que los elementos que lo componen son pares de expresiones correspondientes a la clave y al valor y separadas

por el carácter ":".

```
/*
   a es un array asociativo con las cadenas
   'k0', 'k1' y 'k2' como claves. Sus valores
   son 0, 1 y 2 respectivamente.
*/
a = {'k0' : 0, 'k1' : 1, 'k2' : 2};
/*
   b es un array asociativo con los números 0 y
   10, y la cadena "K" como claves. Sus valores son
   el numero 0, la cadena "str" y otro array con
   los numeros 0 y 1.
*/
b = { 10: 0, 0: "str", "K": { 0, 1 } };
```

> Para más información:
>
> - Operadores sobre arrays (Párrafo 4.4.6.4).
> - Funciones sobre arrays (??).

4.4.5.5 Objetos

En programación orientada a objetos, un objeto es una estructura de datos que presenta un comportamiento asociado. Tiene un estado interno determinado por el valor de los datos que contiene, y se encarga de realizar una serie de tareas en tiempo de ejecución. Una clase de objetos es un modelo que define las propiedades y el comportamiento que tienen objetos afines.

El uso de las clases y los objetos es descrito en profundidad como una referencia del lenguaje (Subsubsección 4.4.13).

En OMI todo dato es considerado de tipo objeto, los otros tipos se ven como clases de objetos. Un dato, además de tener asociado un valor de un tipo concreto, presenta una serie de acciones u operaciones que se pueden realizar con él y que están determinadas por su tipo. Por ejemplo sobre un dato array se puede realizar la operación implode para obtener una cadena con todos los elementos del array separado por una subcadena separadora.

```
/*
   A la variable "a" se le asigna
   el resultado de ejecutar la operación
   implode sobre el array.
*/
a = {"hola","mundo"}->implode(" ");
```

OMI permite al usuario definir, construir y utilizar sus propios objetos. Para ello se vale de las clases y de mecanismos para instanciar los objetos a partir de estas.

Para más información:

- Clases de objetos (Subsubsección 4.4.13).

4.4.5.6 Nulo

Nulo representa la ausencia de valor. Es un tipo de dato especial que se utiliza para expresar referencias que no tienen valor asociado. Paradójicamente el valor nulo conlleva la ausencia de valor.

En OMI cualquier referencia sin valor se considera como nulo. Las variables no inicializadas, o las posiciones de arrays inexistentes tienen el valor nulo.

Es posible expresar un dato nulo mediante el literal "null" (o en mayúsculas "NULL").

```
1   a = null; // a no contiene valor
```

4.4.5.7 Función

En OMI una función hace referencia a un bloque de sentencias que puede recibir una serie de parámetros y tomar un valor. Una función realiza una tarea específica dentro del programa. Cuando una función es llamada se ejecuta el bloque de sentencias al que referencia, pasándole como parámetros los valores en la llamada.

El uso de las funciones es descrito en detalle más adelante, en este manual, como una referencia del lenguaje (Subsubsección 4.4.13).

En OMI las funciones son consideradas tipos de datos, siendo posible operar sobre ellas. Así una función puede ser asignada a una variable, pasada como parámetro o ser devuelta por otra función.

```
1   /*
2      a es una función que recibe un
3      parámetro y lo devuelve como valor
4   */
5   a = ~(param){ return param; };
```

Para más información:

- Funciones (Subsubsección 4.4.11).

4.4.5.8 Expresión regular

Una expresión regular es un patrón definido por un lenguaje regular. Normalmente se aplica a una cadena de caracteres, para determinar si pertenece al lenguaje que define. Se utiliza para realizar búsquedas de subcadenas o llevar a cabo remplazos.

Con OMI se puede expresar cualquier expresión regular usando la sintaxis PERL. Para ello se ha de delimitar entre acento grave (`).

```
1  /*
2     "exp" es la expresión regular correspondiente
3     al lenguaje descrito por cero o mas caracteres "a"
4     seguidos de dos caracteres "b"
5  */
6  exp = `a*bb`;
```

El lenguaje dispone de funciones para crear y operar sobre expresiones regulares.

> Para más información:
>
> - Funciones de expresiones regulares (**??**).

4.4.5.9 Conversión implícita entre tipos de datos

OMI es un lenguaje de tipado débil, es decir, dado un valor de un tipo concreto es posible utilizarlo como otro tipo sin necesidad de llevar a cabo una conversión de forma explícita. Al usar un dato en una operación que espera otro tipo OMI realizará una conversión automática.

Nulo

Booleano: false

Numérico: 0

Cadena de caracteres: ""

Booleanos

false

Numérico: 0

Cadena de caracteres: ""

true

Numérico: 1

Cadena de caracteres: "1"

Numéricos

Número 0

Booleano: false

Cadena de caracteres: ″0″

Número distinto a 0

Booleano: true

Cadena de caracteres: Cadena que representa el número.

Cadena de caracteres

Cadena vacía

Booleano: false

Numérico: 0

Cadena numérica

Booleano: true

Numérico: Número que representa la cadena

Otra cadena

Booleano: true

Numérico: Tamaño de la cadena

Normalmente no es posible usar un dato simple en lugar de uno compuesto. Las funciones y operaciones que esperan recibir como argumento un dato compuesto devolverán un error si no es así. No obstante un dato compuesto puede utilizarse como uno simple sin que se produzcan errores semánticos.

Array

Array vacío

Booleano: false

Numérico: 0

Cadena de caracteres: ″″

Array con elementos

Booleano: true

Numérico: 0

Cadena de caracteres: ″″

En el caso de los objetos, es posible definir métodos que serán invocados cuando sean utilizados como otro tipo de dato. Estos métodos son denominados métodos mágicos (**??**).

4.4.6. Operadores

OMI soporta una serie de operadores de naturaleza muy variada. Mediante un operador se puede construir una expresión que, al ser evaluadas, se aplicará una función sobre un conjunto de datos de entrada denominados operandos, y obtener así un valor.

Un operador normalmente es de naturaleza binaria, presentando dos entradas, aunque existe operadores unitarios o ternarios. Además presenta una sintaxis y semántica propia.

```
1   foo = 1 + 1; // foo es el resultado de evaluar la operación suma
```

Los operadores pueden combinarse para formar expresiones más complejas. Para la evaluación se establece una precedencia de operadores implícita, que determina en qué orden deben de resolverse. Es posible alterar la precedencia de operadores encerrando la expresión entre paréntesis.

```
1   foo = 2 * 8 + 3; // se evalúa la operación producto y luego la suma.
2   bar = 2 * (8 + 3); //se evalúa la operación suma y luego el producto.
```

La mayoría de operadores que se pueden utilizar en OMI tienen una naturaleza matemática y se resuelven igual forma. No obstante también soporta otros operadores propios de los lenguajes de programación.

4.4.6.1 Operadores lógicos

Al aplicarse un operador lógico se produce un resultado booleano derivado de la evaluación de una condición.

Los operadores lógicos son los siguientes:

Negación: !

AND lógico: &&, *and*

OR lógico: ||, *or*

```
1   foo = !true; // foo vale falso
2   foo = true && false; // foo vale falso
3   foo = true or false; // foo vale verdadero
```

Los operadores AND y OR son evaluados de izquierda a derecha en corto circuito. Se puede producir el resultado mediante una evaluación parcial, al conocerse de antemano el valor que se obtendrá mediante la evaluación total. En este caso el resto de la expresión

no será evaluada. Además el resultado siempre será el último operando en ser evaluado.

```
1   foo = null and 3 + 4; // foo vale null, evaluación parcial
2   foo = {} or "10"; // foo vale "10", evaluación total
3   foo = "10" and 3 + 4; // foo vale 7, evaluación total
4   foo = "5" or {}; // foo vale "5", evaluación parcial
```

> Para más información:
>
> - Tipos de dato (Subsubsección 4.4.5).
> - Tipo de dato booleano (Párrafo 4.4.5.1).

4.4.6.2 Operadores de comparación

Se encargan de evaluar el orden entre dos valores. El resultado es un valor booleano de verdadero si se satisface la comparación o falso en caso contrario.

Los operadores de comparación son los siguientes:

Igualdad: ==

Desigualdad: !=

Menor que: <

Menor o igual que: <=

Mayor que: >

Mayor o igual que: >=

Idéntico: ===

No idéntico: !==

```
1   foo = 5 == 5; // foo vale verdadero
2   foo = "5" == 5; // foo vale verdadero
3   foo = 5 != 5; // foo vale falso
4   foo = "3" < 4; // foo vale verdadero
5   foo = "3" <= 4; // foo vale verdadero
6   foo = 4 > 4; // foo vale falso
7   foo = 4 >= 4; // foo vale verdadero
8   foo = 5 === 5; // foo vale verdadero
9   foo = "5" === 5; // foo vale falso
10  foo = "5" !== 5; // foo vale verdadero
```

Para los tipos booleanos se considera que verdadero (true) es mayor que falso (false). Para los datos de tipo numéricos se toma su valor aritmético. Las cadenas de caracteres son ordenadas alfabéticamente.

Cuando se comparan datos de distintos tipos se realiza una conversión al tipo de dato más simple. Por ejemplo al comparar un dato numérico con un booleano se convertirá el numérico a booleano.

```
foo = true <= 5; // foo vale verdadero
foo = "5" == true; // foo vale verdadero
foo = "word" < 4; // foo vale falso
```

Cuando se realiza una comparación de forma idéntica, además de comparar el valor de los datos, se comprueban los tipos.

> Para más información:
>
> - Tipos de dato (Subsubsección 4.4.5).
> - Tipo de dato booleano (Párrafo 4.4.5.1).

4.4.6.3 Operadores aritméticos

Se corresponden con las operaciones aritméticas básicas que son la suma, la diferencia, el producto y la división, y aquellas derivadas de estas. El resultado de la operación será un número.

Los operadores aritméticos toman como operandos expresiones tratadas como numéricos. Todo dato que sea sometido a una operación aritmética será convertido a numérico.

Los operadores aritméticos soportados por OMI son:

Suma: +

Diferencia: −

Producto: ∗

Division: /

Módulo: %

Potencia: ^

```
foo = 1 + 1; // foo vale 2
foo = "5" + 5; // foo vale 10
foo = "word" + 4; // foo vale 8
foo = true + 10; // foo vale 11
foo = 4 - "2"; // foo vale 2
foo = 3 * 4; // foo vale 12
foo = 17 / 4; // foo vale 4.25
foo = 2.1 / "8"; // foo vale 0.2625
foo = 4 % 3; // foo vale 1
foo = 4 ^ 2; // foo vale 16
```

El resultado del operador módulo mantendrá el mismo signo que el operando que hace de dividendo.

> Para más información:
>
> - Tipos de dato (Subsubsección 4.4.5).
> - Tipo de dato numérico (Párrafo 4.4.5.2).

4.4.6.4 Operadores sobre cadenas de caracteres

OMI soporta una serie de operadores que pueden ser aplicados sobre cadenas de caracteres.

Concatenación: .

Acceso a posición: $[n]$

```
1   foo = "ABC" . "DEF"; // foo vale "ABCDEF"
2   foo = "ABCDE"[0]; // foo vale "A"
```

La concatenación se realiza después de que se determinen los números decimales presentes en la expresión. Todo carácter punto no suponga un separador decimal será considerado un operador de concatenación.

```
1   foo = "AB".2.1.1."CD"; // foo vale "AB2.11CD"
2   foo = "AB".2.1.1.1."CD"; // foo vale "AB2.11.1CD"
```

El operador de acceso a posición recibe un índice que indica el elemento al que acceder dentro de la cadena. Si la posición indicada no existe se toma el valor nulo. El índice siempre será considerado como un dato numérico.

El operador de acceso también puede ser utilizado para escribir en una determinada posición de la cadena. No obstante, en este caso, debe existir un elemento en la posición indicada o se producirá un error.

```
1   foo = "ABCD"; // foo vale "ABCD"
2   foo[0] = "E"; // foo vale "EBCD"
3   foo[0] = "FG"; // foo vale "FGBCD"
4   foo[10] = "X"; // se produce un error.
```

No es posible utilizar el operador de acceso para escribir en una determinada posición de una cadena constante.

```
1   "ABCD"[0] = "X"; // se produce un error.
```

El operador de acceso es utilizado para otros tipos de datos como los arrays.

> Para más información:
>
> - Tipos de dato (Subsubsección 4.4.5).
> - Tipo de dato cadena de caracteres (Párrafo 4.4.5.3).
> - Funciones sobre cadena de caracteres (??).
> - Operadores sobre array (Párrafo 4.4.6.8).

4.4.6.5 Operadores de asignación

La asignación es la operación mediante la cual el valor de una expresión puede ser atribuido a un símbolo variable. De esta forma se puede hacer uso del valor en otras expresiones. El operador se representa mediante un signo igual "=" y toma una expresión (operando derecho) para asignarla a un símbolo variable (operando izquierdo).

En OMI se lleva a cabo una asignación destructiva. Cuando una expresión es asignada, por ejemplo a una variable, el valor de esta última cambia, indistintamente de lo que contenía anteriormente.

```
1   foo = 1; // foo vale 1
2   foo = 2; // foo a cambiado al valor 2
3   bar = foo; // bar vale el valor actual de foo que es 2
4   foo = 10; // foo a cambiado al valor 10 y bar sigue valiendo 2
```

OMI, como otros lenguajes, soporta una serie de operadores que combinan una operación determinada y una asignación. Estos simplifican muchas expresiones.

Suma y asignación: $+=$

Diferencia y asignación: $-=$

Producto y asignación: $*=$

División y asignación: $/=$

Potencia y asignación: $\hat{}=$

Módulo y asignación: $\%=$

Concatenación y asignación: $.=$

```
1   foo = 1; // foo vale 1
2   foo += 2; // foo a cambiado al valor 3
3   foo -= 4; // foo a cambiado al valor -1
4   foo *= -9; // foo a cambiado al valor 9
5   foo /= 3; // foo a cambiado a 3
6   foo ^= 3; // foo a cambiado a 27
7   foo %= 2; // foo a cambiado a 1
8   foo .= "F"; // foo a cambiado a "1F"
```

Para más información:

- Operadores aritméticos (Párrafo 4.4.6.3).
- Operadores sobre cadenas de caracteres (Párrafo 4.4.6.4).

4.4.6.6 Operadores de incremento y decremento

Estos operadores se aplican a una expresión variable, incrementando o decrementado su valor numérico. Este tipo de operaciones implican una asignación.

Se distinguen entre si, además de que si el valor es incrementado o decrementado, en el momento en el momento en el que se tomará el valor. Pudiendo aplicarse la operación antes o después de evaluar el resto de la expresión.

Preincremento: $++i$

Posincremento: $i++$

Predecremento: $--i$

Posdecremento: $i--$

```
1   foo = 1; // foo vale 1
2   bar = ++foo; // bar vale 2 y foo vale 2
3   bar = foo++; // bar vale 2 y foo vale 3
4   bar = --foo; // bar vale 2 y foo vale 2
5   bar = foo--; // bar vale 2 y foo vale 1
```

Para más información:

- Operadores aritméticos (Párrafo 4.4.6.3).
- Operadores de asignación (Párrafo 4.4.6.5).

4.4.6.7 Operadores de conversión de tipo

En OMI se definen una serie de operadores que permiten realizar una conversión explícita a un determinado tipo de dato.

Booleano: *bool*

Entero: *int*

Punto flotante: *float*

Cadena de caracteres: *string*

```
foo = bool 4; // foo vale el booleano true
foo = bool 0; // foo vale el booleano false
foo = int "4.5"; // foo vale el numérico 4
foo = int "AA"; // foo vale el numérico 2
foo = float "4.5"; // foo vale el numérico 4.5
foo = string 55; // foo vale la cadena "55"
foo = string true; // foo vale la cadena "1"
```

A pesar de que OMI soporta la conversión a enteros o flotantes, internamente estos se representan de la misma forma. Por lo que se consideran del mismo tipo.

> Para más información:
>
> - Tipos de datos (Subsubsección 4.4.5).
> - Conversión implícita entre tipos de datos (Párrafo 4.4.5.9).

4.4.6.8 Operadores sobre arrays

El lenguaje OMI soporta una serie de operadores que pueden ser aplicados sobre un dato array.

Acceso a posición: $[k]$

Acceso a final: $[]$

Tamaño: *size*

```
foo = {"A","B"}; // foo es un array que contiene las cadenas "A" y "B"
bar = {"A","B"}[0]; // bar vale "A"
bar = foo[1]; // bar vale "B"
bar = size foo; // bar vale 2
foo[] = "C"; // foo es un array que contiene las cadenas "A", "B" y "C"
bar = size foo; // bar vale 3
foo = { "k0": "v0", "k1": "v1" }; // foo es un array asociativo con dos claves
bar = { "k0": "v0", "k1": "v1" }["k0"]; // bar vale "v0"
bar = foo["k1"]; // bar vale "v1"
bar = size foo; // bar vale 2
foo["k2"] = "v2"; // foo es un array asociativo con tres claves
foo[] = "v3"; // foo es un array asociativo con cuatro claves
```

Con el operador de acceso a posición es posible acceder a un determinado elemento dentro del array. Para ello se debe facilitar el índice o la clave a la que se desea acceder.

