Open Modular Interpreter

Pruebas del sistema

Fco. Javier Bohórquez Ogalla

Índice

1. Visión general

El objetivo de esta sección del documento es presentar el plan de pruebas seguido, incuyendo los diferentes tipos de pruebas que se han realizado.

En primer lugar se describe la estrategia de pruebas seguida. Para ello se describe el alcance de estas y el procedimiento de pruebas de regresión realizado.

Otro aspecto tratado es la descripción del entorno que se deberá utilizar para la realización de las pruebas. Esto incluye los requisitos de software y hardware necesarios.

Además se incluyen los perfiles y participantes necesarios para llevar a cabo los casos de prueba. Esto son los roles desde los que se llevarán a cabo las pruebas.

Por último se documentan las diferente pruebas realizadas según el tipo al que estas pertenecen.

2. Estrategia

Las pruebas llevadas a cabo comprenden toda la funcionalidad del sistema. En cada iteración del ciclo de vida en se realizan pruebas relativas a las funcionalidades comprendidas en la misma.

Para el diseño de las pruebas se ha tomado un enfoque funcional o de caja negra, centrada en la epecificación de las funciones, la entrada y salida. La técnica utilizada consiste en definir los casos de pruebas a partir de clases de equivalencias:

1. Identificar las restrinciones de formato y contenido de los datos de entrada.
2. A partir de las restrinciones identificar las clases de equivalencias. Contemplando tanto datos válidos como erróneos.
3. Se identifican y definen los casos de prueba a partir de las clases de equivalencias.

Para asegurar la estabilidad en el resultado de las pruebas realizadas en cada iteración, se ha implementado un mecanismo que lleve a cabo las pruebas de forma regresiva, ejecutandose todo el conjunto de estas cada vez que una nueva funcionalidad del software es desarrollada.

Además de realizar pruebas funcionales en cada iteración, se han realizado una serie de pruebas no funcionales. Esta comprende aquellas que aseguran que se cumplen los requisitos no funcionales.

3. Entorno de pruebas

En este punto se define el entorno utilizado para las pruebas, tanto a nivel de software como a nivel de hardware.

3.1. Hardware

Para las pruebas se ha tomado un PC de características medias, en el momento de la realización del proyecto. Este se conforma de:

CPU: Intel Core i5-4460 3.2Ghz

Disco Duro: 1TB Western/Seagate

Memoria Ram: 4GB DDR3 1333 PC3-10600 CL9 Kingston

Placa Base: Gigabyte GA-H81M-S2H

Tarjeta gráfica: 1GB Integrada Intel

3.2. Software

Para las pruebas se ha utilizado un sistema GNU/Linux, instalado y configurado desde una distribución de paquetes Debian 8 Jessie.

Sistema operativo: GNU/Linux

Distribución de paquetes: Debian Jessie 8

Entorno gráfico: Xfce 4

Interprete de comando: Bash 4.3.8

Interprete OMI: OMI 0.1

4. Roles

Las pruebas se han llevado a cabo desde la perspectiva del único rol que interactuará con el sistema correspondiente al usuario que normalmente será un programador que hará uso del mismo.

El usuario programador hace uso integro del sistema en función a los progrmas que desarrolle en el lenguaje. Un único programa, o conjunto finito de estos no son sufientes para probar todos los aspectos del sistema. Así aunque las pruebas son llevadas a cabo desde un perfil programador, estas no constituyen programas completos con un objetivo específico, sino baterías de pruebas que pretenden probar cada aspecto funcional del sistema.

5. Niveles de pruebas

Las pruebas se presentan en distintos niveles, según el tipo de prueba realziado. Dado la gran cantidad de pruebas estas en su mayoría han sido automatizadas, presentándose junto al sistema y siendo una característica más del mismo, lacual es recomendable ejecutar tras cada instalación.

5.1. Pruebas unitarias

Son pruebas llevadas a cabo sobre cada artefacto o pieza software producido en cada iteración del ciclo de desarrollo. Estas aseguran la correcta implementación de las piezas desarrolladas, comprobando que están libre de errores y que cada entrada es procesada correctamente. Además cada caso de prueba se ha estructurado en función los tipos de entradas para una mejor organización de las mismas.

Las pruebas unitarias estan recojidas en un subsistema de la aplicación que automatizan su ejecución y asegura que son llevadas a cabo de forma regresiva.

A continuación se expone algunos ejemplos de las pruebas unitarias realizadas:

5.1.1. Asignación de booleanos

Estos casos de pruba se centran en el operador asignación cuando el elemento asignado es de valor boolenano.

Entrada:

```
1 a = true;
```

Descripción: Asignación sobre la varible a el valor booleano true.

Salida esperada: a tiene el valor true.

Salida obtenida: a tiene el valor true.

Entrada:

```
1 a = false;
```

Descripción: Asignación sobre la varible a el valor booleano false.

Salida esperada: b tiene el valor false.

Salida obtenida: b tiene el valor false.

Entrada:

```
1 for (i = 0; i < 10; ++i)
2   a = true;
```

Descripción: A la variable a se le asigna el valor booleano true en cada iteración del bucle

Salida esperada: a tiene el valor true.

Salida obtenida: a tiene el valor true.

Entrada:

```
1 a = b = true;
```

Descripción: A la variable b se le asigna el valor booleano true, el valor de b es asignado a a

Salida esperada: Tanto b como a tienen el valor true.

Salida obtenida: Tanto b como a tienen el valor true.

Entrada:

```
1 array[0] = true;
```

Descripción: A el array *array* se le asigna en el índice 0 el valor true

Salida esperada: *array* contiene en la para la clave 0 el valor true. Si *array* no existe es creado.

Salida obtenida: *array* contiene en la para la clave 0 el valor true. Si *array* no existe es creado.

Entrada:

```
1 array[1] = true
```

Descripción: A el array *array* se le asigna en el índice 1 el valor false

Salida esperada: *array* contiene en la para la clave 1 el valor false. Si *array* no existe es creado.

Salida obtenida: *array* contiene en la para la clave 1 el valor false. Si *array* no existe es creado.

Entrada:

```
1 for (i = 0; i < 10; ++i)
2   array[i] = true;
```

Descripción: A el array *array* se le asigna el valor true a los índices que van desde 0 a 9.

Salida esperada: *array* contiene en la para las claves del 0 al 9 el valor true. Si *array* no existe es creado.

Salida obtenida: *array* contiene en la para las claves del 0 al 9 el valor true. Si *array* no existe es creado.

Entrada:

```
1 str = "ABCDEF";
2 str[0] = false;
```

Descripción: A la cadena *str* se le asigna en la posición 0 el valor false.

Salida esperada: La cadena *str* queda con el ínice 0 con el valor false. La cadena resultante es ”BCDEF”.

Salida obtenida: La cadena *str* queda con el ínice 0 con el valor false. La cadena resultante es ”BCDEF”.

Entrada:

```
1 str = "ABCDEF";
2 str[0] = true;
```

Descripción: A la cadena *str* se le asigna en la posición 0 el valor true.

Salida esperada: La cadena *str* queda con el ínice 0 con el valor true. La cadena resultante es ”1BCDEF”.

Salida obtenida: La cadena *str* queda con el ínice 0 con el valor true. La cadena resultante es ”1BCDEF”.

Entrada:

```
1 while (size str)
2   str[0] = false;
```

Descripción: A la cadena *str* se le asigna en la posición 0 el valor false, mientras que la cadena sea distinta a la cadena vacía.

