# MANUAL DE USUARIO DEL PROTOTIPO DE COMPUTACIÓN REFLECTIVA SIN RESTRICCIONES

Este documento narra cómo instalar, programar y utilizar el prototipo de computación reflectiva sin restricciones implementado en el lenguaje de programación Python [Rossum2001]. La programación de lenguajes para el sistema utiliza la semántica de Python, por lo que es necesario conocer éste si se desea diseñar uno.

El código fuente, diseño y distintas baterías de prueba pueden descargarse de la dirección `http://www.di.uniovi.es/reflection/lab/prototypes.html`.

## A.1 Instalación y Configuración

El sistema reflectivo se ha codificado en Python 2.1, por lo que su descarga de `http://www.python.org` es necesaria previamente a su ejecución. Una vez descargado e instalado el intérprete del lenguaje Python, es necesario:

- Ubicar todos los archivos de extensiones `ml`, `py` y `pyw` en un directorio de nuestro sistema de archivos.
- Añadir a la variable de entorno `PYTHONPATH` la ruta en la que hayamos posicionado los archivos comentados.
- Ejecutar el archivo `nitrO.pyw` con el intérprete `pythonw`, para lanzar el sistema reflectivo.

La descripción detallada de cada uno de los archivos y directorios puede encontrarse dentro del archivo `readme.txt` que acompaña a la distribución de la aplicación.

## A.2 Metalenguaje del Sistema

Una de las características del sistema desarrollado es su independencia del lenguaje de programación. Cualquier aplicación utilizando cualquier lenguaje, puede ser ejecutada en nuestro sistema, interactuando dinámicamente con otras aplicaciones desarrolladas en otros lenguajes. Una aplicación podrá también construirse utilizando un conjunto de lenguajes.

Para que esta independencia del lenguaje sea posible en nuestro sistema de programación, es necesario idear un mecanismo de especificación de lenguajes que separe su descripción del sistema computacional. Para ello se ha descrito un lenguaje de especificación

de lenguajes: un metalenguaje. Haciendo uso de éste, se puede describir un lenguaje en nuestro sistema de tres modos:

1. Especificación del lenguaje mediante un archivo de extensión `ml`. Siguiendo la gramática § A.2.1, se describe el lenguaje y se almacena en el directorio del sistema con igual nombre que el lenguaje creado, y extensión `ml`.
2. Especificándolo en el archivo de aplicación y anteponiéndolo a su código (como veremos en § A.3.2).
3. Modificando un lenguaje ya existente previamente a la ejecución de una aplicación (detallado en § A.3.4).

### A.2.1 Gramática del Metalenguaje

La siguiente gramática representa la especificación del metalenguaje utilizado en las especificaciones de lenguajes para nuestro sistema; los elementos en negrita representan componentes léxicos, para separarlos de los propios de la notación EBNF [Cueva98]:

```
<startLang>    ::=   LANGUAGE = ID <scan> <parser>
                     <skip> <notskip>
<scan>         ::=   SCANNER = { <scannerRules> }
<parser>       ::=   PARSER = { <parserRules> }
<skip>         ::=   SKIP = { <tokens> }
<notskip>      ::=   NOTSKIP = { <tokens> }
<scannerRules> ::=   {<SR> ;}
<SR>           ::=   STRING ID -> <rightSR> {<moreRightSR>}
<rightSR>      ::=   {<rightSRItem>} <ruleCode>
<moreRightSR>  ::=   | <rightSR>
<rightSRItem>  ::=   STRING
                |    ID
<parserRules>  ::=   {<PR> ;}
<PR>           ::=   STRING ID -> <rightPR> {<moreRightPR>}
<rightPR>      ::=   <rightPRItems> <ruleCode>
<rightPRItems> ::=   {ID}
                |    _REIFY_
<ruleCode>     ::=   CODE
                |    λ
<moreRightPR>  ::=   | <rightPR>
<tokens>       ::=   {STRING ;}
```

La primera parte es la identificación del lenguaje. Este nombre ha de ser único para el lenguaje y deberá coincidir con el nombre del archivo (si se está creando un archivo `ml`). A continuación se especifican las descripciones léxica, sintáctica, y los tokens de escape y reconocimiento automático.

### A.2.2 Descripción Léxica

La descripción léxica se lleva a cabo dentro de la sección `Scanner` empleando reglas libres de contexto. Cada regla ha de estar precedida de una cadena de caracteres –entre comillas dobles– descriptiva de su significado. La notación en la que puede expresarse cada regla es en BNF (*Backus Naur Form)* [Cueva98].