El operador de acceso a posición en array no realiza ninguna conversión de tipos sobre el índice o clave. Las claves son comparadas con el operador de igualdad (Párrafo 4.4.6.2).

Si se accede mediante un índice o clave que no existe en el array se toma el valor nulo.

El operador de acceso a última posición por si solo toma el valor nulo, pero puede ser utilizado para añadir elementos al final del array.

No es posible asignar a posiciones de un array constante.

```
1   {1,2}[0] = 10; // se produce un error
2   {"key":"val"}["key"] = "other"; // se produce un error
3   {1,2}[] = 10; // se produce un error
```

El operador de acceso es utilizado en otros tipos de datos como las cadenas de caracteres.

> Para más información:
>
> - Tipos de dato (Subsubsección 4.4.5).
> - Tipo de dato array (Párrafo 4.4.5.4).
> - Funciones sobre array (**??**).
> - Operadores sobre cadenas de caracteres (Párrafo 4.4.6.4).

4.4.6.9 Operadores sobre clases y objetos

OMI es un lenguaje de programación orientado a objetos, por lo que pone a disposición del usuario un conjunto de operadores relacionados con estos.

Acceso a atributos y métodos: ->

Instanciación de clases: *new*

Acceso al objeto en ejecución: *this*

Acceso al objeto padre en ejecución: *parent*

Acceso a la clase en ejecución: *static*

El uso de las clases y objetos quedan descritos en la sección correspondiente (Subsubsección 4.4.13).

> Para más información:
>
> - Clases de objetos (Subsubsección 4.4.13).
> - Tipos de datos (Subsubsección 4.4.5).
> - Tipos de datos objetos (Párrafo 4.4.5.5).

4.4.6.10 Operadores sobre funciones

OMI define una serie de operadores que son aplicables a funciones.

Llamada a función: $(p0, ...)$

Aplicación parcial: $P[v1, ...]$

Acceso a función de contexto: $\sim>$

El uso de las funciones queda descritos en la sección correspondiente (Subsubsección 4.4.11).

> Para más información:
>
> - Funciones (Subsubsección 4.4.11).
> - Tipos de datos (Subsubsección 4.4.5).
> - Tipos de datos funciones (Párrafo 4.4.5.7).

4.4.6.11 Operadores comprobación de tipos

En OMI existen una serie de operadores encargados de comprobar el tipo de un dato dado. Estos devolverán un valor booleano que dependerá de que el dato sea del tipo indicado o no.

Comprobación de nulo: *isnull*

Comprobación de booleano: *isbool*

Comprobación de numérico: *isnum*

Comprobación de cadena: *isstring*

Comprobación de array: *isarray*

Comprobación de objeto: *isobject*

```
1   foo = isnull null; // foo es true
2   foo = isnull 4; //foo es false
3   foo = isbool true; //foo es true
4   foo = isbool false; //foo es true
5   foo = isbool 5; //foo es false
6   foo = isnum 4; // foo es true
7   foo = isnum "4"; // foo es false
8   foo = isnum 4.6; // foo es true
9   foo = isstring "4"; // foo es true
10  foo = isstring 4; // foo es false
11  foo = isarray {}; // foo es true
12  foo = isarray {1,2}; // foo es true
13  foo = isarray "{1,2}"; // foo es false
14  foo = isobject new Class1 (); // foo es true
15  foo = isobject 5; // foo es false
```

> Para más información:
>
> - Tipos de datos (Subsubsección 4.4.5).

4.4.6.12 Operadores condicionales

Los operadores condicionales evalúan una expresión que hace de condición, y en función del resultado obtenido realizan una determinada operación.

Operador ternario: $cond\ ?\ op1 : op2$

Fusión de nulos: $[[op1, op2, ..., opn]]$

En el operador ternario si la expresión de condición se cumple se toma el valor del operando precedido por "?", en caso contrario se toma el del último, que es precedido por ":".

```
1   foo = (4 == 4)?"OP1":"OP2"; // foo vale "OP1"
2   foo = (4 != 4)?"OP1":"OP2"; // foo vale "OP2"
3   foo = 5?"OP1":"OP2"; // foo vale "OP1"
```

Existe distintas formas del operador ternario además de la descrita. El operador ternario simplificado en valor verdadero omite el operando que se devolverá en el caso de que la condición sea cierta, en este caso se devolverá el valor de la expresión de condición. Por otro lado existe el ternario simplificado en valor falso, que omite el último operando y que tomará el valor nulo.

```
1   foo = 5?:"OP2"; // foo vale 5
2   foo = 0?:"OP2"; // foo vale "OP2"
3   foo = "COND"?"OP1"; // foo vale "OP1"
4   foo = ""?"OP1"; // foo vale null
```

La fusión de nulos opera sobre una lista de operandos. El operador comprueba los elementos secuencialmente y devuelve el primero que no sea nulo o nulo si todos los son.

```
1   foo = [[ null, null, 4 ]]; // foo vale 4
2   foo = [[ null, 4, 5 ]]; // foo vale 4
3   foo = [[ null, null, null ]]; // foo vale null
```

Para más información:

- Tipos de datos (Subsubsección 4.4.5).
- Operadores lógicos (Párrafo 4.4.6.1).
- Operadores de comparación (Párrafo 4.4.6.2).

4.4.6.13 Operadores de entrada/salida

OMMI presenta una serie de operadores que permiten leer de la entrada estándar o escribir en la salida estándar.

Entrada estándar: >>, *input*

Salida estándar: < ., << , *echo*

El operador de entrada estándar tiene como operando una expresión variable. Este se encarga de leer un flujo de caracteres de la entrada estándar hasta que lee un carácter fin de línea. El conjunto de caracteres leídos es asignado a la variable como una cadena.

```
>> foo; // Se lee de la entrada estándar hasta un carácter fin de linea y se introduce
    en foo
input foo; // Igual que el caso anterior
```

El operador de entrada tiene una segunda forma en la que se puede indicar una cadena de caracteres que se mostrará como prompt.

```
/*
   Se lee de la entrada estándar hasta un carácter fin de linea
   y se introduce en foo. Para indicar que se espera una entrada
   se utiliza la cadena "Example:"
*/
>>["Example:"] foo;
```

Los operadores de salida estándar se encargan de escribir en esta una cadena de caracteres. Estos operadores se diferencian entre si, además de en su forma léxica, en si escriben un carácter de fin de línea automático o no.

```
echo "Hola mundo"; // Escribe la cadena "Hola mundo" en la salida estándar
<. "Hola mundo"; // Escribe la cadena "Hola mundo" en la salida estándar
<< "Hola mundo"; // Escribe la cadena "Hola mundo" en la salida estándar y un salto de
     linea
```

Es posible utilizar los operadores < . y << para concatenar un flujo de cadenas de caracteres en la salida estándar. En el caso del operador < . se resolverá como el operador de concatenación de cadenas ., introduciendo las cadenas una seguida a la otra. Por otro lado, usar el operador << implica que se va a concatenar un salto de línea antes que la cadena

```
echo "Hola " . "mundo"; // Escribe la cadena "Hola mundo" en la salida estándar
echo "Hola " <. "mundo"; // Escribe la cadena "Hola mundo" en la salida estándar
echo "Hola " << "mundo"; // Escribe la cadena "Hola \mundo" en la salida estándar

<. "Hola " . "mundo"; // Escribe la cadena "Hola mundo" en la salida estándar
<. "Hola " <. "mundo"; // Escribe la cadena "Hola mundo" en la salida estándar
<. "Hola " << "mundo"; // Escribe la cadena "Hola \mundo" en la salida estándar

<< "Hola " . "mundo"; // Escribe la cadena "Hola mundo\n" en la salida estándar
<< "Hola " <. "mundo"; // Escribe la cadena "Hola mundo\n" en la salida estándar
<< "Hola " << "mundo"; // Escribe la cadena "Hola \mundo\n" en la salida estándar
```

Para más información:

- Tipos de datos (Subsubsección 4.4.5).
- Operadores sobre cadenas de caracteres (Párrafo 4.4.6.4).

4.4.6.14 Precedencia de operadores

Una expresión puede estar compuesta por varios operadores, se establece pues prioridades entre estos que determinan cómo ha de resolverse la expresión.

Los operadores quedan ordenados en el siguiente listado de más prioritarios a menos:

Llamadas: $(p0, ..)$, new

Operadores de acceso: [], ->, $\sim>$, $this$, $static$

Incrementos y decrementos: $++$, $--$

Conversión de tipos: $bool$, int, $float$, $string$

Operadores condicionales: $[[op1, ...]]$, ? :

Operaciones aritméticas de primer nivel: %, ^

Operaciones aritméticas de segundo nivel: $*$, /

Operaciones aritméticas de tercer nivel: $+$, $-$

Operaciones comparación: $==$, $!=$, $<$, $<=$, $>$, $>=$, $===$, $!==$

Operaciones lógicas de primer nivel: !

Operaciones lógicas de segundo nivel: &&

Operaciones lógicas de tercer nivel: ||

Asignaciones: $=$, $+=$, $-=$, $*=$, $/=$, $\%=$, $.=$, $\hat{}=$

Los operadores en el mismo nivel de prioridad son resueltos mediante una asociación desde la izquierda, excepto los operadores de asignación que son asociativos desde la derecha.

4.4.7. Etiquetas

Es posible etiquetar sentencias dentro del código fuente para poder cambiar el flujo de ejecución al punto que esta ocupa.

Una sentencia está formada por un identificador seguido del carácter ':' y la sentencia etiquetada.

```
1   label: << "Sentencia";
```

Las etiquetas normalmente son referenciadas por la sentencia de control *goto* la cual cambia el flujo de ejecución a la sentencia etiquetada.

Normalmente se desaconseja el uso de etiquetas y sentencias *goto* dado que dificultan la legibilidad del código. No obstante OMI soporta este mecanismo de control.

> Para más información:
>
> - Sentencia *goto* (Párrafo 4.4.8.10).

4.4.8. Sentencias de control

Las sentencias de control son construcciones del lenguaje que permiten alterar el flujo de ejecución del programa. Son muy comunes y utilizadas en los lenguajes de programación.

Se clasifican según su naturaleza, así es posible ver sentencias de control condicionales, iterativas, de salto, inclusivas o de excepción.

Muchas sentencias de control están formadas por bloques de sentencias cuya ejecución controlan. Estos bloques pueden estar formados por una sola sentencia.

4.4.8.1 Sentencia condicional *if... else...*

La sentencia condicional *if... else...* está formada por una expresión de condición, además de dos bloques de sentencias que serán ejecutadas en función la condición sea evaluada como verdadera o falsa.

```
1   >>["Dime el valor de foo:"] foo;
2   if ( foo < 10) {
3      << foo << " es menor que 10"; // Se ejecuta si la condición es verdadera
4   }
5   else {
6      << foo << " no es menor que 10"; // Se ejecuta si la condición es falsa
7   }
```

Si el caso de evaluación negativa no implica la ejecución de ninguna sentencia, el bloque *else* puede omitirse.

```
>>["Dime el valor de foo:"] foo;
if ( foo < 10) {
   << foo << " es menor que 10"; // Se ejecuta si la condición es verdadera
}
```

Otra forma de esta sentencia condicional es *if... elif... else...*, en donde se anidan varias sentencias de este tipo. Se pueden anidar tantos *elif* como sea necesario.

```
>>["Dime el valor de foo:"] foo;
if ( foo < 10) {
   << foo << " es menor que 10"; // La primera condición es verdadera
}
elif (foo == 10) {
   << foo << " es igual que 10"; // La primera es falsa y la segunda verdadera
}
else {
   << foo << " es mayor que 10"; // Todas las condiciones son falsas
}
```

4.4.8.2 Sentencia condicional *switch... case...*

La sentencia *switch... case...* agiliza la toma de decisiones múltiples. Opera de igual forma que varios *if... else...* anidados, pero presenta una sintaxis que en muchos casos favorece la legibilidad y la rapidez de programación.

Se compone de un dato a comparar y un bloque de sentencias etiquetadas con casos. Los casos están formados por una expresión que será comparada con el dato. La ejecución pasará a la primera sentencia etiquetada con el caso que sea igual al dato comparado, procediéndose a ejecutar todas las sentencias siguientes.

```
>>["Sentencia inicial:"] foo;
switch (foo) {
   case 0:
      << "Sentencia 0"; // Se ejecuta si foo es 0
   case 1:
      << "Sentencia 1"; // Se ejecuta si foo es 0 o 1
   case 2:
      << "Sentencia 2"; // Se ejecuta si foo es 0, 1 o 2
}
```

Es una práctica muy común, para omitir la ejecución de algunas sentencias, el utilizar la sentencia *break*, la cual finaliza la ejecución del bloque de sentencias actual.

```
>>["Sentencia inicial:"] foo;
```

```
2   switch (foo) {
3      case 0:
4         << "Sentencia 0"; // Se ejecuta si foo es 0
5         break;
6      case 1:
7         << "Sentencia 1"; // Se ejecuta si foo es 1
8         break;
9      case 2:
10        << "Sentencia 2"; // Se ejecuta si foo es 2
11  }
```

Si ningún caso es igual al dato comparado no se ejecuta ninguna sentencia del bloque, a menos que se encuentre la etiqueta *default*.

```
1   >>["Sentencia a ejecutar:"] foo;
2   switch (foo) {
3      case 0:
4         << "Sentencia 0"; // Se ejecuta si foo es 0
5         break;
6      case 1:
7         << "Sentencia 1"; // Se ejecuta si foo es 1
8         break;
9      case 2:
10        << "Sentencia 2"; // Se ejecuta si foo es 2
11        break;
12     default:
13        << "Sentencia por defecto"; // Se ejecuta si foo no es 0, 1 ni 2
14  }
```

Para más información:

- Sentencia break (Párrafo 4.4.8.8).

4.4.8.3 Sentencia iterativa *while...*

La sentencia *while...* es una estructura de control iterativa que permite la ejecución de un bloque de sentencias repetidamente mientras que se cumpla una determinada expresión de condición. Se compone pues de una expresión que será evaluada como un dato booleano y un bloque de sentencias.

```
1   /*
2      Imprime la cadena
3      "Sentencia" 10 veces.
4   */
5   foo = 0;
6   while (foo < 10) {
7      << "Sentencia";
8      foo++;
9   }
```

La expresión de condición es evaluada al comienzo de cada ejecución del bloque de sentencias, por lo que si alguna sentencia vuelve la condición falsa se seguirá con la ejecución del bloque hasta la próxima iteración.

4.4.8.4 Sentencia iterativa *do... while*

La sentencia *do... while* es una estructura de control iterativa que permite la ejecución de un bloque de sentencias repetidamente mientras que se cumpla una determinada expresión de condición, y al menos una vez. Se compone pues de una expresión que será evaluada como un dato booleano y un bloque de sentencias.

```
1  /*
2     Imprime la cadena
3     "Sentencia" 10 veces.
4  */
5  foo = 0;
6  do {
7     << "Sentencia";
8     foo++;
9  } while (foo < 10);
```

A diferencia de la sentencia *do... while* la condición es evaluada al final de la ejecución del bloque de sentencias por lo que se asegura que este se ejecutará al menos una vez. Al igual que *do... while* si alguna sentencia vuelve falsa la condición se proseguirá con la ejecución hasta que se produzca la evaluación.

4.4.8.5 Sentencia iterativa *for...*

La sentencia *for...* es una estructura de control iterativa que permite la ejecución de un bloque de sentencias repetidamente mientras que se cumpla una determinada expresión de condición. La sentencia *for...* conlleva la ejecución de la expresión de inicialización al inicio de la sentencia, y de una expresión de iteración al final de cada ejecución del bloque.

```
1  /*
2     Imprime la cadena
3     "Sentencia" 10 veces.
4  */
5  for (foo = 0; foo < 10; ++foo){
6     << "Sentencia";
7  }
```

Es OMI es necesario especificar todas las expresiones que conforman la sentencia *for...*, si alguno es omitido se producirá un error sintáctico.

4.4.8.6 Sentencia iterativa *for... as...* y *for... in...*

OMI ofrece unas formas de sentencias *for...* que permiten ejecutar un bloque de sentencias para cada elemento contenido en una determinada expresión. Para ello se debe especificar la expresión a recorrer, el símbolo variable con el que se referenciará al elemento actual y el bloque de sentencias que se ejecutará para cada elemento contenido en la

expresión.

```
/*
   Imprime la cadena
   "Sentencia" y su índice
   10 veces.
*/
for ( 10 as i ){
   << "Sentencia " . i;
}
/*
   Imprime la cadena
   "Sentencia" y su índice
   10 veces.
*/
for ( i in 10 ){
   << "Sentencia " . i;
}
```

La diferencias entre *for... as...* y *for... in...* son puramente sintácticas, siendo el orden en que se especifican la expresión conjunto y el símbolo variable su principal diferencia.