Salida esperada: La cadena *str* queda vacía.

Salida obtenida: La cadena *str* queda vacía.

Entrada:

```
1 a = false;
2 b = &a;
3 a = true;
```

Descripción: A a se le asigna el valor false, a b se le asigna una referencia a a, a a se le asigna el valor true.

Salida esperada: Tanto el valor de b como el de a es true.

Salida obtenida: Tanto el valor de b como el de a es true.

Entrada:

```
1 true = true;
```

Descripción: A la constante true se le asigna la constante true.

Salida esperada: Error asignación a constante.

Salida obtenida: Error asignación a constante.

Entrada:

```
1 true = false;
```

Descripción: A la constante true se le asigna la constante false.

Salida esperada: Error asignación a constante.

Salida obtenida: Error asignación a constante.

Entrada:

```
1 a = true = true;
```

Descripción: A a se le asigna el valor de asignar la constante true se le asigna la constante true.

Salida esperada: Error asignación a constante. *a* permanece con el valor qeu tenía.

Salida obtenida: Error asignación a constante. *a* permanece con el valor qeu tenía.

5.1.2. Operador de igualdad con operandos numéricos

Estos casos de prueba se enfoca en el operador de igualdad cuando las entradas son de tipo numéricas.

Entrada:

```
1 0 == 0;
```

Descripción: Operador igualdad donde el primer operando es 0 y el segundo operando es 0.

Salida esperada: Valor booleano true.

Salida obtenida: Valor booleano true.

Entrada:

```
1 0 == 1;
```

Descripción: Operador igualdad donde el primer operando es 0 y el segundo operando es 1.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1 1 == 0;
```

Descripción: Operador igualdad donde el primer operando es 1 y el segundo operando es 0.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1 10 == 10;
```

Descripción: Operador igualdad donde el primer operando es 10 y el segundo operando es 10.

Salida esperada: Valor booleano true.

Salida obtenida: Valor booleano true.

Entrada:

```
1 10 == 2;
```

Descripción: Operador igualdad donde el primer operando es 10 y el segundo operando es 2.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1 2 == 10;
```

Descripción: Operador igualdad donde el primer operando es 2 y el segundo operando es 10.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1 -12 == -12;
```

Descripción: Operador igualdad donde el primer operando es -12 y el segundo operando es -12.

Salida esperada: Valor booleano true.

Salida obtenida: Valor booleano true.

Entrada:

```
1 -12 == 12;
```

Descripción: Operador igualdad donde el primer operando es -12 y el segundo operando es 12.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1 12 == -12;
```

Descripción: Operador igualdad donde el primer operando es 12 y el segundo operando es -12.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1 18.7 == 18.7;
```

Descripción: Operador igualdad donde el primer operando es 18.7 y el segundo operando es 18.7.

Salida esperada: Valor booleano true.

Salida obtenida: Valor booleano true.

Entrada:

```
1 18.7 == 18;
```

Descripción: Operador igualdad donde el primer operando es 18.7 y el segundo operando es 18.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1 18 == 18.7;
```

Descripción: Operador igualdad donde el primer operando es 18 y el segundo operando es 18.7.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1 -18.7 == -18;
```

Descripción: Operador igualdad donde el primer operando es -18.7 y el segundo operando es -18.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1 -18.69 == -18.7;
```

Descripción: Operador igualdad donde el primer operando es -18.69 y el segundo operando es -18.7.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1 18.069 == -18.0;
```

Descripción: Operador igualdad donde el primer operando es 18.069 y el segundo operando es -18.0.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1 -11.7999999999999999 == -11.7999999999999999;
```

Descripción: Operador igualdad donde el primer operando es -11.7999999999999999 y el segundo operando es -11.7999999999999999.

Salida esperada: Valor booleano true.

Salida obtenida: Valor booleano true.

Entrada:

```
1 -11.7999999999999999 == -11.7999999999999991;
```

Descripción: Operador igualdad donde el primer operando es -11.7999999999999999 y el segundo operando es -11.7999999999999991.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1 -11.7999999999999999 == -11.7999999999999998;
```

Descripción: Operador igualdad donde el primer operando es -11.7999999999999999 y el segundo operando es -11.7999999999999998.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1 -11.7999999999999998 == -11.7999999999999999;
```

Descripción: Operador igualdad donde el primer operando es -11.7999999999999998 y el segundo operando es -11.7999999999999999.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1 -11.79999999999999999 == -11.79999999999999998;
```

Descripción: Operador igualdad donde el primer operando es -11.79999999999999999 y el segundo operando es -11.79999999999999998.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano true. Aviso de precisión sobrepasada por representación numérica finita.

Entrada:

```
1 -11.79999999999999999 == -11.8;
```

Descripción: Operador igualdad donde el primer operando es -11.79999999999999999 y el segundo operando es -11.8.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano true. Aviso de precisión sobrepasada por representación numérica finita.

Entrada:

```
1 999999999999999 == 1000000000000001;
```

Descripción: Operador igualdad donde el primer operando es 999999999999999 y el segundo operando es 1000000000000001.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
1 10000000000000001 == 10000000000000001;
```

Descripción: Operador igualdad donde el primer operando es 10000000000000001 y el segundo operando es 10000000000000001.

Salida esperada: Valor booleano true.

Salida obtenida: Valor booleano true.

Entrada:

```
10000000000000001 == 10000000000000002;
```

Descripción: Operador igualdad donde el primer operando es 10000000000000001 y el segundo operando es 10000000000000002.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
10000000000000002 == 10000000000000001;
```

Descripción: Operador igualdad donde el primer operando es 10000000000000002 y el segundo operando es 10000000000000001.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
10000000000000002 == 10000000000000001;
```

Descripción: Operador igualdad donde el primer operando es 10000000000000002 y el segundo operando es 10000000000000001.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano false.

Entrada:

```
9999999999999999 == 100000000000000001;
```

Descripción: Operador igualdad donde el primer operando es 99999999999999999 y el segundo operando es 100000000000000001.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano true. Aviso de precisión sobrepasada por representación numérica finita.

Entrada:

```
100000000000000001 == 100000000000000002;
```

Descripción: Operador igualdad donde el primer operando es 100000000000000001 y el segundo operando es 100000000000000002.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano true. Aviso de precisión sobrepasada por representación numérica finita.

Entrada:

```
100000000000000001 == 100000000000000002;
```

Descripción: Operador igualdad donde el primer operando es 100000000000000002 y el segundo operando es 100000000000000001.

Salida esperada: Valor booleano false.

Salida obtenida: Valor booleano true. Aviso de precisión sobrepasada por representación numérica finita.

5.1.3. Definición y llamadas de funciones con cuerpo vacío

Los casos de pruebas expuestos a continuación comprenden la definión y posterior llamada de funciones con cuerpo vacío.

Entrada:

```
1   function empty_fun () { }
2   empty_fun ();
```

Descripción: Definición de función sin parámetros y cuerpo vació. Llamada respetando el número de parámetros

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce resultado.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce resultado.

Entrada:

```
1   function empty_fun () { }
2   empty_fun ("param");
```

Descripción: Definición de función sin parámetros y cuerpo vació. Llamada sin respetar el número de parámetros (más de los definidos).

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Entrada:

```
1   function empty_fun (param) { }
2   empty_func ("param");
```

Descripción: Definición de función con un único parámetro y cuerpo vació. LLamada respetando número de parámetros.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Entrada:

```
1   function empty_fun (param) { }
2   empty_func ();
```

Descripción: Definición de función con un único parámetro y cuerpo vació. LLamada sin respetar el número de parámetros (menos de los definidos).

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Entrada:

```
1   function empty_fun (param) { }
2   empty_func ("param1", "param2");
```

Descripción: Definición de función con un único parámetro y cuerpo vació. LLamada sin respetar el número de parámetros (más de los definidos).

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Entrada:

```
1   param = "param";
2   function empty_fun (&param) { }
3   empty_func (param);
```

Descripción: Definición de función con un único parámetro pasado por referencia y cuerpo vació. Llamada dando una variable como valor de la referencia.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce resultado alguno.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce resultado alguno.

Entrada:

```
1   param = {"param"};
2   function empty_fun (&param) { }
3   empty_func (param[0]);
```

Descripción: Definición de función con un único parámetro pasado por referencia y cuerpo vació. Llamada dando la posición de un array como valor de la referencia.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce resultado alguno.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce resultado alguno.

Entrada:

```
1   function empty_fun (&param) { }
2   empty_func (param);
```

Descripción: Definición de función con un único parámetro pasado por referencia y cuerpo vació. Llamada dando una variable no definida como valor de la referencia.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce resultado alguno.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce resultado alguno.

Entrada:

```
1   function empty_fun (&param) { }
2   empty_func ("const");
```

Descripción: Definición de función con un único parámetro pasado por referencia y cuerpo vació. Llamada dando una constante como valor de la referencia.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de constante como referencia.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de constante como referencia.

Entrada:

```
1   function empty_fun (param1, param2) { }
2   empty_func ("param1", "param2");
```

Descripción: Definición de función con dos parámetros y cuerpo vació. Llamada respetando el número de parámetros.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Entrada:

```
function empty_fun (param1, param2) { }
empty_func ("param1");
```

Descripción: Definición de función con dos parámetros y cuerpo vació. Llamada sin respetar el número de parámetros (menos de los definidos).

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Entrada:

```
function empty_fun (param1, param2) { }
empty_func ("param1", "param2", "param3");
```

Descripción: Definición de función con dos parámetros y cuerpo vació. Llamada sin respetar el número de parámetros (más de los definidos).

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Entrada:

```
function empty_fun (param1, param2 = "default") { }
empty_function ("param1", "param2");
```

Descripción: Definición de función con dos parámetros, uno con valor por defecto, y cuerpo vació. Llamada facilitando todos los parámetros.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Entrada:

```
function empty_fun (param1, param2 = "default") { }
empty_function ("param1");
```

Descripción: Definición de función con dos parámetros, uno con valor por defecto, y cuerpo vació. Llamada facilitando los parámetros que no tienen valor por defecto.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Entrada:

```
function empty_fun (param1, param2 = "default") { }
empty_function ();
```

Descripción: Definición de función con dos parámetros, uno con valor por defecto, y cuerpo vació. Llamada sin facilitar los parámetros que no tienen valor por defecto.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Entrada:

```
function empty_fun (param1, param2 = "default") { }
empty_function ("param1", "param2", "param3");
```

Descripción: Definición de función con dos parámetros, uno con valor por defecto, y cuerpo vació. Llamada sin respetar el número de parámetros (más de los definidos).

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Entrada:

```
1   function empty_fun (param1 = "default", param2 = "default") { }
2   empty_fun ("param1", "param2");
```

Descripción: Definición de función con dos parámetros, todos con valor por defecto, y cuerpo vació. Llamada dando valor a todos los parámetros

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Entrada:

```
1   function empty_fun (param1 = "default", param2 = "default") { }
2   empty_fun ("param1");
```

Descripción: Definición de función con dos parámetros, todos con valor por defecto, y cuerpo vació. Llamada dando valor a algunos de los parámetros

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Entrada:

```
function empty_fun (param1 = "default", param2 = "default") { }
empty_fun ();
```

Descripción: Definición de función con dos parámetros, todos con valor por defecto, y cuerpo vació. Llamada tomando valor por defecto de todos los parámetros.

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada no produce ningún resultado.

Entrada:

```
function empty_fun (param1 = "default", param2 = "default") { }
empty_fun ("param1", "param2", "param3");
```

Descripción: Definición de función con dos parámetros, todos con valor por defecto, y cuerpo vació. Llamada sin respetar el número de parámetros (más de los definidos).

Salida esperada: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

Salida obtenida: Se define la función especificada y se asocia al identificador dado. La llamada produce un error de número de parámetros incorrecto.

5.2. Pruebas de integración

Las prueba de integración son llevadas a cabo tras iteración del ciclo de vida. Incluyen casos de prueba correspondiente a la interacción de varios módulos o artefactos, desarrollados en la misma iteracción del ciclo de desarrollo o anteriores. En este tipo de pruebas se ha de repasar las características desarrolladas en iteraciones anteriores con el objetivo de localizar errores en la integración de los módulos desarrollados con el resto del sistema.

En esta sección se recogen algunos casos de pruebas de integración. Junto el software se facilita un sistema de pruebas que comprueba automáticamente todos los casos.

Entrada:

```
<< 22 / 2 + 8 - 5 * 2;
```

Descripción: Impresión de exresión aritmética compuesta de varios operadores y operandos todos constantes.

Salida esperada: Se ha de imprimir en pantalla el resultado de la expresión: 9.

Salida obtenida: Se imprime en pantalla el resultado de la expresión: 9.

Entrada:

```
array = {22};
var = 5;
<< array[0] / 2 + 8 - var * 2;
```

Descripción: Impresión de exresión aritmética compuesta de varios operadores, algunos operandos variables y/o posiciones de array.

Salida esperada: Se ha de imprimir en pantalla el resultado de la expresión: 9.

Salida obtenida: Se imprime en pantalla el resultado de la expresión: 9.

Entrada:

```
function const2 () {
   return 2;
}
array = {22};
var = 5;
<< array[0] / const2 () + 8 - var * const2 ();
```

Descripción: Impresión de exresión aritmética compuesta de varios operadores, algunos operandos variables, llamadas a funciones y/o posiciones de array.

Salida esperada: Se ha de imprimir en pantalla el resultado de la expresión: 9.

Salida obtenida: Se imprime en pantalla el resultado de la expresión: 9.

Entrada:

```
<< !((true && true)  false );
```

Descripción: Impresión de exresión booleana compuesta de varios operadores y operandos todos constantes.

Salida esperada: Se ha de imprimir en pantalla el resultado de la expresión: false.

Salida obtenida: Se imprime en pantalla el resultado de la expresión: false.

Entrada:

```
array = {true};
var = true;
<< !((array [0] && var)  false );
```

Descripción: Impresión de exresión booleana compuesta de varios operadores, algunos operandos variables y/o posiciones de array.

Salida esperada: Se ha de imprimir en pantalla el resultado de la expresión: false.

Salida obtenida: Se imprime en pantalla el resultado de la expresión: false.

Entrada:

```
function identity (param) {
   return param;
}
array = {true};
var = true;
<< !((array [0] && var)  identity (false));
```

Descripción: Impresión de exresión booleana compuesta de varios operadores, algunos operandos variables, llamadas a función y/o posiciones de array.

Salida esperada: Se ha de imprimir en pantalla el resultado de la expresión: false.

Salida obtenida: Se imprime en pantalla el resultado de la expresión: false.

Entrada:

```
class fac {
 function factorial (a) {
   fac = 1;
   this->recursiva (a, fac);
   return fac;
 }
 private function recursiva (a, &factorial){
   if (a > 0){
     factorial *=  a;
     this->recursiva (a - 1, factorial);
   }
 }
}
f = new fac ();
<< f->factorial (4);
```

Descripción: Definición de clases con métodos públicos y privados. Paso de parámetros por referencia. Operadores artiméticos.

Salida esperada: Calcula el factorial de 4 de forma recursiva: 24.

Salida obtenida: Calcula el factorial de 4 de forma recursiva: 24.

5.3. Pruebas funcionales del sistema

Este tipo de pruebas tienen como objetivo comprobar la funcionalidad del sistema. La funcionalidad principal que debe cumplir el interprete es recibir un programa en forma de código fuente, interpretarlo y producir el resultado esperado.

Para comprobar que el sistema cumple la funcionalidad para el que fue diseñado se han realizado una serie de programas tipos, recojiendo diversos estilos de programación, y de una naturaleza distinta. Estos programas han sido codificados en el lenguaje reconocido por el intérprete y se han realizado distintas comprobaciones sobre los mismos.

5.3.1. Calculadora

Este programa representa una calculadora sencilla en la que se le pide al usuario dos operándos numéricos y una operación (suma, resta, producto o cociente). El programa muestra el resultado de la operación. El programa se ejecuta hasta que la operación dada no es reconocida.

```
#!/usr/local/bin/omi
#Calculadora sencilla.
while ( true ) {
   << "========================";
   << "Calculadora";
   << "Dame un numero";
   >> a;
   << "Dame otro";
   >> b;
   << "Dame una operacion [0=>suma], [1=>resta], [2=>multi], [3=>divide], [otro=>sale]";
   >> op;
   if (op == 0)
      << a << " + " << b << " = " << ( a + b );
   elif (op == 1)
      << a << " - " << b << " = " << (a - b);
   elif (op == 2)
      << a << " * " << b << " = " << (a * b);
   elif (op == 3) {
      if (op == 0)
         << "Error: no es posible dividir entre 0";
      else
         << a << " / " << b << " = " << (a / b);
   }
   else {
      << "Adios";
      break;
   }
}
```

5.3.2. Sistema de cuestionarios

El siguiente programa representa un sistema de cuestionarios. Es en si mismo un DSL (lenguaje específico de dominio) definido de forma interna, por lo que presenta una estructura y gramática similar al lenguaje reconocido por el intérprete.

Un cuestioanrios se define mediante preguntas y posibles respuestas. Cada pregunta presenta una valor, la suma de los valores de todas las preguntas se corresponde con el valor del cuestionario. El usuario que realice el cuestionario sacará una nota que se corresponderá con una parte del total según las preguntas que responda correctamente.

Cada pregunta del cuestionario tendrá una serie de posibles respuestas, las respuestas pueden ser de dos tipos:

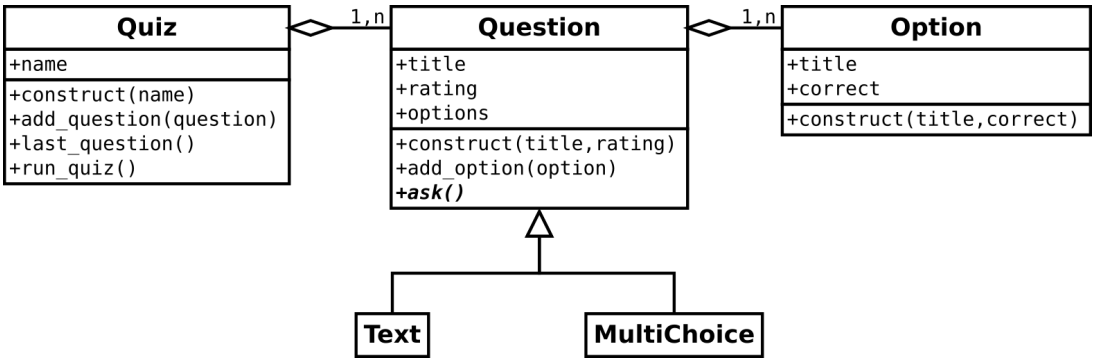
De selección: En este caso tras la pregunta se dará una serie de respuestas acompañadas de un valor que indique si es correcta o falsa. Para una misma pregunta pueden existir más de una respuesta correcta. En la ejecución del cuestionario al usuario se le permitirá eligir entre todas las respuestas aquella (solo una) que considere es correcta.

De texto: En este caso tras la pregunta se dará una serie de respuestas todas correctas. En la ejecución del cuestionario se le permitirá al usuario introducir textualmente

la solución, y solo en el caso de que se coincida con alguna de las repuesta esta será dada como correcta.

Este programa ha sido modelado usando programación orientada a objetos. Se presenta un fichero de código fuente por cada clase que conforma el sistema de cuestionarios, otro fichero la definición del DSL junto con el flujo principal que conforma el motor de cuestionarios, y otro fichero que se corresponde con un custionario de ejemplo.

A continuación el diagrama de clases que ilustrna el diseño del sistema:



file: quiz_system.omi

```
#!/usr/local/bin/omi
#Sistema de cuestionarios
#================================================================================
include "quiz.class.omi";
#================================================================================
global quiz;
#================================================================================
#Funciones del DSL
~multichoice (text, rating) {
   quiz->add_question (new MultiChoice (text, rating));
}

~text (text, rating) {
   quiz->add_question (new Text (text, rating));
}

~option (text, correct = null) {
   quiz->last_question()->add_option(new Option (text, correct));
}
#================================================================================
if (args[1]){
   title = args[2]?:"Sin titulo";
   << title;
   quiz =  new Quiz (title);
   include args[1];
   quiz->run_quiz();
}
else
   << "Debe indicar un cuestionario";
#================================================================================
```

file: quiz.class.omi

```
#quiz.class.omi
#============================================================================
include "question.class.omi";
#============================================================================
class Quiz {
   name = '';
   questions = {};
   ~ Quiz (name) {
      this->name = name;
   }
   ~ add_question (question) {
      this->questions[size this->questions] = question;
   }
   ~ last_question () {
      return this->questions [(size this->questions) - 1];
   }
   ~ run_quiz () {
      count = 0;
      total = 0;
      $(this->questions) {
         if ($->ask ()) count += $->rating;
         total += $->rating;
      }
      << "Tienes " << count << " respuestas correctas de " << total;
   }
}
#============================================================================
```

file: question.class.omi

```
#question.class.omi
#============================================================================
include "option.class.omi";
#============================================================================
class Question {
   title = "";
   rating = 0;
   options = {};
   ~ Question (title, rating) {
      this->title = title;
      this->rating = rating;
   }
   ~ add_option (option) {
      this->options [(size this->options)] = option;
   }
}
#----------------------------------------------------------------------------
class Text extends Question {
   ~ Text (title, rating) {
      this->Question (title, rating);
   }
   ~ ask () {
      << "";
      << this->title << " (" << this->rating << ")";
      << "Introducir respuesta: ";
      >> ans;
      n = size (this->options);
      for (i = 0; i < n; ++i) {
         if (ans == this->options[i]->title)
            return true;
      }
      return false;
   }
}
#----------------------------------------------------------------------------

```