Los elementos no terminales son identificadores y los terminales cadenas de caracteres sensibles a mayúsculas / minúsculas. Toda producción ha de finalizar con un punto y coma, y la parte derecha de la producción se separa de la izquierda con la pareja de caracteres "`->`". Toda producción puede tener alternativas en su parte derecha, separadas con el carácter "`|`".

Puesto que el tratamiento de estas producciones está basado en un algoritmo descendente con retroceso (*backtracking*), si dos partes derechas pueden ser válidas para la entrada analizada, la primera será analizada y la segunda ignorada. Es buen criterio a seguir si esto se produce, el ubicar con anterioridad aquellas reglas que posean una parte derecha de mayor longitud que aquellas con las que pueda tener conflictos. Por lo tanto, la producción al vacío ($\lambda$) deberá ubicarse como la última parte derecha de toda producción.

Las producciones pueden tener asociadas reglas semánticas para constituir definiciones dirigidas por sintaxis [Aho90]. El modo en que éstas son codificadas se describe en § A.2.5.

### A.2.3 Descripción Sintáctica

Las reglas sintácticas de nuestro metalenguaje se ubican en la sección `Parser`. El modo en el que éstas son representadas coincide con las empleadas en la descripción léxica. La única diferencia es que la parte derecha de una regla sintáctica no puede poseer símbolos gramaticales terminales –éstos deben estar previamente especificados en la parte léxica.

El símbolo gramatical no terminal, ubicado en la parte izquierda de la primera producción, representa el símbolo inicial de la gramática; no es necesario que éste posea un identificador determinado.

La separación entre reglas léxicas y sintácticas supone básicamente una agrupación conceptual. Además, como se comentará en § A.2.4, en el reconocimiento de una producción léxica se ignora la detección de tokens de escape –no en el caso de las reglas sintácticas.

### A.2.4 Tokens de Escape y Reconocimiento Automático

Las dos secciones restantes –`Skip` y `NotSkip`– facilitan la eliminación y reconocimiento automático de componentes léxicos en la aplicación a analizar. Si queremos que el analizador léxico del procesador de lenguaje elimine automáticamente un conjunto de tokens, debemos especificar éstos dentro de la sección `Skip`. El analizador sintáctico nunca tendrá noción de ellos.

La sección `NotSkip` produce el efecto contrario que `Skip`. Si queremos reconocer automáticamente un conjunto de tokens, sin necesidad de especificar su aparición sintácticamente, podremos hacerlo ubicándolos en esta sección. Un ejemplo de este tipo de tokens puede ser el tabulador en el lenguaje Python [Rossum2001]: el código ha de esta identado mediante tabuladores para saber a qué estructura de control pertenece, pero, ¿cómo contemplar esto en su gramática?

Si un token `NotSkip` es reconocido automáticamente, su lexema aparecerá en el texto (atributo `text` del nodo –ver siguiente punto) asociado al no terminal de la parte izquierda de la producción analizada.

### A.2.5 Especificación Semántica

Las producciones pueden especificar al final de ellas código Python representativo de su semántica. Este código ha de estar ubicado entre los pares de caracteres "<#" y "#>". Al codificar esta semántica, se ha de tener en cuenta que Python utiliza los tabuladores como identadores obligatorios y el salto de línea como separador de instrucciones.

La ejecución de una acción semántica posee el siguiente contexto:

- Todos los símbolos gramaticales del árbol tienen asociado un objeto. Éstos forman parte de una lista denominada `nodes`. El primer elemento (`nodes[0]`) es el objeto asociado al no terminal de la izquierda; el resto representan los nodos de la parte derecha, enumerados de izquierda a derecha.
- Todo nodo del árbol posee un atributo `text` que representa el código reconocido, eliminando los tokens `Skip` y habiendo reconocido automáticamente los `NotSkip`.
- La función global `write` visualiza en la ventana gráfica de la aplicación la cadena de caracteres pasada como parámetro.
- El objeto global `nitrO`, nos brinda todos los servicios de nuestro sistema computacional –ver § A.4.3.

Puesto que todo nodo es un objeto Python, y este lenguaje posee reflectividad estructural, podemos asignarle cualquier atributo dinámicamente. Al no tener chequeo estático de tipos [Cardelli97], podemos crear en tiempo de ejecución nuevos atributos mediante el operador de asignación. Esto hace que la herramienta suponga un mecanismo de codificación de definiciones dirigidas por sintaxis [Aho90].