El comportamiento de estas estructuras cambia en función el tipo de dato de la expresión conjunto.

Booleano: Si es true se ejecutará el bloque indefinidamente y el símbolo variable tendrá el valor true. Si es falso el bloque de sentencias no se ejecutará.

Numéricos: Si es mayor que se 0 el bloque se ejecutará desde 0 hasta el valor de la expresión y el símbolo variable tendrá el valor numérico de cada iteración. En otro caso no se llega a ejecutar el bloque.

Cadenas de caracteres: Se ejecuta el bloque de sentencias por cada carácter en la cadena, el símbolo variable tomará como valor la cadena correspondiente al caracter de cada iteración.

Array: Se ejecuta el bloque de sentencias por cada elemento en el array, el símbolo variable tomará como valor el elemento de cada iteración.

Es posible especificar un par de símbolos variables separados por el carácter ":". Esto hará que si el elemento actual en el recorrido tiene una clave que lo referencia esta se guarde en el primero y el valor en el segundo. Si no tuviera clave solo se asignará el valor al segundo símbolo variable.

```
/*
   Imprime todas las
   claves y valores del array
*/
array = {'k0':'v0', 'k1':'v1', 'k2':'v2'};
for ( array as key:value ){
   << key . " => " . value;
}
```

4.4.8.7 Sentencia iterativa ágil

OMI ofrece una estructura de control iterativa cuya sintaxis ha sido simplificada, su funcionalidad extendida. La sentencia de control de iteración ágil funciona igual que una sentencia *for... in...*, pero sin que se produzca una asignación de cada elemento a un símbolo variable. En su lugar se puede acceder al elemento en curso mediante un operador de acceso especial.

La sentencia de iteración ágil se compone de una expresión a recorrer y de un bloque de sentencias que se ejecutará para cada elemento en la expresión.

```
1  /*
2     Imprime la cadena
3     "Sentencia" 10 veces.
4  */
5  $(10){
6     << "Sentencia";
7  }
```

Durante la ejecución del bloque de sentencias es posible acceder al elemento actual mediante el operador "$".

```
1  /*
2     Imprime la cadena
3     "Sentencia" y su índice
4     10 veces.
5  */
6  $(10){
7     << "Sentencia " . $;
8  }
```

Es posible anidar dos o más sentencias de iteración ágil y acceder al índice de cada una de ellas, añadiendo una expresión que se corresponde con el nivel de anidamiento tras el operador "$" y delimitada entre llaves.

```
1  /*
2     Imprime las tablas de multiplicar del
3     0 al 9.
4  */
5  $(10){ // Nivel 0
6    << "Tabla de multiplicar del " . ${0};
7    $(10) { // Nivel 1
8      << ${0} . " x " . ${1} . " = " . (${0} * ${1});
9    }
10 }
```

Si el operador de acceso no presenta índice este se corresponderá con el nivel actual.

4.4.8.8 Sentencia de salto *break*

La sentencia *break* permite terminar la ejecución de un bloque de sentencias correspondientes a una estructura iterativa y saltar a la siguiente sentencia tras el bloque.

```
1  /*
2     Imprime el valor dado
3     hasta que este es 0.
4  */
5  while(true){
6    >>["Dame un valor (0 para salir)"] foo;
7    if (foo === "0")
8      break;
9    << foo;
10 }
11 << "Finalizando";
```

Es posible salir de dos bloques iterativos que se encuentren anidados, indicando tras *break* cuantos satos se desean realizar.

```
1  /*
2     Imprime el valor dado
3     hasta que este es 0.
4     Imprimiendo cuantas
5     solicitudes se han
6     realizado cada 10 veces
7  */
8  count = 0;
9  while (true) {
10   i = 0;
11   while(i < 10){
12     >>["Dame un valor (0 para salir)"] foo;
13     count ++;
14     if (foo === "0")
15       break 2;
16     << foo;
17     i ++;
18   }
19
20   << "Se han solicitado " . count . " valores ";
21 }
22 << "Finalizando";
```

4.4.8.9 Sentencia de salto *continue*

La sentencia *continue* permite terminar la ejecución actual de un bloque de sentencias correspondientes a una estructura iterativa. De esta forma se salta a la comprobación y ejecución (si fuera el caso) de la siguiente iteración.

```
1  /*
2     Imprime el valor dado,
3     excepto si este es "jump",
4     hasta que es 0.
5  */
6  while(true){
```

```
7     >>["Dame un valor (0 para salir, jump para saltar)"] foo;
8     if (foo === "0")
9       break;
10    if (foo === "jump")
11      continue;
12    << foo;
13  }
14  << "Finalizando";
```

Es posible utilizar la sentencia *continue* para saltar a la siguiente iteración de una estructura iterativa concreta de varias anidadas. Para ello tras el literal se debe indicar una expresión cuyo valor sea el número de bloques anidados que se desea saltar.

```
1   /*
2      Imprime el valor dado hasta que este es 0.
3      Imprimiendo cuantas solicitudes se han realizado cada 10 veces.
4      Es posible forzar el reinicio del contador mediante el valor "restart"
5   */
6   count = 0;
7   while (true) {
8     i = 0;
9     while(i < 10){
10      >>["Dame un valor (0 para salir, restart para reiniciar contador)"] foo;
11      count ++;
12      if (foo === "0")
13        break 2;
14      if (foo === "restart"){
15        << "Reiniciando contador ";
16        count = 0;
17        continue 2;
18      }
19      << foo;
20      i ++;
21    }
22    << "Se han solicitado " << count << " valores ";
23  }
24  << "Finalizando";
```

4.4.8.10 Sentencia de salto *goto*

La sentencia *goto* se compone de un identificador y permite cambiar el flujo de ejecución a la sentencia etiquetada con dicho identificador.

```
1   /*
2      Se imprime la cadena
3      "Sentencia" indefinidamente
4   */
5   label: << "Sentencia";
6   goto label;
```

En general se desaconseja el uso de etiquetas y sentencias *goto* dado que dificultan la legibilidad del código.

Lo habitual es que la sentencia *goto* se encuentre condicionada para que el salto no se

ejecute siempre.

```
/*
   Se imprime la cadena
   "Sentencia" 10 veces
*/
foo = 0;
label: << "Sentencia " . foo;
if (foo < 10){
   foo++;
   goto label;
}
```

> Para más información:
>
> - Etiquetas (Subsubsección 4.4.7).
> - Sentencia condicional *if... else...* (Párrafo 4.4.8.1).

4.4.8.11 Sentencia *include*

La sentencia *include* permite separar el código fuente en diferentes ficheros. Se compone de una expresión que se corresponde con el fichero a incluir. Al ser evaluada, la ejecución pasa a la primera sentencia en el fichero.

```
/*
   fichero1.omi
   Se imprime el contenido de este fichero
   y se incluye el fichero 2
*/
<< "Contenido fichero 1";
include "fichero2.omi";
```

```
/*
   fichero2.omi
   Se imprime el contenido de este fichero
*/
<< "Contenido fichero 2";
```

Si la cadena facilitada como expresión no se corresponde con un fichero del sistema se producirá un error y finalizará la ejecución del script.

4.4.8.12 Sentencia *exit*

La sentencia *exit* permite finalizar la ejecución del script en cualquier parte del mismo.

```
/*
   Finaliza el script sin llegar
   a ejecuta la ultima sentencia
*/
<< "SENTENCIA EJECUTADA";
exit;
```

```
7    << "SENTENCIA NO EJECUTADA";
```

A diferencia de otros lenguajes de programación en OMI no existe una forma de finalizar el script devolviendo un estado de error.

4.4.8.13 Sentencia *with*

La sentencia *with* permite establecer un objeto como contexto dentro de un bloque de sentencias. Se forma mediante una expresión correspondiente al objeto y un bloque de sentencias que será ejecutada con el objeto como contexto.

Cuando se establece un objeto como contexto todas las funciones que se llamen, y que no se encuentren definidas, serán llamadas como métodos del objeto.

```
1    class Foo {
2       ~ identity () {
3          << "Soy Foo";
4       }
5    }
6    foo = new Foo ();
7    with (foo) {
8       identity(); // Llama al método identity del objeto foo
9    }
```

4.4.8.14 Sentencia *try... catch...*

La sentencia *try... catch...* permite delimitar mediante un bloque de sentencias un comportamiento en el que se pueden dar excepciones. Además, mediante otro bloque de sentencias y un símbolo variable, se puede especificar cómo serán tratadas las excepciones que se den.

A diferencia de otros lenguajes de programación OMI solo permite un bloque de sentencias *catch*. No comprueba la clase de la excepción que se ha dado, esto recae en resposabilidad del programador.

```
1    class Exc {
2       var = null;
3       ~ Exc (num) {
4          this->var = num;
5       }
6       ~ printError () {
7          << "Error " . this->var;
8       }
9    }
10   try {
11      >>["Dame un valor:"] foo;
12      if ( foo === "" ){
13         throw new Exc (404);
```

```
14      }
15      << foo;
16   }catch (exc) {
17      exc->printError ();
18   }
```

Para más información:

- Excepciones (??).
- Sentencia *throw* (Párrafo 4.4.8.15).

4.4.8.15 Sentencia *throw*

La sentencia *throw* permite lanzar una excepción que puede ser atrapada por una sentencia *try... catch...*. Esta sentencia se construye mediante una expresión cuyo valor será asociado al símbolo variable del bloque de sentencias *catch*.

```
1    class Exc {
2       var = null;
3       ~ Exc (num) {
4          this->var = num;
5       }
6       ~ printError () {
7          << "Error " << this->var;
8       }
9    }
10   try {
11      >>["Dame un valor:"] foo;
12      if ( foo === "" ){
13         throw new Exc (404);
14      }
15      << foo;
16   }catch (exc) {
17      exc->printError ();
18   }
```

Si una excepción lanzada con *throw* no sucede dentro de una sentencia *try... catch* controlada se producirá un error y se detendrá la ejecución del script.

Para más información:

- Excepciones (??).
- Sentencia *try... catch...* (Párrafo 4.4.8.14).

4.4.8.16 Sentencia *sleep*

La sentencia *sleep* hace que la ejecución del programa se pare un número de segundos dados.

```
1    << "Antes de parar";
2    sleep (10);
3    << "Tras 10 segundos";
```

4.4.8.17 Sentencia *typeof*

La sentencia *typeof* imprime en la salida estándar el tipo de dato del valor referenciado por un símbolo variable.

```
1    foo = 10;
2    typeof foo; // Imprime Arithmetic
3    foo = "STR";
4    typeof foo; // Imprime String
5    foo = {1,2};
6    typeof foo; // Imprime Array(2)
```

4.4.8.18 Sentencia *datInfo*

La sentencia *datInfo* permite consultar el estado interno de un dato. Esto es la posición de memoria que ocupa, el tipo y el número de referencias que tiene.

```
1    datInfo 5; // Imprime ptr(0x1d72e00), type(Arithmetic: 5), refs(0)
2    datInfo "A"; // Imprime ptr(0x1d72f70), type(String: A), refs(0)
3    foo = 10;
4    datInfo foo; // ptr(0x1d73f00), type(Arithmetic: 10), refs(1)
5    bar = foo;
6    datInfo bar; // ptr(0x1d73f00), type(Arithmetic: 10), refs(2)
```

La sentencia *datInfo* también admite la forma *datinfo*.

4.4.9. Variables

En programación una variable se define como un espacio que es asociado a un nombre o identificador, el contenido alojado en dicho espacio es llamado el valor de la variable. Mediante el nombre es posible utilizar la variable en expresiones con independencia de la información exacta que esta referencia. Cuando una expresión que contiene una variable es resuelta se obtiene el valor contenido en la misma.

El valor de una variable puede cambiar durante la ejecución del programa. Para alterar el contenido de una variable lo habitual es utilizar el operador de asignación. En OMI la asignación de una variable es destructiva, es decir, la variable modifica su valor sobrescribiendo el valor anterior.

Las variables suponen un recurso esencial para los lenguajes de programación imperativos, ya que son el mecanismo básico para guardar el estado del sistema. Las variables en matemáticas forman parte de definiciones y una vez atribuido su valor este no variará, esto difiere del concepto de variable que presentan los lenguajes de programación imperativos.

OMI es un lenguaje de programación de tipado dinámico, y por tanto, el tipo de dato de una variable está determinado por el valor actual de la misma, y no se encuentra asociado a la variable. En un momento dado una variable podría contener un valor entero, y en otro una cadena de caracteres.

En OMI el tipo de dato del valor almacenado en una variable es desconocido hasta que es obtenido en la resolución de una expresión. Los operadores, funciones y demás recursos del lenguaje usan el valor para operar, con independencia de la variable que lo contuviera.

En OMI no es necesario declarar ni inicializar una variable antes de su uso. Si una variable sin valor es usada en una expresión se toma el valor nulo.

Internamente OMI representa las variables como referencias a los valores, más que como contenedores de estos. En OMI no es necesario llevar una gestión los datos que han sido creados y asignados a las variables de forma dinámica, un subsistema denominado recolector de basura será el encargado de esta tarea.

Las variables deben ser nombradas mediante un identificador válido, y no deben coincidir con ninguna palabra reservada.

Las variables en OMI pueden presentar un ámbito local y global. Una variable puede ser local al flujo principal o a una función o método. Por defecto las variables son locales. Una variable local solo puede ser accesible desde donde se utilizó inicialmente. Si en otro lugar se hace uso del mismo identificador se consideran variables distintas. Las variables globales son accesibles desde cualquier parte de la aplicación. Para usar una variable como global debe ser declarada como tal con el literal *global*.

```
1  foo = 10; // foo es una variable local al flujo principal
2  global bar; // bar es una variable global.
3  bar = 20;
4  ~ func () {
5     foo = 30; // foo es una variable local a la función func
6     bar = 40; // Se asigna a la variable global bar
7  }
```

Para más información:

- Operador de asignación (Párrafo 4.4.6.5).
- Identificadores (Subsubsección 4.4.4)

4.4.10. Referencias

Una referencia es un valor que permite acceder indirectamente a un dato almacenado en un símbolo variable. Cuando se opera sobre una referencia se hace sobre el dato almacenado en la variable.

Es posible obtener una referencia anteponiendo el símbolo "&" ante el símbolo variable.

```
1   foo = 10;
2   bar = &foo; // Se obtiene una referencia de foo
3   << foo . " - " . bar; // Imprime 10 - 10
4   bar = 20;
5   << foo . " - " . bar; // Imprime 20 - 20
```

En OMI es posible obtener una referencia de cualquier expresión que sea variable, y acceder su valor de forma indirecta. Así por ejemplo se puede obtener una referencia a una posición de un array, a un atributo de un objeto, etc.

```
1   foo = {{1,2}, 3};
2   bar = &foo[0][1];
3   bar = 4;
4   << foo[0][1];
```

No es posible obtener una referencia a una posición dentro de una cadena, esto es debido a que en OMI las cadenas de caracteres son un tipo de dato simple. En su lugar se copiará el valor del carácter en la posición como otra cadena, por lo que la referencia no tendrá efecto.

```
1   foo = "ABCD";
2   bar = &foo[0];
3   bar = "Z";
4   << bar; // Imprime "Z"
5   << foo; // Imprime "ABCD". La referencia no tiene efecto.
```

Sintácticamente es posible obtener una referencia de un valor constante, no obstante esto hará que se tome el valor en si, sin que tenga efecto alguno.

Las referencias pueden ser utilizadas como un dato, por lo que pueden ser pasadas como parámetro. En este caso cualquier acceso al parámetro dentro de la función se hará sobre las variables a las que referencia.

```
1   ~ inc (param) {
2       param += 1;
3   }
4
5   a = 20;
6   inc(&a);
7   << a; // Imprime 21
```

Es posible hacer que algunos parámetros de una función sean pasados siempre por referencia. Para ello se antepone & al parámetro en la definición de la función. Ver paso de parámetros por referencia (Párrafo 4.4.11.2).

Las funciones y métodos pueden devolver referencias. De esta forma si una llamada a función es asignada se hará la operación sobre la referencia devuelta, por lo que cualquier acceso se hará sobre el símbolo variable a la que esta apunta. Si la referencia es a un símbolo variable de ámbito local, el cual es liberado al terminar la llamada, el valor será copiado.

```
1   class example {
2      private attr = null;
3
4      ~ example (attr) {
5         this->attr = attr;
6      }
7
8      ~ getRef () {
9         return &this->attr;
10     }
11
12     ~ printAttr () {
13        << this->attr;
14     }
15  }
16
17  foo = new example (20);
18  foo->printAttr(); // Imprime 20
19  << foo->attr; // Error acceso a elemento privado
20  bar = foo->getRef(); // bar es una referencia a foo->attr
21  bar = 40;
22  foo->printAttr(); // Imprime 40
```

4.4.11. Funciones

Una función es un conjunto de sentencias que tienen un objetivo particular. Se corresponde con un subalgoritmo dentro del algoritmo principal.