```
class MultiChoice extends Question {
   ~ MultiChoice (title, rating) {
      this->Question (title, rating);
   }
   ~ ask () {
      << this->title << " (" << this->rating << ")";
      count = 0;
      $(this->options) <<  (++count) << " - " << $->title;
      << "Introducir respuesta: ";
      >> ans;
      return (this->options[ans - 1]) && this->options[ans - 1]->correct;
   }
}
#============================================================================
```

file: option.class.omi

```
#option.class.omi
#============================================================================
class Option {
   title = "";
   correct = "";
   ~ Option (title, correct){
      this->title = title;
      this->correct = correct;
   }
}
#============================================================================
```

input file: quiz.q

```
#Ejemplo de cuestionario.
multichoice("Cuanto tiempo duro la guerra de los 100 annos?", 2.5);
option (100, false);
option (116, true);
option (90, false);
option (102, false);
multichoice("Un simil es ...", 2.5);
option ("Una comparacion", true);
option ("Una duda", false);
option ("Un aparato para medir el tiempo", false);
text ("En que provincia desemboca el rio Gualquivir?", 2.5);
option("Cadiz", true);
option("cadiz", true);
```

Para ejecutar el cuestionario en una terminal de comandos:

```
prompt$ ./quiz_system.omi quiz.q "Cuestionario"
```

5.3.3. Tic-Tac-Toe

El programa que se presenta a continuación se corresponde con el juego del Tic-Tac-Toe. Este juego, también llamado el juego de tres en raya, enfrenta a dos jugadores en una cuadrícula de 3x3. Cada jugador se corresponde con un símbolo, y se turnan para ponerlo o dibujarlo en una posición vacía de la cuadrícula. El jugador que consiga poner tres de sus símbolos en línea gana la partida.

No es difícil darse cuenta que si ambos jugadores utilizan la estratégia más óptima el juego terminará en empate. Al ser un juego sencillo se utiliza para enseñar conceptos de teoría de juegos y, dentro de la inteligencia artificial, la búsqueda de árboles de juego.

El programa en primer lugar solicita el nombre y tipo de los jugadores, pudiéndose ser estos humanos o máquinas. Luego turno a turno va solicitando a cada jugador una posición vacía de la cuadrícula en la que efectuar el movimiento, esto se hace hasta que se da una línea ganadora o hasta que se completa la cuadrícula.

Para determinar el movimiento o acción llevada en cada turno se prosigue de la siguiente forma:

- Si el jugador es humano el sistema solicita la posición (fila y columna) en la que poner el símbolo.
- Si el jugador es de tipo máquina se utiliza el algoritmo recursivo minimax para calcular el mejor movimiento a partir de busquedas en árboles de juego y la estado del tablero. Esto se hace con la salvedad de los primeros turnos, ya que en estos la estratégia óptima es fija.

El programa se divide en tres módulos, cada uno correspondiente a un fichero. En un fichero se disponen las funciones de entrada y salida, tales como solicitar los datos de los jugadores, imprimir el tablero, etc. En otro fichero se encuentran las funciones de inteligencia artifical para el caso de jugadores de tipo máquina. Y en otro se encuentra el flujo principal correspondiete al bucle de juego.

file: IO.omi

```
#IO.omi
#============================================================================
~IOJugadores () {
   jugadores = {};
   for (i = 1; i <= 2; ++i) {
      << "Nombre Jugador " << i;
      >> nombre;
      << "Tipo Jugador " << i << " [ 0 => Humano, otro => Maquina ]";
      >> tipo;
      if (tipo != 0)
         tipo = 1;
      jugadores [] = {
         'nombre' : nombre,
         'tipo' : tipo,
         'token' : (( i == 1)?1:-1),
      };
   }
   return jugadores;
}
#----------------------------------------------------------------------------
~ IOTablero (tablero) {
   for (i = 0; i < 3; ++i)
      << IOToken(tablero[i][0]) << " or "
         << IOToken(tablero[i][1]) << " or "
         << IOToken(tablero[i][2]);
}
#----------------------------------------------------------------------------
```

```
~ IOToken (pos) {
   if (pos == 1)
      return 'X';
   elif (pos == -1)
      return 'O';
   else
      return '#';
}
#-------------------------------------------------------------------------
~ IOMover (tablero) {
   do {
      << "Dame la fila";
      >> row;
      << "Dame la columna";
      >> col;
   }while (tablero[row][col] !== 0);
   return {row, col};
}
#-------------------------------------------------------------------------
~IOGanador (jugadores, ganador) {
   if (ganador == 1 ){
      << "El ganador es " << jugadores[0]["nombre"];
   }elif (ganador == -1){
      << "El ganador es " << jugadores[1]["nombre"];
   }else{
      << "La partida ha quedado en empate";
   }
}

#=========================================================================
```

file: AI.omi

```
#AI.omi
#=========================================================================
~primerosMov (tablero) {
   if (!tablero[1][1])
      return {1,1};
   do {
      col = row = time () % 2;
      if (row == 1) row = 2;
      if (col == 1) col = 2;
   }while (tablero[row][col] !== 0);
   return {row, col};
}
#-------------------------------------------------------------------------
~ miniMax (A, turno){
  mejor = turno * -1;
   minMov = 9;
   poda = 1;
   Mov = 0;
   posicion = {0, 0};
  if (!(t_ganador = procesarTablero (A)) && !tableroLleno (A)){
    for (cont = 0; cont < 3 && poda; cont ++)
      for (cont2 = 0; cont2 < 3 && poda; cont2 ++){
        if ( A [cont] [cont2] == 0){
          A [cont] [cont2] = turno;
          actual = miniMax_R (A, turno * -1,0,Mov);
          if (turno == 1 ){
            if ( actual  >= mejor && Mov <= minMov){
              mejor =actual;
              posicion [0] = cont;
              posicion [1] = cont2;
              if (mejor == turno){
                minMov = Mov;
                if (mejor == 1 && minMov == 0)
                  poda = 0;
              }
```

```
36            }
37          } else
38            if ( actual <= mejor && Mov <= minMov){
39              mejor =actual;
40              posicion [0] = cont;
41              posicion [1] = cont2;
42              if (mejor == turno){
43                minMov = Mov;
44                if (mejor == 1 && minMov == 0)
45                  poda = 0;
46              }
47            }
48          A [cont] [cont2] = 0;
49        }
50      }
51  }
52   return posicion;
53}
54#--------------------------------------------------------------------------
55~ miniMax_R (&A, turno, nMov, &Mov){
56  mejor = turno * -1;
57   poda = 1;
58   minMov = 9 ;
59  if (!(t_ganador = procesarTablero (A)) && !tableroLleno (A)){
60    for (cont = 0; cont < 3 && poda; cont ++){
61      for (cont2 = 0; cont2 < 3 && poda; cont2 ++){
62        if ( A [cont] [cont2] == 0){
63          A [cont] [cont2] = turno;
64          actual = miniMax_R (A, turno * -1, nMov +1, Mov);
65          if (turno == 1 ){
66            if ( actual  >= mejor && Mov <= minMov){
67              mejor =actual;
68              if (mejor == turno){
69                minMov = Mov;
70                if (mejor == 1 && minMov == 0)
71                  poda = 0;
72              }
73            }
74          } else
75            if ( actual <= mejor && Mov <= minMov){
76                mejor =actual;
77                if (mejor == turno){
78                  minMov = Mov;
79                  if (mejor == 1 && minMov == 0)
80                    poda = 0;
81              }
82            }
83          A [cont] [cont2] = 0;
84        }
85      }
86    }
87    Mov = minMov;
88    return mejor;
89  }
90  Mov = nMov;
91  return  t_ganador;
92}
93#--------------------------------------------------------------------------
94~procesarTablero (A) {
95    ganador = 0;
96    cont = 0;
97  for (cont = 0; cont < 3 && ganador == 0; cont ++){
98    if (A [cont] [0] == A[cont] [1] && A [cont] [1] == A [cont] [2])
99      ganador = A [cont] [1];
100   }
101  for (cont = 0; cont < 3 && ganador == 0; cont ++)
102    if (A [0] [cont] == A [1] [cont] && A [1] [cont] == A [2] [cont])
103      ganador = A [0] [cont];
104  if (A [0][0] == A [1][1] && A [1][1] == A [2][2]  && ganador == 0 )
```