Una función se define normalmente mediante un identificador, una lista de parámetros (delimitados por paréntesis y separados por coma) y un bloque de sentencias. La definición de una función debe empezar por el carácter "~" o la palabra reservada *function*.

```
1   ~ func1 (param1, param2) {
2      << "Soy fun1 con dos parametros:";
3      << "param1 => " . param1;
4      << "param2 => " . param2;
5   }
6
7   function func2 (param1, param2) {
8      << "Soy func2 con dos parametros:";
9      << "param1 => " . param1;
10     << "param2 => " . param2;
11  }
```

Los parámetros de la función suponen un mecanismo para que esta pueda recibir

datos con los que operar y realizar su tarea. Los parámetros son variables que pueden ser utilizadas en el bloque de sentencias y cuyos valores serán dado cuando la función es llamada.

Las funciones pueden ser llamadas desde cualquier parte del código, lo que supondrá la ejecución del bloque de sentencias. Para realizar una llamada a una función se utilizará el nombre de la función seguido de los valores que serán atribuidos a los parámetros, delimitados por paréntesis y separados por coma. La atribución de valores a los parámetros se hace de forma posicional.

```
~ func1 (param1, param2) {
   << "Soy fun1 con dos parametros:";
   << "param1 => " . param1;
   << "param2 => " . param2;
}

func1 ("valor1", "valor2");
```

En una llamada los parámetros son pasados, a no ser que se indique lo contrario, por valor. Esto quiere decir que los valores son copiados a los parámetros, lo que implica que cualquier modificación de estos solo tendrá efecto en el bloque que define la función.

```
~ func1 (foo) {
   foo = 20;
   << foo;  // Imprime 20
}

foo = 40;
func1 (foo);
<< foo; // Imprime 40
```

Una función puede devolver un valor que será atribuido a la llamada. Para ello se hace uso de la sentencia *return*. Esta sentencia, que puede formar parte del bloque de sentencias de una función, esta formada por el valor que será devuelto y atribuido a la llamada. La ejecución de la sentencia *return* implica la finalización de la función.

La llamada a un función es considerada un operador y puede formar parte de expresiones. El valor de la llamada es dado por la ejecución de la sentencia *return* en el bloque de sentencias de esta. Si la ejecución de la función termina sin una sentencia *return* se tomará el valor nulo.

```
~ sum (op1, op2) {
   return op1 + op2;
}

<< sum(1,2) * 4; // Imprime 12
```

Las llamadas a función son resueltas antes que cualquier otro operador, ya que presentan el mayor nivel de prioridad.

Si una función es llamada sin que esta se encuentre definida se producirá un error semántico. También se producirá un error si el número de valores facilitados como parámetros en la llamada no se corresponden con la definición de la función. Cabe destacar que las definiciones de funciones son ejecutadas de forma secuencial dentro del flujo del programa, por lo que realizar un llamada de un función que se encuentre definida en sentencias posteriores producirá un error como si esta no se encontrara definida, por ello es buena práctica disponer las definiciones de funciones al comienzo del programa.

Los nombres de las funciones deben seguir las reglas normales de uso de identificadores, y no corresponderse con ninguna palabra reservada.

Si dos o más definiciones de función comparten el mismo identificador la última definición interpretada será la que se corresponda con dicho identificador.

```
1  ~ func () {
2      << "FUNC 1";
3  }
4
5  ~ func () {
6      << "FUNC 2";
7  }
8
9  func(); // Imprime "FUNC 2"
```

A pesar de ser considerado un lenguaje imperativo, OMI presenta algunos recursos y mecanismos propios de la programación funcional. En las subsecciones siguientes se presentan los recursos y mecanismos que pueden ser utilizados y aplicados a las funciones.

4.4.11.1 Parámetros con valores por defecto

En OMI es posible definir funciones cuyos parámetros presenten un valor por defecto, es decir, en caso de ser omitido en una llamada dicho parámetro tendrá el valor especificado.

En una llamada es obligatorio dar el valor de todos los parámetros excepto los que tienen definido un valor por defecto. Dado que la atribución de valores a los parámetros se hace de forma posicional, los parámetros con valores por defecto serán los últimos para que puedan omitirse en la llamada.

En la definición de una función se puede especificar que determinados parámetros tendrán valores por defecto siguiendo estos mediante el signo igual y el valor que tendrán.

```
1  ~ sum (op1, op2 = 4) {
2      return op1 + op2;
3  }
4
5  << sum(1,2) * 4; // Imprime 12
6  << sum(1) * 4; // Imprime 20
```

4.4.11.2 Paso de parámetros por referencia

Por defecto los parámetros de una llamada a función son pasados por valor, lo que implica que cualquier modificación en los mismos dentro de la función no tendrá efecto fuera de esta.

Es posible definir funciones donde algunos parámetros sean pasado como referencia en las llamadas. Esto es que si en la llamada se facilita un símbolo variable como valor de dicho parámetro, cualquier modificación realizada en el mismo será aplicado a la variable fuera de la función.

En la definición de una función se puede indicar que un determinado parámetro será pasado por referencia anteponiendo el símbolo "&" a dicho parámetro.

```
1   ~ func1 (&foo) {
2       foo = 20;
3       << foo;  // Imprime 20
4   }
5
6   foo = 40;
7   func1 (foo);
8   << foo; // Imprime 20
```

Si un valor constante es dado como parámetro por referencias se producirá un error.

4.4.11.3 Definición como parte de expresiones

En OMI las funciones son consideradas tipos de datos, por lo que es posible asignarlas a variables, operar sobre ellas, etc. Sin embargo, no se puede realizar una conversión de una función a otro tipo de dato y viceversa, por lo que la mayoría de operadores producirán un error al resolverse si operan con una función y otro tipo de dato.

Es posible utilizar la definición de una función como expresión en una operación de asignación.

```
1   foo = ~ sum (param1, param2) { return param1 + param2; };
2   << foo (4, 5); // Imprime 9
3   ((x < 10)?~(){<<"A";}:~(){<<"B"})(); // Si x < 10 imprime "A", si no imprime "B"
```

4.4.11.4 Referencias a funciones

En OMI se nombran de igual forma a las funciones, las variables y las clases. Para determinar si un identificador en una expresión hace referencia a una función u otro elemento se utiliza el contexto.

En muchos casos es necesario indicar explícitamente que el objeto que se desea utilizar es una función ya definida, y no una variable u otro elemento. Para ello se antepone al identificador de la función el par de símbolos "∼ &".

```
1   ~ foo () {
2      return "FUNCIÓN";
3   }
4   foo = "VARIABLE";
5   bar = foo; // bar vale "VARIABLE";
6   bar = ~&foo; // var vale la función que devuelve "FUNCIÓN"
```

Por regla general todos los operadores toman, a no ser que se indique lo contrario, los identificadores como variables. Las excepciones a esta regla son los siguientes operadores:

- Llamada a función
- Aplicación parcial
- *new*.

4.4.11.5 Funciones anónimas

OMI permite definir funciones sin identificador que las nombre, lo que se denomina funciones anónimas. Normalmente las funciones anónimas son utilizadas como dato en otras expresiones.

Para definir una función anónima solo se ha de omitir el identificador.

```
1   bar = foo = ~() { << "FUNCIÓN ANÓNIMA"; };
2   foo(); // IMPRIME "FUNCIÓN ANÓNIMA";
3   bar(); // IMPRIME "FUNCIÓN ANÓNIMA";
```

4.4.11.6 Expresiones parametrizadas

p Una expresión parametrizada es similar a una función. Permite definir una expresión a partir de unos parámetros.

La expresión parametrizada puede ser utilizada como dato para llevar a cabo operaciones sobre ella. Así es posible asignarla, llamarla con unos valores dado, ser devuelta como valores de función, etc.

Para definir una expresión parametrizada se utiliza el símbolo "∼" seguido de una lista de parámetros separados por coma, el símbolo ":" y la expresión a parametrizar, la cual

normalmente hará uso de los parámetros. Es posible obtener el valor para unos determinados parámetros utilizando el operador de llamada a función.

```
sum = ~ x, y : x + y;
<< sum (4,5);
```

4.4.11.7 Funciones de orden superior

Las funciones de orden superior son funciones que cumplen una de las siguientes premisas:

- Reciben como parámetros una o más funciones.
- Devuelven como valor una función.

En OMI las funciones son tipos de datos, por lo que es posible crear funciones de orden superior de forma natural a la sintaxis del lenguaje.

```
~ call (f) {  // Función de orden superior que recibe una función como parámetro
   f();
}

~ printA () { << "A" }
~ printB () { << "B" }

call (~&printA); // Imprime A
call (~&printB); // Imprime B
```

```
~ getOp (code) {  // Función de orden superior que devuelve una función
   switch (code) {
      case 0:  return ~(a,b) { return a + b }; break;
      case 1: return ~(a,b) { return a - b }; break;
      default: return ~(a,b) { << "WRONG CODE"; return null; }
   }
}

func = getOp (0);
<< func (4, 5); // Imprime 9

func = getOp (1);
<< func (4, 5); // Imprime -1

func = getOp ("*");
<< func (4, 5); // Imprime "WRONG CODE"
```

4.4.11.8 Clausura de funciones

Si una función es definida en el bloque de sentencias de otra, cuando la función interna sea llamada, en su ejecución podrá acceder a las variables definidas en el bloque en el momento en el que se definió.

La clausura permite que funciones que sean definidas en un entorno puedan acceder a las variables de dicho entorno.

```
~ addX (X) {
    return ~(num) {
        return num + X; // X pertenece al entorno de addX
    }
}

add1 = addX(1);
add2 = addX(2);

<< add1 (4); // Imprime 5
<< add2 (4); // Imprime 6
```

4.4.11.9 Función de contexto

Por regla general la función de contexto se refiere a la función en ejecución. En OMI es posible acceder a la función de contexto mediante el operador "∼>", esto permite definir funciones recursivas de forma simple.

```
~ factorial (num) {
    if (num == 2) return 2;
    return num * ~>(num-1);
}
```

En otros casos la función de contexto puede mantener otras referencias diferentes a la función en ejecución.

4.4.11.10 Decoradores

Un decorador es una función que:

- Toma una función como parámetro.
- Devuelve otra función que utiliza la función recibida como parámetro en su bloque de sentencias.

Es posible crear un decorador mediante funciones gracias al principio de clausura.

```
~ generateFileHTML (func) { //Decorador mediante funciones
   return ~(title, params){
      << "<html>";
      << "<head>";
      << "<title>" << title << "</title>";
      << "</head>";
      << "<body>";
      func(params);
      << "</body>";
      << "</html>";
   }
}

search = ~(params) { << "Buscando: " << params; };
info = ~(params) { << "Información sobre: " << params; };

generateSearchHTML = generateFileHTML (search);
generateInfoHTML = generateFileHTML (info);

generateSearchHTML("Búsqueda", "concepto");
generateInfoHTML("Información", "producto");
```

En OMI existe una construcción propia del lenguaje denominada decorador. Esta simplifica el proceso anterior. Un decorador se define de forma parecida a una función: presenta un identificador que lo nombra, una lista de parámetros y un bloque de sentencias. La lista de parámetros se corresponde con los que recibirá la función devuelta por el decorador, la cual se definirá a partir del bloque de sentencias. En el bloque de sentencias se podrá hacer referencia a la función a decorar mediante el operador de función de contexto "∼>". Para definir un decorador se utiliza el literal "∼∼".

```
~~ generateFileHTML (title, params) { // Decorador mediante construcciones del
    lenguaje
   << "<html>";
   << "<head>";
   << "<title>" << title << "</title>";
   << "</head>";
   << "<body>";
   ~>(params);
   << "</body>";
   << "</html>";
}

search = ~(params) { << "Buscando: " << params; };
info = ~(params) { << "Información sobre: " << params; };

generateSearchHTML = generateFileHTML (search);
generateInfoHTML = generateFileHTML (info);

generateSearchHTML("Búsqueda", "concepto");
generateInfoHTML("Informaciún", "producto");
```

4.4.11.11 Aplicación parcial

La aplicación parcial toma una función y devuelve otra basada en esta, donde algunos de sus parámetros presentan ya un valor.

Para la aplicación parcial se utiliza el operador con el mismo nombre. Este consiste en el literal "*P*" seguidos de una lista con los valores que se aplicarán y la función sobre la que se realizará la operación. La lista de parámetros consiste en pares parámetro/valor separados por el signo "=", y entre si por coma. Los parámetros se han de corresponder con parámetros de la función sobre la que se realizará la operación. La función resultante será una función que recibe tantos parámetros como la función original exceptuando aquellos que tienen aplicados un valor.

```
1  ~ addX (num, X) {
2      return num + X;
3  }
4
5  add4 = P[X=4]addX;
6  add8 = P[X=8]addX;
7
8  << add4(5); // Imprime 9
9  << add8(5); // Imprime 13
```

4.4.12. Listas por comprensión

En OMI es posible definir una lista o un diccionario por compresión. Para ello se utiliza una construcción del lenguaje similar a la sentencia iterativa *for*, salvo que en cada iteración genera un elemento del array asociado a la lista o diccionario.

La definición de una lista por compresión debe ir entre paréntesis.

La forma más simple de crear una lista por compresión es mediante una expresión de generación, el literal *for* y una variable que iterará sobre una expresión conjunto. En cada iteración la variable iteradora tomará un valor del conjunto. Para cada elemento se calculará el valor de la expresión generadora, la cual normalmente contendrá la variable iteradora, y será introducido en el array resultado. El conjunto sobre el que se itera dependerá del tipo de dato obtenido al evaluar la expresión conjunto:

Booleano: Se itera hasta que se evalúa como falso. En cada iteración se asigna el entero correspondiente al número de la misma.

Numérico: Si es positivo se itera desde cero hasta el valor sin incluirlo. Si es negtivo no se itera.

Cadena de caracteres: Se itera por cada carácter en la cadena.

Array: Se itera por cada elemento en el array.

```
/*
   Array de 10 elementos: desde el 0 al 9
*/
 foo = ( y++ for x in y < 10 )
/*
   Array de 10 elementos donde las
   posiciones pares valen 0 y las
   impares 1
*/
foo = ( x % 2 for x in 10 );
/*
   Array que contiene "A", "B", "C" y "D"
*/
foo = ( x for x in "ABCD" );
/*
   Array que contiene 1, 2 y 4
*/
foo = ( x/2 for x in {2,4,8} );
```

En cada iteración la expresión de conjunto es evaluada antes de obtener el elemento correspondiente.

Es posible filtrar los datos que serán incluidos en el array, añadiendo el literal if y la condición que se debe cumplir.

```
/*
   Array de los pares
   del 0 al 9
*/
foo = ( x for x in 10 if x % 2 == 0 );
```

Dado un array asociativo como conjunto es posible iterar sobre los pares clave/valor que lo conforman , para ello se define el par de variables que se van a utilizar separadas por ":".

```
/*
   Array que contiene "key0 => val0"
   y "key1 => val1"
*/
foo = ( k." => ".v for k:v in {"key0":"val0","key1":"val1"} );
```

El array generado puede ser asociativo utilizando dos expresiones de generación separadas por ":" que se utilizarán para construir las claves y el valores.

```
/*
   Array asociativo con las claves "key0", "key1" y "key2",
   cuyos valores son 2, 4 y 6 respectivamente.
*/
foo = ( "key".k : x for k:x in {2,4,6} );
```

Es posible indicar un bloque de sentencias que será ejecutado con cada iteración. Este bloque irá después del conjunto a iterar.

```
/*
   Array con los valores dados.
   Se solicita valores y se van introduciendo
   en el array hasta que no se da valor
*/
foo = ( x for i in x !== "" {>>["Posición ".i.": "] x} if x !== "" );
```

4.4.13. Clases de objeto

OMI es un lenguaje orientado a objetos, por tanto permite hacer uso de los conceptos y recursos enmarcados dentro de este paradigma de programación.

Una clase es una construcción del lenguaje que modela un concepto del dominio del problema. Se trata de la definición de un conjunto de entidades u objetos con características y propiedades comunes. Describe los datos necesarios para representarlos y el comportamiento que tienen asociado.

Los objetos son casos concretos de los conceptos modelados mediante las clases. Guardan un estado dando valores a los datos descritos. Además presentan el comportamiento ligado al concepto.

Los datos que representan el estado de un objeto son modelados mediante atributos, mientras que el comportamiento es definido mediante métodos.

En OMI una clase se construye mediante la palabra reservada "*class*", un identificador que la nombra y un bloque de sentencias en la que se definirán los atributos y métodos. Un atributo se define de forma similar a una variable, y un método a una función.

```
/*
   Se define una clase gato. Un gato
   se representa mediante un nombre y tiene
   asociado el comportamiento de aullar.
*/
class gato {
   nombre;
   ~ aullar () {
      << "Miauuu";
   }
}
```

Los objetos pueden ser creados mediante la instanciación de una clase. El objeto dará valores a los atributos y se le podrá aplicar los métodos definidos. Al crear una instancia se obtendrá un objeto llamando al método constructor, que es aquel que tenga el mismo nombre que la clase. Para crear una instancia se utiliza el operador "*new*" seguido del nombre de la clase y los parámetros del método constructor.

```
/*
```

```
2       Se define una clase gato. Un gato
3       se representa mediante un nombre y tiene
4       asociado el comportamiento de aullar.
5    */
6    class gato {
7       nombre;
8       ~ gato (nombre) {
9          << "Construyendo el gato ".nombre;
10      }
11      ~ aullar () {
12         << "Miauuu";
13      }
14   }
15   /*
16      Se instancia un gato
17      llamado "MacAlistair"
18   */
19   mac = new gato ("MacAlistair"); // Imprime "Construyendo el gato MacAlistair"
```

Si se instancia una clase que no se encuentra definida se produce un error semántico y se continua con la ejecución del programa.

Es posible acceder a los atributos y métodos de un objeto mediante el operador "$->$". También se puede hacer referencia al objeto en ejecución dentro de los métodos, para ello se utiliza el operador "*this*".

```
1    /*
2       Se define una clase gato. Un gato
3       se representa mediante un nombre y tiene
4       asociado el comportamiento de aullar.
5 */
6    class gato {
7       nombre;
8       ~ gato (nombre) {
9          this->nombre = nombre;
10      }
11      ~ aullar () {
12         << "Miauuu";
13      }
14   }
15   /*
16      Se instancia un gato
17      llamado "MacAlistair"
18   */
19   mac = new gato ("MacAlistair");
20   mac->aullar(); // Imprime "Miauuu"
21   << mac->nombre; // Imprime "MacAlistair"
```

Si se accede a un atributo o método no existente se llaman al método "_get" o "_call" respectivamente (ver Métodos mágicos **??**). Si estos no existen se produce un error semántico y se continua con la ejecución.

Si no existe ningún método llamado igual que la clase, se considera que esta tiene un constructor vacío. Este es un método con el bloque de sentencias vacío y sin parámetros.

Una clase solo puede disponer de un constructor, si se dan varios métodos llamados igual que la clase solo tendrá efecto el último definido.

4.4.13.1 Accesibilidad de métodos y atributos

Los métodos y atributos definidos en un clase pueden tener restricciones a la accesibilidad de los mismos. Estos pueden ser públicos o privados.

Los atributos y métodos públicos pueden ser accedidos desde un ámbito externo a la clase. A no ser que se especifique lo contrario todos los métodos y atributos de una clase son públicos.

```
1  /*
2     Se define una clase gato. Un gato
3     se representa mediante un nombre y tiene
4     asociado el comportamiento de aullar.
5  */
6  class gato {
7     nombre; // Atributo público
8     ~ gato (nombre) { // Método público
9        this->nombre = nombre; // Acceso a atributo público dentro de la clase
10    }
11    ~ aullar () { // Método público
12       << "Miauuu";
13    }
14 }
15 /*
16    Se instancia un gato
17    llamado "MacAlistair"
18 */
19 mac = new gato ("MacAlistair");
20 mac->aullar(); // Llamada a método público fuera de la clase
21 << mac->nombre;  // Acceso a atributo público fuera de la clase
```

Los métodos o atributos privados no son accesibles desde fuera de la propia clase, solo desde los métodos definido dentro de esta. Para declarar un atributo o método como privado se antepone la palabra clave *private* a la definición.

```
1  /*
2     Se define una clase gato. Un gato
3     se representa mediante un nombre y tiene
4     asociado el comportamiento de aullar.
5  */
6  class gato {
7     private nombre; // Atributo privado
8     ~ gato (nombre) { // Método público
9        this->nombre = nombre; // Acceso a atributo privado dentro de la clase
10    }
11    ~ aullar () { // Método público
12       << "Miauuu";
13    }
14 }
15 /*
16    Se instancia un gato
17    llamado "MacAlistair"
18 */
19 mac = new gato ("MacAlistair");
20 mac->aullar(); // Llamada a método público fuera de la clase
21 << mac->nombre;  // Acceso a atributo privado fuera de la clase (Error)
```

Si el constructor de una clase es privado solo se podrán crear instancias desde los

métodos de esta, los cuales normalmente serán métodos estáticos.

4.4.13.2 Atributos y métodos estáticos

En OMI es posible declarar un método o atributo como estático, esto hará que pertenezca a la clase y no a los objetos instanciados. Los atributos y métodos estáticos son accesibles utilizando la clase y no utilizando un objeto. Generalmente guardan datos o definen un comportamiento aplicables a un conjunto de objetos definido por la clase.

Para declarar un atributo o método como estático se utiliza la palabra reservada "*static*". Para acceder a un método o atributo se utiliza el nombre de la clase, el operador de resolución de nombre de dominio "::" y el nombre del atributo o método. Es posible especificar que un atributo o método será a la vez estático y privado.

```
1   class gato {
2      private nombre;
3      private raza;
4      private energia = 100;
5      private hambre = 100;
6
7      static private energia_min = -10;
8      static private hambre_min = -10;
9
10     static ~ pasar_tiempo (gatos, h = 1) {
11        if (h > 0) {
12           $(gatos) {
13              $->energia -= 10;
14              $->hambre -= (30 * h);
15              if ($->energia < gato::energia_min  $->hambre < gato::hambre_min)
16                 << $->nombre." a pasado a un lugar mejor";
17           }
18        }else{
19           << "Imposible";
20        }
21     }
22
23     ~ gato (nombre, raza) {
24       this->nombre = nombre;
25       this->raza = raza;
26     }
27
28     ~ getNombre () {
29        return this->nombre;
30     }
31
32     ~ getRaza () {
33        return this->raza;
34     }
35
36     ~ aullar () {
37        << "Miauuu";
38     }
39
40     ~ comer () {
41        this->hambre += 10;
42        << "El animal come";
43     }
44
45     ~ dormir () {
```

```
46          this->energia += 10;
47          << "El animal duerme";
48      }
49
50      ~ jugar () {
51          if (this->energia <= 0 or this->hambre <= 0)
52              << "El gato ".this->nombre." no quiere jugar.";
53          else{
54              << "A ".this->nombre." le encanta jugar";
55              this->energia -= 10;
56              this->hambre -= 10;
57          }
58      }
59
60  }
61
62  mac = new gato ("MacAlistair", "Siamés");
63  ada = new gato ("Ada", "Persa");
64  bab = new gato ("Babbage", "Persa");
65
66  $(15) {
67      << "Gatos disponibles";
68      << "[0]=> ".mac->getNombre()." [".mac->getRaza()."] ";
69      << "[1]=> ".ada->getNombre()." [".ada->getRaza()."] ";
70      << "[2]=> ".bab->getNombre()." [".bab->getRaza()."] ";
71      << "[otro]=> Salir ";
72      >>["Seleccione un gato para interactuar:"] s;
73      switch (s) {
74          case 0:
75              g = mac;
76              break;
77          case 1:
78              g = ada;
79              break;
80          case 2:
81              g = bab;
82              break;
83          default:
84              exit;
85      }
86      << "-----------------------------------------------------------------";
87      << "Acciones disponibles";
88      << "[0]=> Dormir ";
89      << "[1]=> Comer ";
90      << "[2]=> Jugar ";
91      << "[otro]=> Aullar ";
92      >>["Seleccione una acción:"] s;
93      switch (s) {
94          case 0:
95              g->dormir();
96              break;
97          case 1:
98              g->comer();
99              break;
100         case 2:
101             g->jugar();
102             break;
103         default:
104             g->aullar();
105     }
106     gato::pasar_tiempo ({mac, ada, bab});
107 }
```

4.4.13.3 Herencia de clases

En OMI es posible definir una relación de herencia entre dos clases. Que una clase herede de otra significa que es una especialización de esta.

La herencia permite establecer una jerarquía entre clases, tal que la clase que especializa es denominada hija de la clase que más genérica, denominada padre. La clase hija deriva de la padre y extiende su funcionalidad y definición. Una clase que extiende a otra toma todos sus métodos y atributos, pudiendo añadir nuevos o redefinir los existentes.

La herencia de clases permite que se dé la propiedad de polimorfismo, de forma que es posible mandar mensajes sintácticamente iguales a objetos de tipos distintos.

Para definir una relación de herencia entre dos clases se se sigue el nombre de la clase hija de la palabra reservada "*extends*" y del nombre de clase a la que extiende.

```
1   class mascota {
2      private nombre;
3      private raza;
4
5      ~ mascota (nombre, raza) {
6        this->nombre = nombre;
7        this->raza = raza;
8      }
9
10     ~ getNombre () {
11       return this->nombre;
12     }
13
14     ~ getRaza () {
15       return this->raza;
16     }
17
18  }
19
20  class gato extends mascota {
21      ~ aullar () {
22        << "Miauuu";
23     }
24  }
25
26  class perro extends mascota {
27      ~ aullar () {
28        << "Guauuu";
29     }
30  }
31
32  mac = new gato ("MacAlistair", "Siamés");
33  ada = new perro ("Ada", "Bulldog");
34  mac->aullar(); // Imprime "Miauuu"
35  ada->aullar(); // Imprime "Guauuu"
```

Si alguna clase hija no define constructor se toma el constructor de la clase padre si esta tuviera.

Es posible definir una serie de clases de forma que presenten varios niveles de jerarquía.

Desde un método definido en la clase hija es posible hacer referencia a los métodos definidos en la clase padre mediante la palabra reservada "*parent*".

```
class mascota {
   private nombre;
   private raza;

   ~ mascota (nombre, raza) {
     this->nombre = nombre;
     this->raza = raza;
   }

   ~ getNombre () {
      return this->nombre;
   }

   ~ getRaza () {
      return this->raza;
   }

}

class gato extends mascota {
    ~ aullar () {
      << "Miauuu";
   }
}

class gato_persa extends gato {
    ~ gato_persa (nombre) {
      parent->gato (nombre, "Persa"); // Llama al método constructor de la clase padre
    }
}

mac = new gato_persa ("MacAlistair");
mac->aullar(); // Imprime "Miauuu"
<< mac->getRaza(); // Imprime "Persa"
```

Cuando un método de una clase padre utiliza el operador "*this*" se accede al objeto en si, de la misma forma que se haría desde un método de la clase hija.

```
class mascota {
   private nombre;
   private raza;

   ~ mascota (nombre, raza) {
     this->nombre = nombre;
     this->raza = raza;
   }

   ~ getNombre () {
      return this->nombre;
   }

   ~ getRaza () {
      return this->raza;
   }

   ~ saludar () {
      // Accede al método getType definido en la clase hija
      << "Hola soy ".this->nombre.", un ".this->getType()." ".this->raza;
   }

```

```
23  }
24
25  class gato extends mascota {
26      ~ getType () {
27        return "gato";
28      }
29
30      ~ aullar () {
31        << "Miauuu";
32     }
33  }
34
35  class perro extends mascota {
36      ~ getType () {
37        return "perro";
38      }
39
40      ~ aullar () {
41        << "Guauuu";
42     }
43  }
44
45  mac = new gato ("MacAlistair", "Persa");
46  mac->aullar(); // Imprime "Miauuu"
47  mac->saludar(); // Imprime "Hola soy MacAlistair, un gato Persa"
48
49  ada = new perro ("Ada", "Bulldog");
50  ada->aullar(); // Imprime "Guauuu"
51  ada->saludar(); // Imprime "Hola soy Ada, un perro Bulldog"
```

En OMI es posible acceder a un atributo o método estático que es definido en la clase hija, desde un método de la clase padre. Para ello se utiliza la palabra reservada "*static*" seguido del operador "::" y el nombre del atributo o método al que se accederá.

```
1   class mascota {
2      private nombre;
3      private raza;
4
5      ~ mascota (nombre, raza) {
6        this->nombre = nombre;
7        this->raza = raza;
8      }
9
10     ~ getNombre () {
11       return this->nombre;
12     }
13
14     ~ getRaza () {
15       return this->raza;
16     }
17
18     ~ saludar () {
19       // Accede al atributo estático type definido en la clase hija
20       << "Hola soy ".this->nombre.", un ".static::type." ".this->raza;
21     }
22
23  }
24
25  class gato extends mascota {
26      static type = "gato";
27
28      ~ aullar () {
29        << "Miauuu";
30     }
31  }
```

```
32
33    class perro extends mascota {
34       static type = "perro";
35
36        ~ aullar () {
37          << "Guauuu";
38       }
39    }
40
41    mac = new gato ("MacAlistair", "Persa");
42    mac->aullar(); // Imprime "Miauuu"
43    mac->saludar(); // Imprime "Hola soy MacAlistair, un gato Persa"
44
45    ada = new perro ("Ada", "Bulldog");
46    ada->aullar(); // Imprime "Guauuu"
47    ada->saludar(); // Imprime "Hola soy Ada, un perro Bulldog"
```

En OMI no es posible llevar a cabo herencia múltiple. Tampoco dispone de ningún mecanismo con el que se pueda simular tales como traits.

4.4.13.4 Redefinición de clases

En OMI es posible redefinir una clase, de forma que se pueden añadir métodos o sobrescribir los existentes.

Para redefinir una clase basta con definirla nuevamente con el mismo identificador. La nueva clase tomará todos los atributos y métodos anteriores.

```
1     class miClase {
2        ~ meth1 () {
3           << "Método 1";
4        }
5     }
6
7     class miClase {
8        ~ meth2 () {
9           << "Método 2";
10       }
11    }
12    foo = new miClase ();
13    foo->meth1(); // Imprime "método 1"
14    foo->meth2(); // Imprime "método 2"
15    class miClase {
16       ~ meth1 () {
17          << "Redefinición de método 1";
18       }
19    }
20    foo = new miClase ();
21    foo->meth1(); // Imprime "Redefinición de método 1"
22    foo->meth2(); // Imprime "Método 2"
```

La redefinición de clases es un mecanismo que ofrece flexibilidad a la hora de construir clases de objetos, no obstante debe ser utilizado con cuidado ya que hace que la definición de una clase sea dinámica y pueda cambiar en la ejecución pudiendo ocasionar confusión en el código e inestabilidad del programa.

4.4.13.5 Clases bases

En OMI existen una serie de clases denominadas clases bases, a partir de las cuales se definen los tipos de datos básicos. Estas son:

- logicClass
- arithClass
- stringClass
- arrayClass

Es posible redefinir una clase base y modificar su comportamiento, añadiendo o sustituyendo métodos.

```
class stringClass {
   ~ OReq (elems) {
      $(elems)
         if (this == $)
            return true;
      return false;
   }
   ~ concatX () {
      return this .= "X";
   }
}

foo = "ABC"->OReq({"A", "AB", "ABC"}); // foo vale true
foo = "ABCD"->OReq({"A", "AB", "ABC"}); // foo vale false
foo = "ABCD"->concatX(); // foo vale "ABCDX"

str = "ABCD";
<< str->concatX(); // Imprime "ABCDX"
<< str; // Imprime "ABCDX"
```

En una clase base el operador "*this*" obtiene el dato en si, por lo que este será tratado como una variable de este tipo.

Es posible extender una clase base para especializarla mediante otras subclases.

```
class stringA extends stringClass {
   ~ stringA (){
      this = "A";
   }
}
class stringB extends stringClass {
   ~ stringB (){
      this = "B";
   }
}

strA = new stingA ();
strB = new stingB ();
<< strA; // Imprime "A"
```

```
15    << strB; // Imprime "B"
16    << strA . strB; // Imprime "AB"
```

Las clases bases logicClass y arithClass no disponen inicialmente de métodos. Las clases stringClass y arrayClass tienen como métodos los equivalentes a las funciones aplicables a estos tipos de datos.

4.4.13.6 Duck typing

OMI es un lenguaje de tipado dinámico que presenta un estilo duck typing. La validez semántica en el uso de un objeto viene determinada por el conjunto de atributos y métodos de este.

Si en una expresión se accede a un método o atributo de un objeto únicamente se comprobará que el elemento exista, indistintamente de la clase a la que pertenezca el objeto. De esta forma es posible utilizar un objeto independientemente de la clase a la que pertenece, sin que sea necesario un chequeo de tipos.

```
1    class pato {
2       ~ sonar () {
3          << "Cuack";
4       }
5    }
6
7    class persona {
8       ~ sonar () {
9          << "La persona imita el sonido de un pato";
10      }
11   }
12
13
14   ~ sonar (obj) {
15      /*
16       * Sea cual sea la clase del objeto,
17       * si tiene el método sonar
18       * será llamado
19       */
20      obj->sonar();
21   }
22
23   obj1 = new pato ();
24   obj2 = new persona ();
25   sonar (obj1); // Imprime "Cuack"
26   sonar (obj2); // Imprime "La persona imita el sonido de un pato"
```

El estilo de tipado duck typing permite que se de polimorfismo sin herencia de clases, de forma que a objetos de distinto tipos se le puede enviar mensajes sintácticamente iguales.

4.4.13.7 Métodos mágicos

En OMI se definen una serie de métodos mágicos. Estos son métodos que serán llamados en distintas circunstancias, como cuando se utilice al objeto como una cadena o se acceda a un atributo no existente.