```
105     ganador = A [0][0];
106   if (A [0][2] == A [1][1] && A [1][1] == A [2][0] && ganador == 0)
107     ganador = A [0][2];
108   return ganador;
109 }
110 #-------------------------------------------------------------------------
111 ~ tableroLleno (A){
112   resp = 1;
113   for (cont = 0; cont < 3 && resp; cont ++)
114     for (cont2 = 0; cont2 < 3 && resp; cont2 ++)
115       resp = A [cont] [cont2] != 0;
116   return resp;
117 }
118 #=========================================================================
```

file: tictactoe.omi

```
1 #!/usr/local/bin/omi
2 #=========================================================================
3 include "IO.omi";
4 include "AI.omi";
5 #-------------------------------------------------------------------------
6 ~juego () {
7    tablero = {{0,0,0},{0,0,0},{0,0,0}};
8    posicion = {0,0};
9    turno = 0;
10   jugadores = IOJugadores ();
11   while (!(ganador = procesarTablero (tablero)) && !tableroLleno (tablero)){
12      << "-----------------------------------------------";
13      IOTablero (tablero);
14      << "\nTurno " << jugadores[turno%2]['nombre'];
15      if (jugadores[turno%2]['tipo'] == 0) {
16         posicion = IOMover (tablero);
17      }else {
18         << "Calculando movimiento...";
19         if (turno <= 1) {
20            posicion = primerosMov (tablero);
21         }else {
22            posicion = miniMax (tablero, jugadores[turno%2]['token']);
23         }
24      }
25      tablero[posicion[0]][posicion[1]] = jugadores[turno%2]['token'];
26      turno ++;
27   }
28   IOTablero (tablero);
29   IOGanador (jugadores, ganador);
30 }
31 #-------------------------------------------------------------------------
32 juego ();
33 #=========================================================================$
```

5.4. Pruebas no funcionales del sistema

Estas pruebas están enfocadas a comprobar que el sistema cumple con los requisitos no funcionales determinado en las fases de especificación.

El interprete debe presentar un rendimiento óptimo en cuanto tiempo de interpretación. Dado que su objetivo no es constituir una herramienta para la producción de softwatre este aspecto no es crítico, no obstante debe cumplir unos mínimos para que sea

operativo.

Por otro lado un programa interpretado no puede excederse en la memoria física que ocupa. Para ello se debe medir la cantidad de memoria de las entidades que conforma el programa.

Además se debe asegurar dentro de unos márgenes que el intérte está libre de vulnerabilidades y que no es posible hacer un uso indebido del mismo para explotar la plataforma sobre la que se ejecuta o el propio sistema software.

5.4.1. Rendimiento de tiempo

Para comprobar que el rendimiento que ofrece el software en relación a los tiempos tomados para la interpretación se ha sometido al sistema a una serie benchmarks conocidos, comparando los resultados con los obtenidos con otros lenguajes de programación interpretados.

Se considera que el sistema supera las pruebas de rendimiento siempre y cuando el tiempo en pasar los benchmarks sea inferior al doble de los lenguajes pensados para la producción software.

5.4.1.1 Fibonacci

Esta prueba conciste en medir el tiempo para dististas entradas de un programa que trata de calcular, de forma recursiva, el número correspondiente a una determinada posición de la sucesión de Fibonacci.

A continuación el código fuente:

file: fibonacci.omi

```
#!/usr/local/bin/omi
#fibonacci.omi
#=============================================================================
~ fibonaci (n) {
   if (n == 1  n == 2)
      return 1;
   else
      return fibonaci (n - 1) + fibonaci (n - 2);
}
#-----------------------------------------------------------------------------
<< fibonaci (args[1]);
#=============================================================================
```

Los tiempos obtenidos frente otros lenguajes son los siguientes:

- OMI:

Tamaño de entrada	Tiempo (s)
10	0.020
20	0.104
30	8.720

- PHP:

Tamaño de entrada	Tiempo (s)
10	0.043
20	0.054
30	5.622

- Python:

Tamaño de entrada	Tiempo (s)
10	0.023
20	0.029
30	4.963

5.4.1.2 N-body

El primer benchmark al que ha sido sometido el sistema se denomina "n-body". Este conciste en una simulación de un sistema dinámico de partículas que se encuentran bajo la influencia de fuerzas físicas como la gravedad.

El código fuente para la prueba es el siguiente:

file: n-body.omi

```
1#!/usr/local/bin/omi
2#============================================================================
3~ energy(&b) {
4   e = 0.0;
5   m = size (b);
6   for (i=0; i < m; ++i) {
7       b1=b[i];
8       e += 0.5*b1[6]*(b1[3]*b1[3]+b1[4]*b1[4]+b1[5]*b1[5]);
9       for (j=i+1; j<m; j++) {
```

```
10          b2=b[j];
11          dx=b1[0]-b2[0]; dy=b1[1]-b2[1]; dz=b1[2]-b2[2];
12          e -= (b1[6]*b2[6])/sqrt(dx*dx + dy*dy + dz*dz);
13        }
14    }
15    return e;
16}
17
18pi=3.141592653589793;
19solar_mass=4*pi*pi;
20days_per_year=365.24;
21
22bodies = {
23    {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, solar_mass }, //Sun
24    {
25       4.84143144246472090, //Jupiter
26       -1.16032004402742839,
27       -0.103622044471123109,
28       0.00166007664274403694 * days_per_year,
29       0.00769901118419740425 * days_per_year,
30       -0.0000690460016972063023 * days_per_year,
31       0.0009.54791938424326609 * solar_mass
32    },
33    {
34       8.34336671824457987,    // Saturn
35       4.12479856412430479,
36       -0.403523417114321381,
37       -0.00276742510726862411 * days_per_year,
38       0.00499852801234917238 * days_per_year,
39       0.00002.30417297573763929 * days_per_year,
40       0.00028588598066613081 2 * solar_mass
41    },
42    {
43       12.8943695621391310, // Uranus
44       -15.1111514016986312,
45       -0.223307578892655734,
46       0.00296460137564761618 * days_per_year,
47       0.00237847173959480950 * days_per_year,
48       -0.0000296589568540237556 * days_per_year,
49       0.0000436624404335156298 * solar_mass
50    },
51    {
52       15.3796971148509165, // Neptune
53       -25.9193146099879641,
54       0.179258772950371181,
55       0.00268067772490389322 * days_per_year,
56       0.00162824170038242295 * days_per_year,
57       -0.0000951592254519715870 * days_per_year,
58       0.0000515138902046611451 * solar_mass
59    }
60};
61
62// offset_momentum
63px=py=pz=0.0;
64for (bodies as e) {
65    px+=e[3]*e[6];
66    py+=e[4]*e[6];
67    pz+=e[5]*e[6];
68}
69bodies[0][3] = -1 * (px) / solar_mass;
70bodies[0][4] = -1 * (py) / solar_mass;
71bodies[0][5] = -1 * (pz) / solar_mass;
72
73pairs = {};
74m=size(bodies);
75for (i=0; i<m; ++i)
76    for (j=i+1; j<m; j++)
77       pairs[] = {bodies[i], bodies[j]};
78
```

```
n = args[1];

<< energy(bodies);

i=0;
do {
   for (pairs as p) {
      a=p[0]; b=p[1];
      dx=a[0]-b[0]; dy=a[1]-b[1]; dz=a[2]-b[2];

      dist = sqrt(dx*dx + dy*dy + dz*dz);
      mag = 0.01/(dist*dist*dist);
      mag_a = a[6]*mag; mag_b = b[6]*mag;

      a[3]-=dx*mag_b; a[4]-=dy*mag_b; a[5]-=dz*mag_b;
      b[3]+=dx*mag_a; b[4]+=dy*mag_a; b[5]+=dz*mag_a;
    }

    for (bodies as b) {
        b[0]+=0.01*b[3]; b[1]+=0.01*b[4]; b[2]+=0.01*b[5];
    }

} while(++i<n);

<< energy(bodies);
```

Los tiempos obtenidos frente otros lenguajes son los siguientes:

- OMI:

Tamaño de entrada	**Tiempo (s)**
500.000	14.10
5.000.000	240.4
50.000.000	1243.02

- PHP:

Tamaño de entrada	**Tiempo (s)**
500.000	7.10
5.000.000	69.00
50.000.000	719.66

- Python:

Tamaño de entrada	Tiempo (s)
500.000	9.86
5.000.000	96.17
50.000.000	967.81

5.4.2. Espacio de memoria

El espacio de meoria que ocupa un determinado programa en ejecución es de vital improtancia. La memoria es un recurso físico limitado, y los programas deben hacer un buen uso de la misma.

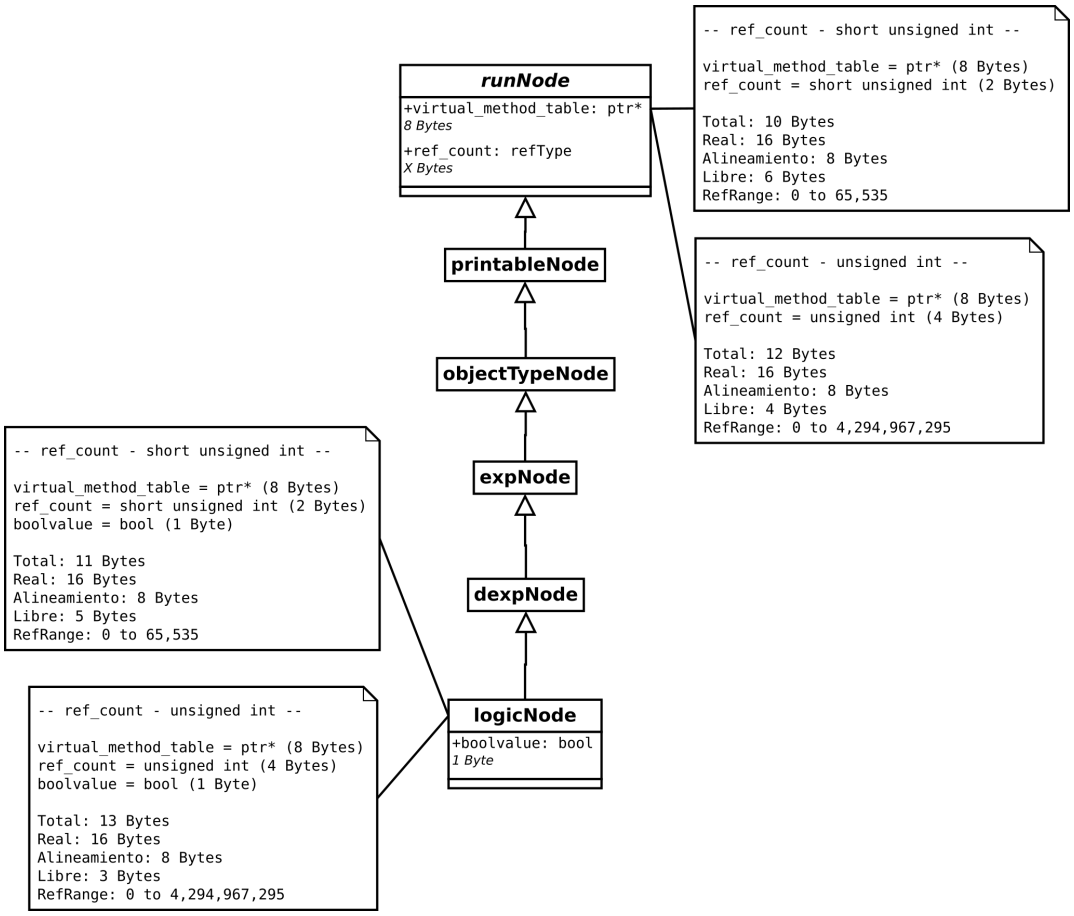
Dado que el interprete es una herramienta con la que se van a escribir otros programas, el uso de memoria que haga este influye en gran medida en la cantidad que ocuparán los programas con son interpretados. De esta forma si el interprete no hace un uso óptimo de la memoria los programas que este procesará tampoco.

Las unidades mínimas sobre las que opera el interprete son los nodos ejecutables. Así en primer lugar se va a medir cuánto ocupa los nodos ejecutables básicos y más comunes. Cabe decir que la representación interna de los distintos tipos de datos puede ser configurada como opciones de compilación, es por ello que se presenta la medición en función las distintas configuraciones posibles, indcándose solo aquellas que presentan un esquema óptimo debido a factores como el alineamiento.