Los métodos mágicos comienzan por el signo "_" y son los siguientes:

_str: Se llama cuando se accede a un objeto como una una cadena de caracteres.

_get: Se llama cuando se accede a un atributo que no existe.

_call: Se llama cuando se accede a un método que no existe.

El método "_str" no tendrá parámetros y devolverá la cadena de caracteres que represente el objeto.

```
1   class persona {
2      private nombre;
3      private apellidos;
4
5      ~ persona (nombre, apellidos) {
6         this->nombre = nombre;
7         this->apellidos = apellidos;
8      }
9
10     ~ _str () {
11        return this->apellidos.', '.this->nombre;
12     }
13
14  }
15
16  p = new persona ('Fco. Javier', 'Bohórquez Ogalla');
17  << p; // Imprime "Bohórquez Ogalla, Fco. Javier"
```

El método "_get" tendrá un único parámetro que se corresponderá con el nombre del atributo no existente al que se ha accedido. Y devolverá el valor para dicho atributo.

```
1   class persona {
2      private nombre;
3      private apellidos;
4
5      ~ persona (nombre, apellidos) {
6         this->nombre = nombre;
7         this->apellidos = apellidos;
8      }
9
10     ~ _get (attr) {
11        << "El atributo ".attr." no existe";
12        return "No existe";
13     }
14
15  }
16
17  p = new persona ('Fco. Javier', 'Bohórquez Ogalla');
18  foo = p->test; // Imprime "El atributo test no existe"
19  << foo; // Imprime "No existe"
```

El método "_call" tendrá dos parámetros que se corresponderán con el nombre del método no existente al que se ha accedido y con un array que contendrá los parámetros pasados en la llama. El valor devuelto será el valor de la llamada.

```
class persona {
   private nombre;
   private apellidos;

   ~ persona (nombre, apellidos) {
      this->nombre = nombre;
      this->apellidos = apellidos;
   }

   ~ _call (meth, params) {
      <."El método ".meth." no existe [ ";
      $(params) <.$." ";
      << "]";
      return "No existe";
   }

}

p = new persona ('Fco. Javier', 'Bohórquez Ogalla');
foo = p->test("1", 2, "4"); // Imprime "El método test no existe [ 1 2 4 ]"
<< foo; // Imprime "No existe"
```

4.4.14. Entorno del programa

En OMI es posible acceder al entorno en el que se ejecuta el programa, definido por el sistema operativo y la entrada del programa.

La mayoría de sistemas operativos permiten definir una serie de variables de entorno que pueden ser usadas por los procesos y por el propio sistema operativo. En OMI es posible acceder a estas variables mediante la función *getenv* y una cadena que representa el nombre de la variable a la que se desea acceder.

```
<< getenv ("USER"); // Imprime el nombre de usuario
```

Un programa escrito en OMI puede recibir una serie de parámetros cuando es ejecutado. Los argumentos serán almacenados en una variable denominada *args*. El valor de esta será un array donde el primer elemento es el nombre del script y los siguientes los argumentos dados.

```
// File: args.omi
$(args) << $; // Imprime el nombre del script y los argumentos.
```

```
omi args.omi "ARG1" "ARG2" # Imprime args.omi ARG1 y ARG2
```

4.4.15. Excepciones

Las excepciones son un mecanismo de programación que permite tratar casos no habituales, normalmente de error, durante la ejecución de un programa. Las excepciones permiten separar el código correspondiente al caso de normal o de éxito, del código extraordinario o de error.

El uso de excepciones permite realizar programas robustos y tolerantes a errores, a la vez que se gana claridad en el código.

Para hacer uso del manejo de excepciones se utiliza la sentencia *try...catch...*, esta consiste en la palabra reservada *try* seguida del bloque de sentencias en la que puede darse la excepción. Tras lo cual se utiliza la palabra reservada *catch*, un símbolo variable al que se le va a asignar el valor de la excepción y un bloque de sentencias en la que se va a tratar. Para lanzar la excepción se utilizar la palabra reservada *throw* seguida de una expresión que le da valor.

```
1   class Exc {
2      var = null;
3      ~ Exc (num) {
4         this->var = num;
5      }
6      ~ printError () {
7         << "Error " << this->var;
8      }
9   }
10  try {
11     >>["Dame un valor:"] foo;
12     if ( foo === "" ){
13        throw new Exc (404);
14     }
15     << foo;
16  }catch (exc) {
17     exc->printError ();
18  }
```

A diferencias de otros lenguajes de programación OMI solo permite un bloque catch que será ejecutado sea cual sea la excepción que se produzca. No se comprobará el tipo de dato del valor asociado a la excepción.

4.4.16. Reflexión

OMI es un lenguaje que presenta características reflexivas. Un programa escrito en OMI puede hacer uso del entorno construido durante su ejecución para cambiar su propia estructura y comportamiento.

En OMI es posible utilizar expresiones cuyo valor sea una cadena de caracteres como un identificador para una variable, una función, una clase, un método, etc. Para ello se utiliza el símbolo "@" seguido de la expresión entre llaves.

```
class Foo {
   ~ hello () {
      << "Hola mundo";
   }
}

class_pre = "Fo";
class_post = "o";
var = "object";
method = "hello";
@{var} = new @{class_pre.class_post}();
object->@{method}(); // Imprime "Hola mundo"
```

En OMI también es posible evaluar una cadena de caracteres como si de una sentencia de código fuente se tratase. Para ello se utiliza la sentencia "*eval*".

```
foo = " x = 4 + 3 ";
eval (foo);
<< x; // Imprime 7
```

4.4.17. Introspección de tipos

OMI es un lenguaje en el que es posible llevar a cabo introspección de tipos. Permite examinar, en tiempo de ejecución, el tipo de los objetos creados.

Para obtener la clase de un objeto dado se utiliza la palabra clave "*getclass*" seguida de la expresión, cuyo valor representará el objeto, entre paréntesis. Se devolverá la cadena de caracteres correspondiente al nombre de la clase, o la cadena vacía si la expresión no es un objeto.

```
class Foo {
   ~ hello () {
      << "Hola mundo";
   }
}
foo = new Foo ();
if (getclass(foo) == 'Foo')
   foo->hello(); // Imprime "Hola mundo"
```

4.4.18. Errores

Cuando un programa escrito en OMI es interpretado se pueden dar diferentes tipos de errores. Un error de interpretación puede tener diferentes efectos, pero en cualquier caso se presentará información sobre el error ocurrido. Si el código fuente está contenido en un fichero se indicará el nombre del mismo y la línea de código en la que se ha producido el error.

4.4.18.1 Léxicos

Los errores léxicos son originados durante el procesado del código fuente, cuando existe un carácter o cadena que no es reconocida por el intérprete. En este caso se detendrá el programa no será ejecutado y fallará el proceso de interpretación.

```
1   æ = 10;
2   Error: lexical error, unexpected character (æ)
```

4.4.18.2 Sintácticos

Los errores sintácticos tienen lugar durante el procesado del código fuente, cuando una sentencia o expresión no está construida de forma correcta y no respeta las reglas establecidas por el lenguaje. Si se detecta un error sintáctico al procesarse el código fuente este no será ejecutado y el programa se detendrá.

```
1   [ id );
2   Error: syntax error, unexpected ')', expecting ',' or ']'
```

4.4.18.3 Semánticos

Los errores semánticos se dan durante la ejecución del programa. Estos son dependientes del entorno y los datos.

Cuando una sentencia o expresión es interpretada es posible que se dé un error semántico si no es capaz de operar sobre los datos facilitados, o no se da el contexto adecuado.

Existe gran variedad de posibles errores semánticos, estos son descritos en este manual al tratar las sentencias, expresiones y demás construcciones en las que se pueden dar. Cuando se produce un error semántico puede detenerse o no la ejecución del programa, dependerá del error en si. Si no se indica lo contrario en el manual, cuando se produce un error semántico, se prosigue con la ejecución del programa.

```
1   a = new noExist ()
2   Error: data type error, worng class identifier
```

OMI es un lenguaje de tipado dinámico, por lo que cualquier error de tipo se da en tiempo de ejecución. Las funciones y operadores del lenguaje esperan tratar con unos tipos de datos concretos. Si una función u operador recibe un valor de un tipo que no pueda manipular producirá un error semántico. En este manual se indica los tipos de datos

con los que esperan operar las funciones y métodos, se entiende que otro tipo distinto producirá un error.

4.5. Funciones del lenguaje

4.5.1. Cadenas de caracteres

OMI pone a disposición del programador un conjunto de funciones que operan sobre cadenas de caracteres.

En OMI las cadenas de caracteres pertenecen a una clase de objetos, así toda función aplicable a una cadena puede ser llamada como un método de la cadena.

Toda función aplicable a una cadena comienza con el prefijo "str" y toman como primer parámetro la cadena sobre la que operar. Por otro lado los métodos de cadena no disponen de prefijo y se toma implícitamente como primer parámetro el objeto cadena.

4.5.1.1 str_explode

La función str_explode divide en partes una cadena de caracteres según una subcadena denominada separador. El resultado será un array de elementos que contendrá las cadenas correspondientes a cada una de las partes.

array **str_explode** (*string* **str**, *string* **delimeter**)

Parámetros:

str: Cadena de caracteres de entrada.

delimeter: Cadena delimitadora.

Valores devueltos:

Array de elementos que contiene las subcadenas resultado de dividir **str** usando el delimitador **delimeter**.

Si el separador no se encuentra en la cadena el resultado será un array de un elemento que contendrá la cadena completa.

```
1  // foo es el array {"A", "B", "C", "D", "E"}
2  foo = str_explode ("A,B,C,D,E", ",");
3
4  // foo es el array {"A", "B", "C", "D", "E"}
```

```
5   foo = "A,B,C,D,E"->explode (",");
```

4.5.1.2 str_find

La función str_find obtiene la primera posición de una subcadena dentro de una cadena de caracteres.

numeric **str_find** (*string* **str**, *mixed* **substr** [, *numeric* **position**])

Parámetros:

str: Cadena de caracteres de entrada.

substr: Expresión de búsqueda. Puede ser una cadena de caracteres o una expresión regular.

position: Posición dentro de la cadena **str** a partir de la cual se comenzará la búsqueda. Si no es dado se comenzará desde la primera posición de la cadena.

Valores devueltos:

Devuelve el número entero correspondiente a la primera posición en la que se encuentre **substr** dentro de **str**. Se ha de tener en cuenta que la primera posición de la cadena se corresponde con el entero 0.

Si **substr** es una expresión regular esta se corresponderá con la subcadena de mayor longitud perteneciente al lenguaje definido por esta.

Si la subcadena no se encuentra devuelve el valor -1.

Si la posición de inicio **position** es mayor que la longitud de la cadena **str** se devuelve el valor -1.

```
1   foo = str_find ("A,B,C,D,C,F", "C"); // foo vale 4
2   foo = "A,B,C,D,C,F"->find ("C"); // foo vale 4
3
4   foo = str_find ("A,B,C,D,C,F", "C", 5); // foo vale 8
5   foo = "A,B,C,D,C,F"->find ("C", 5); // foo vale 8
6
7   foo = str_find ("A,B,C,D,C,F", `C(.*)C`); // foo vale 4
8   foo = "A,B,C,D,C,F"->find (`C(.*)C`); // foo vale 4
```

4.5.1.3 str_replace

La función str_replace busca las ocurrencias de una subcadena dada en una cadena de caracteres, sustituyéndolas por una cadena de remplazo. El valor devuelto por la función

es la cadena con los remplazos efectuados.

string **str_replace** (*string* **str**, *mixed* **search**, *string* **replace** [, *numeric* **n**])

Parámetros:

str: Cadena de caracteres de entrada.

search: Expresión de búsqueda. Puede ser una cadena de caracteres o una expresión regular.

replace: Cadena de remplazo.

n: Número de reemplazos a efectuar. Si no es dado se realizan todos los reemplazos.

Valores devueltos:

Devuelve un cadena de caracteres resultado de remplazar la subcadena **search** por **replace** en la cadena de caracteres **str**, tantas veces como indique el parámetro **n**.

Si **search** es una expresión regular se buscarán subcadenas que pertenezcan al conjunto de palabras del lenguaje definido por esta. La expresión regular puede estar compuesta por subexpresiones delimitadas por paréntesis, si es así es posible hacer referencia a las subcadenas correspondientes a estas usando el carácter "\" seguido de la posición que ocupa la subexpresión.

```
1  // foo vale "A,B,P,D,P,F"
2  foo = str_replace ("A,B,C,D,C,F", "C", "P");
3
4  // foo vale "A,B,P,D,P,F"
5  foo = "A,B,C,D,C,F"->replace ("C", "P");
6
7  // foo vale "A,B,P,D,C,F"
8  foo = str_replace ("A,B,C,D,C,F", "C", "P", 1);
9
10 // foo vale "A,B,P,D,C,F"
11 foo = "A,B,C,D,C,F"->replace ("C", "P", 1);
12
13 // foo vale "A,B,D,F"
14 foo = str_replace ("A,B,C,D,C,F", 'C,(.)', "\1");
15
16 // foo vale "A,B,D,F"
17 foo = "A,B,C,D,C,F"->replace ('C,(.)', "\1");
```

4.5.1.4 str_replace_sub

La función str_replace_sub remplaza la subcadena dada por una posición inicial y una longitud dentro de un cadena de caracteres por otra cadena.

string **str_replace_sub** (*string* **str**, *numeric* **ini**, *numeric* **len**, *string* **replace**)

Parámetros:

str: Cadena de caracteres de entrada.

ini: Posición inicial para el remplazo.

len: Longitud de la subcadena de remplazo.

replace: Cadena de remplazo.

Valores devueltos:

Devuelve un cadena de caracteres resultado de remplazar la subcadena dada por la posición **ini** y la longitud **len** dentro de la cadena de caracteres **str** por la cadena **replace**.

Si la posición inicial dada se encuentra fuera de la cadena se produce un error semántico.

```
1   // foo vale "A,B,P,C,F"
2   foo = str_replace_sub ("A,B,C,D,C,F", 4, 3, "P");
3
4   // foo vale "A,B,P,C,F"
5   foo = "A,B,C,D,C,F"->replace_sub (4, 3, "P");
```

4.5.1.5 str_upper

La función str_upper convierte todos los caracteres alfabéticos de una cadena en mayúsculas.

string **str_upper** (*string* **str**)

Parámetros:

str: Cadena de caracteres de entrada.

Valores devueltos:

Devuelve la cadena de caracteres **str** con todos los caracteres alfabéticos convertidos a mayúsculas.

```
1   foo = str_upper ("a,b,c,D"); // foo vale "A,B,C,D"
2   foo = "a,b,c,D"->upper (); // foo vale "A,B,C,D"
```

4.5.1.6 str_lower

La función str_lower convierte todos los caracteres alfabéticos de una cadena en minúsculas.

string **str_lower** (*string* **str**)

Parámetros:

str: Cadena de caracteres de entrada.

Valores devueltos:

Devuelve la cadena de caracteres **str** con todos los caracteres alfabéticos convertidos a minúsculas.

```
1   foo = str_lower ("A,B,C,d"); // foo vale "a,b,c,d"
2   foo = "A,B,C,d"->lower (); // foo vale "a,b,c,d"
```

4.5.1.7 str_search

La función str_search permite llevar a cabo una búsqueda aplicando un patrón sobre una cadena de texto. Se obtendrá un array con los resultados de la búsqueda.

array **str_search** (*mixed* **str**, *regexp* **pattern** [, *string* **keys...**])

Parámetros:

str: Cadena de entrada. Puede ser un array de cadenas.

pattern: Patrón de búsqueda.

keys...: Listado de claves para las coincidencias. Si no se da las claves serán numéricas.

Valores devueltos:

Devuelve un array que contiene todas las coincidencias del patrón dado por la expresión regular **pattern** en la cadena **str**, donde las claves del array vienen dadas por la lista de cadenas **keys...**.

La expresión regular puede estar formada por subexpresiones delimitadas por "()". En dicho caso se buscará en la cadena **str** subcadenas que pertenezcan al

conjunto delimitado por la expresión regular. Por cada subcadena encontrada se creará un array con las correspondencias de cada subexpresión.

Si **str** es un array de cadenas se aplicará el algoritmo de búsqueda de foram iterativa a cada cadena en el mismo.

```
atag = '(?i)<a[^>]*href\s*=\s*\"([^\"]*)\"[^>]*>(.+?)<\s*/\s*a\s*>';
web = "<a href=\"url01\">link01 -> web01< /a>";
web .= "\n<div><a id=\"id\" href=\"url02\" class=\"clase\">link02 -> web02</a></div>";
/*
   links es el siguiente array:
   {
      {
         "url" : "url01",
         "label" : "link01 -> web01"
      },
      {
         "url" : "url02",
         "label" : "link02 -> web02"
      }
   }
*/
links = str_search (web, atag, "url", "label");
```

Las cadenas de caracteres tienen el método search, que opera de igual forma que la función str_search, salvo por que no acepta la lista de claves.