La longitud del alineamiento en todos los casos viene dado por el puntero a la tabla de métodos virtuales. Este elemento referencia a una tabla que indexa los métodos de los que dispone un objeto debido a la jerarquía de herencia con la que se definió. Al ser un puntero su tamaño dependerá de la arquitectura del equipo.

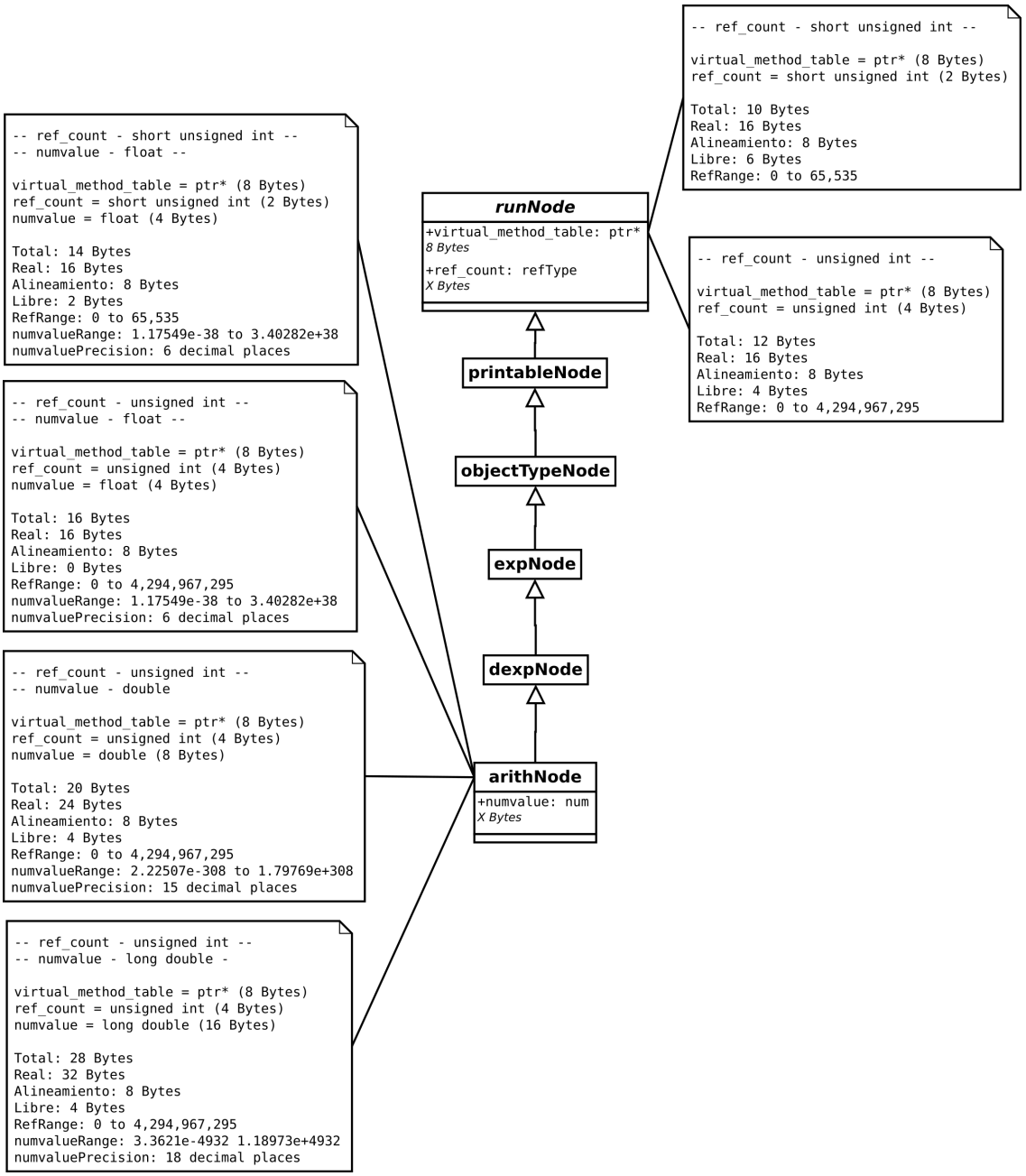
5.4.2.1 Nodos lógicos

En el siguiente diagrama se presenta la memoria ocupada por un nodo de tipo lógico según las distintas combinaciones para la representación interna de los datos:



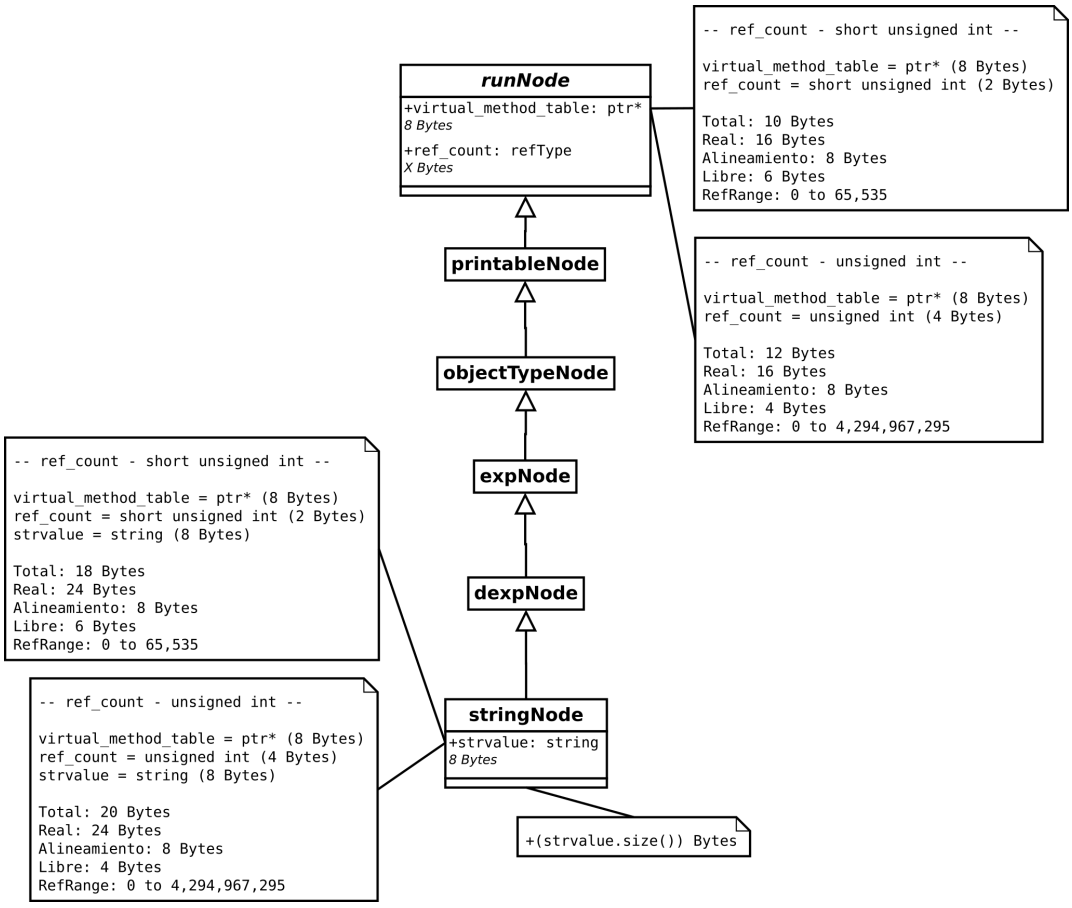
5.4.2.2 Nodos aritméticos

En el siguiente diagrama se presenta la memoria ocupada por un nodo de tipo aritmético según las distintas combinaciones para la representación interna de los datos:



5.4.2.3 Nodos cadenas de caracteres

En el siguiente diagrama se presenta la memoria ocupada por un nodo de tipo cadena de caracteres según las distintas combinaciones para la representación interna de los datos:



5.4.3. Seguridad

La seguidad digital de un empresa se debe comprobar en una serie de niveles, tal que se han de realiza auditorias internas, perimetrales, pruebas APT, etc. Sin embargo, el objetivo de este punto es medir la calidad en función de la seguridad del software desarrollado. Así, para asegurar que la aplicación cumple un mínimo de seguridad se han realizado una serie de auditorías al software.

En el tipo de pruebas realizadas solo se ha tenido en cuenta el software correspondiente al interprete, dejando fuera todo el sistema web que conforma la plataforma de distribución. Un sistema web se ve afectado por tipos de vulnaribilidades tales como DoS, XSS, CSRF, SQL injectión, sistemas de autentificación... Por otro lado en un software de escritorio, como puede ser el interprete, se ve afectado por otro tipos de vulnerabilidaes comunes.

Las pruebas que se han realizado sobre el software se enfocan en la entrada del usuario y se pueden categorizar de la siguinete forma:

Desbordamiento de buffer
Desbordamiento de buffer por variables de entorno
Desbordamiento de buffer por recursos binarios
Injección de código
Salto de directorio (path traversal)