4.5.1.8 str_match

La función str_match comprueba si una cadena de caracteres pertenece o no al lenguaje definido por una expresión regular.

bool **str_match** (*string* **str**, *regexp* **pattern**)

Parámetros:

str: Cadena de entrada.

pattern: Expresión regular patrón.

Valores devueltos:

Devuelve un valor booleano verdadero si la cadena de caracteres **str** pertenence al conjunto de palabras del lenguaje definido por la expresión regular **pattern**. Falso en caso contrario.

```
foo = str_match ("ABCD",'A[^D]*D'); // foo vale true
foo = "ABCD"->match('A[^D]*D'); // foo vale true
```

4.5.1.9 regexp

La función regexp convierte una cadena de caracteres dada en una expresión regular

regexp **regexp** (*string* **str**)

Parámetros:

str: Cadena de entrada.

Valores devueltos:

Devuelve la expresión regular correspondiente a la cadena **str**.

```
1   pattern = regexp ("A[^D]*D");
2   foo = "ABCD"->match(pattern); // foo vale true
```

4.5.1.10 sprintf

La función sprintf permite obtener una cadena de caracteres formateada a partir de una serie de valores.

string **sprintf** (*string* **format** [, *mixed* **values...**])

Parámetros:

format:

La cadena formato contendrá una serie de directivas de formato. Estas directivas serán sustituidas por el valor correspondiente, según posición, de la lista. Cuando se realiza cada sustitución el valor es formateado según la directiva.

Las directivas de formato tienen el siguiente forma:

$$\%[operador][precisión][formato]$$

Los posibles operadores serán los siguientes:

+: Fuerza la impresión del símbolo + cuando se formatean números positivos.

^ : Convierte el caracteres a mayúsculas cuando se formatean cadenas de texto.

#: Añade el carácter 0x cuando se formatean números hexadecimales y el carácter 0 cuando se formatean octales.

La precisión se refiere al número de decimales que se imprimirán en el caso de formatear números o el número de caracteres en el caso de formatear cadenas.

El carácter de formato indica que tipo de formato se le dará al valor:

i|d: Número entero.

u: Sin signo.

f: Coma flotante.

%: Carácter %.

e: Notación científica.

o: Octal.

x: Hexadecimal.

s|c: Cadena de texto.

values:

Lista de valores. Se deben de dar tantos como directivas existan en la cadena de formato.

Valores devueltos:

Devuelve la cadena de caracteres resultante de sustituir las directivas de formato de la cadena **format**, por los valores de la lista **values** formateados según la directiva correspondiente posicionalmente.

```
<< sprintf("Esto es una %s formateada.", "cadena"); // Esto es una cadena formateada.
<< sprintf("Ya son %2d", 2.003); // Ya son 02
<< sprintf("Y %4s", "tres gatos"); // Y tres
<< sprintf("Y %+4i", "4"); // Y +0004
<< sprintf("Y %u", -5); // Y 5
<< sprintf("Y %+2f", 6.6666 ); // Y +6.7
<< sprintf("Y %+5e", 777.77777777); // Y +7.77778e+02
<< sprintf("Y %F", "88888888888888888888888888"); // Y 8.88889E+25
<< sprintf("Y %3E", "99.99"); // Y 9.999E+01
<< sprintf("Y %^s", "diez"); // Y DIEZ
<< sprintf("%d en octal es %#o", 31, 31); // 31 en octal es 037
// 31 en hexadecimal es 0x1f o 0X1F
<< sprintf("%d en hexadecimal es %#x o %#X", 31, 31, 31);
<< sprintf("Texto sin expresiones de formato"); // Texto sin expresiones de formato
<< sprintf("%2sdenas %3stas", "Cadiz", "junio"); // Cadenas juntas
// El 100% de aciertos en 100 intento
<< sprintf("El %s%% de aciertos en %d intentos", 100, 100);
<< sprintf("%%%%%%"); // %%%%
```

4.5.2. Arrays

OMI pone a disposición del programador un conjunto de funciones que operan sobre arrays.

En OMI los arrays pertenecen a una clase de objetos, así toda función aplicable a un array puede ser llamada como un método del mismo.

Toda función aplicable a un array con el prefijo "array" y toman como primer parámetro el array sobre el que operar. Por otro lado los métodos de array no disponen de prefijo y se toma implícitamente como primer parámetro el objeto array.

4.5.2.1 array_implode

Forma una cadena de caracteres a partir de un array de cadenas y una cadena separadora.

string **array_implode** (*array* **elements**, *string* **sep**)

Parámetros:

elements: Array de elementos

sep: Cadena separadora

Valores devueltos:

Devuelve un string correspondiente a todos los elementos del array **elements** en el mismo orden, con el string **sep** entre cada elemento.

```
1   << array_implode ({"Hola", "Mundo"}, " "); // Imprime "Hola Mundo"
2   << {"Hola", "Mundo"}->implode (" "); // Imprime "Hola Mundo"
```

4.5.2.2 array_first

Obtiene el primer elemento de un array.

mixed **array_first** (*array* **elements**)

Parámetros:

elements: Array de elementos

Valores devueltos:

Devuelve el primer elemento del array **elements**. Si **elements** es un array numérico con claves numéricas devuelve el elemento con la posición 0. Si es un array asociativo devuelve el elemento correspondiente a la primera clave que se introdujo. Si el array es vacío devuelve un valor nulo.

```
1   << array_first ({"Hola", "Mundo"}); // Imprime "Hola"
2   << {"Hola", "Mundo"}->first (); // Imprime "Hola"
```

4.5.2.3 array_last

Obtiene el último elemento de un array.

mixed **array_last** (*array* **elements**)

Parámetros:

elements: Array de elementos

Valores devueltos:

Devuelve el último elemento del array **elements**. Si **elements** es un array numérico con claves numéricas devuelve el elemento con la posición $size - 1$. Si es un array asociativo devuelve el elemento correspondiente a la última clave que se introdujo. Si el array es vacío devuelve un valor nulo.

```
1   << array_last ({"Hola", "Mundo"}); // Imprime "Mundo"
2   << {"Hola", "Mundo"}->last (); // Imprime "Mundo"
```

4.5.2.4 array_insert

Inserta un elemento en una determinada posición de un array dado.

array **array_insert** (*array* &**elements**, *numeric* **position**, *mixed* **element**)

Parámetros:

elements: Array de elementos. Es una referencia por lo que el array dado como valor de este parámetro será modificado.

position: Entero sin signo que determina la posición en la que se insertará el elemento. Debe ser un valor entero entre 0 y el tamaño del array, cualquier otro valor dará error de índice.

element: Elemento a insertar

Valores devueltos:

Devuelve el array resultado de insertar **element** en la posición **position** del array **elements**. Si **elements** es un array de índices numéricos inserta el elemento en la posición dada. Si es un array asociativo inserta el elemento en la clave que ocupa la posición dada, desplazando los demás valores y creando una nueva clave correspondiente al número de elementos.

```
1  /*
2   * foo vale el array { "Hola", "Mundo", "Digital"}
3   */
4  foo = array_insert ({"Hola", "Mundo"}, 2, "Digital");
5  /*
6   * foo vale el array { "Hola", "Mundo", "Digital"}
7   */
8  foo = {"Hola", "Mundo"}->insert (2, "Digital");
9
10 foo = {"A", "C"};
11 array_insert (foo, 1, "B"); // foo vale {"A", "B", "C"}
12 foo->insert (3, "D"); // foo vale {"A", "B", "C", "D"}
```

4.5.2.5 array_delete

Elimina el elemento ocupa una determinada posición en un array dado.

array **array_delete** (*array* &**elements**, *mixed* **position**)

Parámetros:

elements: Array de elementos. Es una referencia por lo que el array dado como valor de este parámetro será modificado.

position: Puede ser un entero sin signo que determina la posición que será eliminada. Debe ser un valor entero mayor que 0 y menor que el tamaño del array, cualquier otro valor no tendrá efecto. Además puede ser una cadena de caracteres que represente la clave del array que será eliminada. Si la clave no existe en el array la función no realizará ninguna acción.

Valores devueltos:

Si el array **elements** presenta índices numéricos devuelve el array resultado de eliminar la posición **position** del array.

Si el array **elements** es asociativo devuelve el array resultado de eliminar la clave **position** del array.

```
/*
 * foo vale el array { "Hola" }
 */
foo = array_delete ({"Hola", "Mundo"}, 1);
/*
 * foo vale el array { "Hola" }
 */
foo = {"Hola", "Mundo"}->delete (1);

foo = {"A", "C"};
array_delete (foo, 0); // foo vale {"C"}
foo->delete (0); // foo vale {}

foo = { 'key0' : 'val0', 'key1' : 'val1' };
foo->delete ('key0'); // foo vale { 'key1' : 'val1' }
```

4.5.2.6 array_push

Inserta un elemento al final de un array dado.

array **array_push** (*array* &**elements**, *mixed* **element**)

Parámetros:

elements: Array de elementos. Es una referencia por lo que el array dado como valor de este parámetro será modificado.

element: Elemento a insertar al final del array

Valores devueltos:

Devuelve el array **elements** con el elemento **element** insertado en la última posición. Si **elements** es asociativo la clave del elemento insertado será el número de elementos del array antes de la insercción.

```
/*
 * foo vale el array { "Hola", "Mundo", "Digital" }
 */
foo = array_push ({"Hola", "Mundo"}, "Digital");
/*
 * foo vale el array { "Hola", "Mundo", "Digital"}
 */
```

```
8   foo = {"Hola", "Mundo"}->push ("Digital");
9
10  foo = {"A", "B"};
11  array_push (foo, "C"); // foo vale { "A", "B", "C" }
12  foo->push ("D"); // foo vale { "A", "B", "C", "D" }
13
14  foo = { 'key0' : 'val0', 'key1' : 'val1'};
15  foo->push ('val2'); // foo vale { 'key0' : 'val0', 'key1' : 'val1', '2' : 'val2' }
```

4.5.2.7 array_pop

Elimina y devuelve el último elemento de un array dado.

mixed **array_pop** (*array* &**elements**)

Parámetros:

elements: Array de elementos. Es una referencia por lo que el array dado como valor de este parámetro será modificado.

Valores devueltos:

Devuelve el último elemento del array **elements** y lo elimina del mismo. Si el array es asociativo devuelve el elemento de la última clave introducida.

```
1   /*
2    * foo vale la cadena "Mundo"
3    */
4   foo = array_pop ({"Hola", "Mundo"});
5   /*
6    * foo vale la cadena "Mundo"
7    */
8   foo = {"Hola", "Mundo"}->pop ();
9
10  foo = {"A", "B"};
11  array_pop (foo); // foo vale { "A" }
12  foo->pop (); // foo vale { }
13
14  foo = { 'key0' : 'val0', 'key1' : 'val1'};
15  foo->pop (); // foo vale { 'key0' : 'val0' }
```

4.5.2.8 array_unshift

Inserta un elemento al inicio de un array dado.

array **array_unshift** (*array* &**elements**, *mixed* **element**)

Parámetros:

elements: Array de elementos. Es una referencia por lo que el array dado como valor de este parámetro será modificado.

element: Elemento a insertar al inicio del array

Valores devueltos:

Devuelve el array **elements** con el elemento **element** insertado en la primera posición. Si **elements** es asociativo la clave del elemento insertado será la misma que la anterior, llevándose a cabo un desplazamiento.

```
/*
 * foo vale el array { "Digital", "Hola", "Mundo" }
 */
foo = array_unshift ({ "Hola", "Mundo" }, "Digital");
/*
 * foo vale el array { "Digital", "Hola", "Mundo"}
 */
foo = {"Hola", "Mundo"}->unshift ("Digital");

foo = {"B", "C"};
array_unshift (foo, "A"); // foo vale { "A", "B", "C" }
foo->unshift ("-"); // foo vale { "-", "A", "B", "C" }

foo = { 'key0' : 'val0', 'key1' : 'val1'};
foo->unshift ('val2'); // foo vale { 'key0' : 'val2', 'key1' : 'val0', '2' : 'val1' }
```

4.5.2.9 array_shift

Elimina y devuelve el primer elemento de un array dado.

mixed **array_shift** (*array* &**elements**)

Parámetros:

elements: Array de elementos. Es una referencia por lo que el array dado como valor de este parámetro será modificado.

Valores devueltos:

Devuelve el primer elemento del array **elements** y lo elimina del mismo. Si el array es asociativo devuelve el elemento de la primera clave introducida.

```
/*
 * foo vale la cadena "Hola"
 */
foo = array_shift ({"Hola", "Mundo"});
```

```
5   /*
6    * foo vale la cadena "Hola"
7    */
8   foo = {"Hola", "Mundo"}->shift ();
9
10  foo = {"A", "B"};
11  array_shift (foo); // foo vale { "B" }
12  foo->shift (); // foo vale { }
13
14  foo = { 'key0' : 'val0', 'key1' : 'val1'};
15  foo->shift (); // foo vale { 'key1' : 'val1' }
```

4.5.2.10 array_reduce

Lleva a cabo la reducción de un array a un único valor, aplicando iterativamente una función.

mixed **array_reduce** (*array* **elements**, *function* **iterative**)

Parámetros:

elements: Array de elementos.

itereative: Función de iteración para la reducción. La función debe recibir dos parámetros donde el primero será el acumulador y el segundo el elemento de la iteración actual, y devolverá el acumulador para la próxima iteración. La iteración comienza con el primer elemento como valor del acumulador y el segundo como primer elemento para iterar.

Valores devueltos:

Devuelve el resultado de aplicar la función **iterative** iterativamente sobre los elementos del array **elements**.

```
1   /*
2    * foo vale la cadena "Hola Mundo Digital"
3    */
4   foo = array_reduce (
5      {"Hola", "Mundo", "Digital"},
6      ~(acumulador, elemento) {
7         return acumulador." ".elemento;
8      }
9   );
10  /*
11   * foo vale la cadena "Hola Mundo Digital"
12   */
13  foo = {"Hola", "Mundo", "Digital"}->reduce (
14     ~(acumulador, elemento) {
15        return acumulador." ".elemento;
16     }
17  );
```

4.5.3. Procesos

OMI proporciona una serie de funciones para gestionarlos procesos. Así es posible abrir varios hilos de ejecución creando procesos.

Es posible enviar señales a los procesos creados, y estos podrán manejarlas según sea necesario.

4.5.3.1 exec

Ejecuta un comando del sistema y devuelve la salida.

string **exec** (*string* **cmd**)

Parámetros:

cmd: Cadena de caracteres que representa el comando a ejecutar

Valores devueltos:

Devuelve el contenido de la salida estándar al ejecutar el comando del sistema indicado por **cmd**.

```
1  /*
2   * foo vale el listado de procesos en ejecución
3   */
4  foo = exec ('ps -e');
```

4.5.3.2 fork

Crea un proceso hijo del proceso en ejecución.

numeric **fork** ( )

Valores devueltos:

Produce una bifurcación en la ejecución, creando un nuevo proceso copia del proceso actual. La ejecución en el proceso hijo prosigue en la misma sentencia y con el mismo estado que el proceso padre.

El valor devuelto en el proceso padre será de el identificador del proceso hijo creado.

El valor devuelto en el proceso hijo será 0.

```
1  /*
2   * Se lleva a cabo la bifurcación de la ejecución mediante la
3   * creación de un proceso que imprimirá "Soy el proceso Hijo",
4   * mientras que en el proceso principal imprimirá
5   * "Soy el proceso Padre de PID" donde PID es el identificador
6   * del proceso hijo.
7   */
8   if (pid = fork ())
9     << "Soy el proceso Padre de " . pid;
10  else
11    << "Soy el proceso Hijo";
```

4.5.3.3 wait

Espera la finalización de todos o algunos de los procesos hijos creados.

numeric **wait** ([*numeric* pid])

Parámetros:

pid: Identificador del proceso hijo cuya ejecución se desea esperar a que finalice. Si no es dado se esperará a que finalicen todos los procesos hijos.

Valores devueltos:

Devuelve el código de salida producido por el último proceso en finalizar.

```
1  /*
2   * Se crea un proceso y se espera
3   * a que este finalice
4   */
5   if (pid = fork ()) {
6     wait( pid );
7     << "Finaliza proceso padre";
8  }else{
9     sleep (10);
10    << "Finaliza proceso hijo";
11 }
```

4.5.3.4 getpid

Obtiene el identificador del proceso actual.

numeric **getpid** ( )

Valores devueltos:

Devuelve el identificador del proceso actual

```
/*
 * Se crea un proceso y se imprimen
 * los identificadores.
 */
if (pid = fork ()) {
   << "PID del Padre " . getpid (); // Imprime el pid del padre
   << "PID del Hijo " . pid; // Imprime el pid del hijo
} else {
   << "PID del Hijo " . getpid (); // Imprime el pid del hijo
}
```

4.5.3.5 getppid

Obtiene el identificador del proceso padre.

numeric **getppid** ( )

Valores devueltos:

Devuelve el identificador del proceso padre del actual.

```
/*
 * Se crea un proceso y se imprimen
 * los identificadores.
 */
if (pid = fork ()) {
   << "PID del Padre " . getpid (); // Imprime el pid del padre
   << "PID del Hijo " . pid; // Imprime el pid del hijo
} else {
   << "PID del Padre " . getppid (); // Imprime el pid del padre
   << "PID del Hijo " . getpid (); // Imprime el pid del hijo
}
```

4.5.3.6 signal

Envía una señal a un proceso.

bool **signal** (*numeric* **pid**, *numeric* **code**)

Parámetros:

pid: Identificador de proceso al que se le envía la señal

code: Código numérico de señal POSIX

Valores devueltos:

Un valor booleano que indica si la señal se envió correctamente.

```
/*
 * Se crea un proceso y envía una señal pasados
 * unos segundos para finalizar su ejecución
 */
if (pid = fork ()) {
   << "Padre: Esperando para finalizar";
   sleep (2);
   << "Padre: Enviando señal";
   signal (pid, 9);
} else {
   << "Hijo: Esperando señal";
   sleep (30);
   << "Hijo: Finaliza normal";
}
```

4.5.3.7 shandler

Permite establecer una función para manejar señales enviadas al proceso actual.

bool **shandler** (*numeric* **code**, *function* **handler**)

Parámetros:

code: Código numérico de señal POSIX a la que se le atribuirá el manejador.

handler: Función que manejará la ejecución cuando se dé la señal.

Valores devueltos:

Un valor booleano que indica si el manejador de señal se estableció correctamente.

```
/*
 * Se crea un proceso y envía una señal
 * que será manejada
 */
```

```
5   if (pid = fork ()) {
6      << "Padre: Esperando para interrumpir";
7      sleep (2);
8      << "Padre: Enviando interrupción";
9      signal (pid, 2);
10  } else {
11     if (shandler (2, ~(){ << "Interrupción"; })){
12        sleep (30);
13        << "Hijo: Finaliza normal";
14     }else {
15        << "Error estableciendo manejador de señal";
16     }
17  }
```

4.5.3.8 exit_process

Finaliza el proceso en ejecución y todos los hijos de este. Si se da en el proceso principal se sale del programa.

void **exit_process** ( )

```
1   /*
2    * Se crea un proceso y si el identificador de
3    * proceso es menor que 20000 se cierra.
4    */
5   if (pid = fork ()) {
6      sleep (30);
7   } else {
8      if (getpid() < 20000){
9         exit_process ();
10     }
11     << "Exit";
12  }
```

4.5.3.9 process

Realiza una llamada a función como un proceso.

bool **process** (*function* **func** [, *mixed* **args** **...**])

Parámetros:

func: Función que será llamada como un proceso.

args...: Lista de parámetros que serán pasados a la función.

Valores devueltos:

Devuelve un valor verdadero si el proceso fue llamado con éxito.

```
1   process ( ~() { $(100) << "P1:".$; } );
2   process ( ~(i) { $(i) << "P2:".$; }, 100 );
```

4.5.4. Ficheros

OMI dispone de una serie de funciones para gestionar ficheros. Estas funciones permiten abrir un fichero, leer su contenido, escribir en el mismo, etc.

OMI presenta un tipo de dato especial que se corresponde con un puntero a fichero. Este tipo de dato sólo puede ser usado en las funciones de fichero, en cualquier otro contexto tomará un valor booleano que indica si el fichero se encuentra abierto.

Un puntero a fichero referencia una posición dentro del mismo, posición a partir de la cual se llevarán a cabo las operaciones de lectura/escritura. Esta posición puede ser manipulada.

OMI solo es capaz de operar directamente con ficheros en texto plano. No es posible crear ficheros de tipo binario que no sea texto plano.

4.5.4.1 file

Abre un fichero según el modo especificado.

file **file** (*string* **name**, *string* **mode**)

Parámetros:

name: Nombre del fichero que se desea abrir. Se puede utilizar una ruta absoluta o relativa al programa

mode: Modo en el que se abrirá el fichero:

r: Solo lectura

r+: Lectura y/o escritura

w: Escritura truncando el fichero

w+: Lectura y/o escritura truncando el fichero

a: Escritura posicionando el puntero al final del fichero

a+: Lectura y/o escritura posicionando el puntero al final del fichero

Todos los modos excepto el solo lectura crean el fichero si este no existe.

Valores devueltos:

Devuelve un puntero a fichero si el fichero ha sido abierto correctamente. Y un valor falso si no se ha abierto el fichero con éxito

```
1   if (f = file ("example.file", "w+")){
2      << "Fichero abierto con éxito";
3   } else {
4      << "Error abriendo el fichero";
5   }
```

4.5.4.2 fclose

Cierra un fichero abierto mediante la función "file"

bool **fclose** (*file* **fpointer**)

Parámetros:

fpointer: Puntero a fichero.

Valores devueltos:

Devuelve un valor booleano que indica si el fichero se ha cerrado correctamente.

```
1   if (f = file ("example.file", "w+")){
2      fclose (f);
3   } else {
4      << "Error abriendo el fichero";
5   }
```

4.5.4.3 fput

Escribe un contenido dado en la posición de lectura/escritura a la que apunta un puntero a fichero. Esta función presenta un alias "f>:".

numeric **fput|f>:** (*file* **fpointer**, *string* **content**)

Parámetros:

fpointer: Puntero a fichero.

content: Cadena de caracteres que será escrita en el fichero a partir de la posición en la que se encuentre el puntero.

Valores devueltos:
Devuelve un valor numérico que representa los bytes que han sido escritos.

```
if (f = file ("example.file", "w+")){
   f>: (f, "Contenido a escribir");
   fclose(f);
} else {
   << "Error abriendo el fichero";
}
```

4.5.4.4 fget

Lee el un número de caracteres de un fichero, a partir de la posición de lectura/escritura dada por el puntero al mismo. Esta función presenta un alias "f<:".

string **fget**|**f<:** (*file* **fpointer** [, *numeric* **size**])

Parámetros:

fpointer: Puntero a fichero.

size: Número de caracteres que serán leídos. Si no es dado se lee hasta el primer carácter de salto de línea encontrado o hasta el final del fichero

Valores devueltos:
Devuelve la cadena de caracteres leída.

```
if (f = file ("example.file", "w")){
   f>: (f, "Contenido escrito");
   fclose(f);
} else {
   << "Error abriendo el fichero";
}
if (f = file ("example.file", "r")){
   << f<:(f); // Imprime "Contenido escrito"
   fclose (f);
} else {
   << "Error abriendo el fichero";
}
```

4.5.4.5 fseek

Cambia la posición de lectura/escritura de un puntero a fichero.

bool **fseek** (*file* **fpointer**, *numeric* **offset** [, *enum* **position**])

Parámetros:

fpointer: Puntero a fichero.

offset: Número de bytes que será desplazado el puntero

position: Posición de referencia desde la que mover el puntero. Puede ser algún valor FSET, FCUR o FEND, para referirse al inicio, la posición actual o el final. Si no es dado se toma como referencia el inicio del fichero.

Valores devueltos:

Devuelve un booleano que indica si la posición cambió correctamente. Si la posición nueva se encuentra fuera del fichero el puntero se colocará al final del mismo.

```
1   if (f = file ("example.file", "w+")){
2      f>:(f, "ABCD");
3      fseek (f, 2);
4      << f<:(f, 1); // Imprime "C"
5      fseek (f, -1, FEND);
6      << f<:(f, 1); // Imprime "D"
7      fseek (f, -1, FCUR);
8      << f<:(f, 1); // Imprime "C"
9      fseek (f, -1, FCUR);
10     << f<:(f, 1); // Imprime "B"
11     fseek (f, -1, FCUR);
12     << f<:(f, 1); // Imprime "A"
13     fclose(f);
14  } else {
15     << "Error abriendo el fichero";
16  }
```

4.5.4.6 ftell

Obtiene la posición de lectura/escritura a la que apunta un puntero a fichero.

numeric **ftell** (*file* **fpointer**)

Parámetros:

fpointer: Puntero a fichero.

Valores devueltos:

Devuelve un la posición del puntero al fichero como un número que representa el número de bytes desde el inicio del mismo.

```
1   if (f = file ("example.file", "w+")){
2      f>:(f, "ABCD");
3      fseek (f, 2);
4      << "Antes de leer: " . ftell (f); // Imprime 2
5      << f<:(f, 1); // Imprime "C"
6      << "Después de leer: " . ftell (f); // Imprime 3
7      fclose(f);
8   } else {
9      << "Error abriendo el fichero";
10  }
```

4.5.4.7 fread

Lee el contenido de un fichero.

string **fread** (*mixed* **file**)

Parámetros:

file: Fichero que será leído. Puede ser dado con ruta relativa al programa o absoluta, o como un puntero a fichero. Si el fichero no existe devolverá el valor nulo.

Valores devueltos:

Devuelve una cadena de caracteres que representa el contenido del fichero.

```
1   foo = fread ('/etc/hosts'); // foo vale el contenido del fichero /etc/hosts
```

4.5.4.8 fwrite

Escribe el contenido de un fichero.

string **fwrite** (*mixed* **file**, *string* **content**)

Parámetros:

file: Fichero que será leído. Puede ser dado con ruta relativa al programa o absoluta, o como un puntero a fichero. Si el fichero no existe será creado.

content: Cadena de caracteres que será el contenido del fichero.

Valores devueltos:

Devuelve la cadena de caracteres escrita en el fichero.

```
1  /*
2  * El fichero /etc/hosts queda
3  * con el contenido:
4  * 127.0.0.1 localhost
5  */
6  fwrite ('/etc/hosts', '127.0.0.1 localhost');
```

4.5.4.9 fappend

Escribe el contenido al final de un fichero.

string **fappend** (*mixed* **file**, *string* **content**)

Parámetros:

file: Fichero que será leído. Puede ser dado con ruta relativa al programa o absoluta, o como un puntero a fichero. Si el fichero no existe será creado.

content: Cadena de caracteres que será añadida al contenido del fichero.

Valores devueltos:

Devuelve la cadena de caracteres escrita en el fichero.

```
1  /*
2  * Al fichero /etc/hosts se le
3  * ha añadido el contenido:
4  * 127.0.0.1 localhost
5  */
6  fappend ('/etc/hosts', '127.0.0.1 localhost');
```

4.5.5. Fechas y tiempo

OMI dispone de un conjunto de funciones de tiempo que permiten operar con fechas y con marcas de tiempo.

4.5.5.1 date

Obtiene una fecha formateada.

string **date** (*mixed* **format** [, *numeric* **timestamp**])

Parámetros:

format: Cadena de formato. Establece el formato de la fecha mediante una serie de directivas.

%d: Día del mes con dos dígitos.
%j: Día del mes sin ceros iniciales.
%l: Día de la semana de forma alfabética completa.
%D: Día de la semana de forma alfabética y con tres letras.
%w: Día de la semana de forma numérica (0-domingo,6-sábado).
%z: Día del año de forma numérica.
%F: Mes de forma alfabética.
%m: Mes de forma numérica con dos dígitos.
%n: Mes de forma numérica sin ceros iniciales.
%M: Mes de forma alfabética con tres letras.
%Y: Año con cuatro dígitos.
%y: Año con dos dígitos.
%a: Periodo del día (am/pm) en minúsculas.
%A: Periodo del día (am/pm) en mayúsculas.
%g: Hora en formato 12h sin ceros iniciales.
%G: Hora en formato 24h sin ceros iniciales.
%h: Hora en formato 12h con dos dígitos.
%H: Hora en formato 24h con dos dígitos.
%i: Minutos con dos dígitos.
%U: Segundos desde la Época Unix (1 de Enero del 1970 00:00:00 GMT).
%%: Carácter %.

timestamp: Número entero que representa la marca de tiempo Unix a la que se desea dar formato. Si no es dado se toma el momento actual.

Valores devueltos:

Devuelve la cadena de caracteres correspondiente a la marca de tiempo **timestamp** formateada según **format**.

```
1   << date ("%d/%m/%Y %H:%i:%s"); // Fecha y hora actual
2   << date ("%d/%m/%Y %H:%i:%s", 0); // 01/01/1970 00:00:00
```

4.5.5.2 time

Obtiene la marca de tiempo Unix actual.

numeric **time** ( )

Valores devueltos:
Devuelve el entero correspondiente a la marca de tiempo Unix actual.

```
<< time (); // Imprime la marca de tiempo Unix actual
<< date ("%d/%m/%Y %H:%i:%s", time()); // Fecha y hora actual
```

Esta es una función interna de OMI que puede ser utilizada sin paréntesis.

4.6. Extensiones del lenguaje

OMI es un lenguaje modular que puede ser extendido. Los módulos o extensiones permiten añadir funciones al lenguaje, ampliando así su funcionalidad y uso.

Los módulos son bibliotecas dinámicas que son cargadas en tiempo de ejecución. Estas han sido programadas en un lenguaje de más bajo nivel (C/C++) y compiladas a código máquina.

Es posible obtener módulos de OMI desde la web del proyecto (http://wwww.omi-project.com/download/extensions).

4.6.1. Carga de módulos

Para cargar un módulo desde el código fuente se puede utilizar la sentencia *load* seguida de una cadena entre paréntesis que representa la biblioteca dinámica que se usará como módulo.

```
load ('libomi_gettext.so');
<< _('cadena traducida');
```

Se dispone de un mecanismo para autocargar módulos de extensiones del lenguaje. Al iniciarse el intérprete se cargarán todos los módulos indicados en el fichero "libs.ini" localizado en el directorio de datos de la aplicación (por defecto /usr/local/share/omi).

Cada línea de este fichero será un módulo que será cargado, exceptuando las líneas que comiencen por ";" que son consideradas comentarios.

4.6.2. Creación de módulos

OMI permite al programador extender el lenguaje haciendo uso de una biblioteca para el desarrollo de módulos. Esta dispone de los elementos sobre los que se construye el intérprete.

Para construir un módulo se ha de disponer de la biblioteca OMI correctamente instalada. Para ello se dispone de dos opciones:

Paquete .deb: http://wwww.omi-project.com/download/deb/libomi-dev_$VERSION.deb

Código fuente: A partir del código fuente de OMI se puede construir e instalar la biblioteca mediante las reglas *dev* e *install-dev*

Un módulo OMI está escrito en C++ y se define mediante unas series de clases cada una de las cuales representará una función que se desea añadir.

El intérprete OMI se construye mediante una colección de nodos ejecutables. Todo es un nodo ejecutable: los booleanos, los números, las variables, las funciones... Existen diferentes niveles de abstracción en los nodos ejecutables, así por ejemplo un número es representado mediante un nodo ejecutable numérico, que a su vez es un nodo expresión aritmética, que a su vez es una expresión de tipo definido... Las clases de un módulo extenderán los tipos de nodos definidos por la biblioteca de desarrollo OMI, normalmente nodos expresiones.

Las clases que integran un modulo OMI deben cumplir lo siguiente:

- Ser un tipo de nodo ejecutable.
- Ser un nodo extensión e inherir su constructor.
- Implementar un método *run* que lleve a cabo la funcionalidad.
- Ser añadidas al cargador de módulos.

Dependiendo el tipo de nodo ejecutable que se tome de base se dispondrá de un atributo que guardará el valor interno del nodo. Así un nodo de tipo cadena de caracteres tendrá un atributo que representa la cadena guardada. El método *run* llevará a cabo la funcionalidad y dará valor a este atributo.

Normalmente el método *run* obtendrá los parámetros con los que se ha llamado a la función, les aplicará la operativa asociada y atribuirá el valor interno del nodo. Mediante una llamada al método *getParams* es posible obtener un vector con los parámetros pasados a la función en una ejecución.

```
// libomi_strings.h
#include <omi/omi.h>

class ucfirstNode : public stringNode, public extensionNode {
   public:
      using extensionNode::extensionNode; // Si std=c++11
      // ucfirstNode (listNode* list) : extensionNode(list) {}  //Si std=c++98
      void run ();
};
```

```
// libomi_strings.cpp
#include "libomi_strings.h"

void ucfirstNode::run () {
   vector<runNode*> v = this->getParams ();
   if (v.size() == 1){
      strvalue_ = stringNode::to_str (v[0]);
      strvalue_[0] = toupper(strvalue_[0]);
   }else
       throw errorException(
         "One parameter is required",
         "ucfirstNode::run, size " + to_string (v.size()),
         ERROR_PARAMS
      );
}

extern "C" void load (PluginsLoader* loader) {
   loader->add ("ucfirst", create<ucfirstNode>);
}
```

```
$PS1 # Compilación del módulo
$PS1 g++ -c -fPIC \
$PS2     -DNUMTYPE='double' -DNUMPRECISION='15' -DREFCTYPE='unsigned int' \
$PS2     -ansi -pedantic -g  -std=c++11  libomi_strings.cpp -o libomi_strings.o
$PS1 g++ -g -O2 -shared -rdynamic -g -o libomi_strings.so libomi_strings.o -lomi
```

```
// user.omi
load ("./libomi_strings.so")
<< ucfirst ("example"); // Imprime "Example"
```