Una vez que el árbol sintáctico de una aplicación haya sido creado, se ejecutará únicamente el código asociado a la producción del símbolo inicial; ésta deberá encargarse de llamar al resto de las evaluaciones semánticas. La regla semántica de un nodo se ejecuta al pasarle a éste el mensaje `execute`. Por lo tanto, si queremos que se evalúe un nodo, debemos invocar su método `execute`.

Para ver ejemplos prácticos de la descripción de lenguajes, examínense los archivos de extensión `ml` de la distribución del prototipo.

### A.2.6 Instrucción Reify

Aunque nuestro sistema sea independiente del lenguaje de programación, su flexibilidad se centra en la utilización de una instrucción que todo lenguaje posee: la instrucción `reify`. El programador de lenguajes sólo debe ubicar el terminal `_REIFY_` en aquella parte de la gramática donde pueda aparecer dicha instrucción.

La utilidad y funcionamiento de esta instrucción será explicada en § A.3.3.

## A.3 Aplicaciones del Sistema

La codificación de una aplicación en nuestro sistema, indistintamente del lenguaje de programación seleccionado, ha de seguir la siguiente gramática EBNF:

### A.3.1 Gramática de Aplicaciones

```
<startApp>      ::=   APPLICATION = STRING
```

```
                        LANGUAGE = <langSpec> <langAlt> APPCODE
<langSpec>      ::=     STRING
                |       <startLang>
<langAlt>       ::=     + CODE
                |       λ
```

El identificador único de la aplicación es la cadena de caracteres entre comillas dobles que se asigna a la palabra reservada `Application`. La parte final de un archivo de aplicación siempre es el código de ésta (`APPCODE`), siguiendo la gramática del lenguaje utilizado.

A la hora de implementar una aplicación, o un módulo de aplicación, dentro de nuestro sistema, debemos tener en cuenta que el lenguaje a utilizar ha de haber sido especificado de alguna de las tres formas mencionadas en A.2.

Si optamos por ubicar la especificación del lenguaje de programación en un archivo de extensión `ml`, éste deberá llamarse igual que el identificador del lenguaje y será asignado a la palabra reservada `Language`. El sistema buscará su especificación en el directorio del sistema, empleando el nombre de archivo especificado.

### A.3.2 Aplicaciones Autosuficientes

El segundo modo de especificar el lenguaje de programación a utilizar por una aplicación es incluyéndolo en la propia aplicación. Antes su codificación, puede especificarse el lenguaje a utilizar siguiendo la gramática descrita en A.2.1. Una vez descrito éste, la aplicación se codificará e interpretará en base a esta descripción.

La oportunidad de crear aplicaciones que puedan describir su propia sintaxis y semántica, hacen que éstas sean autosuficientes y directamente ejecutables. En cualquier plataforma en la que nuestro sistema esté instalado, este tipo de aplicaciones puede ejecutarse e interactuar con el resto de programas existentes en el sistema. Un ejemplo práctico de su utilización, haciendo uso de un paquete de distribución, es el desarrollo de un sistema de agentes móviles [Hohl96] independientes del lenguaje; las aplicaciones viajan por la red y se ejecutan en cualquier máquina con el lenguaje que ellas deseen, sin necesidad de que éste esté instalado en la plataforma de ejecución.

### A.3.3 Reflectividad No Restrictiva

Cuando desarrollamos la especificación de un lenguaje para nuestro sistema, definimos sus aspectos léxicos, sintácticos y semánticos. Una vez descritos éstos, una aplicación se codifica, valida y ejecuta en base a esta especificación que se mantiene invariable a lo largo de su ciclo de vida. Uno de los objetivos principales de este prototipo es obtener un modo de flexibilizar la especificación de un lenguaje, desde sus propias aplicaciones.

Como mostrábamos en A.2.6, todo lenguaje de nuestro sistema dispone de una instrucción `Reify` con tan solo ubicar el terminal `_REIFY_` en aquella posición de la gramática donde pueda aparecer dicha sentencia. Esta instrucción está constituida por la palabra reservada `reify`, seguida de código Python entre las parejas de caracteres “<#” y “#>”.

El código asociado a una instrucción `reify` será evaluado en el espacio computacional del intérprete en lugar ejecutarse en el contexto de la aplicación. Se produce un salto real en la torre de teórica de intérpretes propuesta por Smith [Smith82]. El resultado es que, en la codificación de aplicaciones, es posible especificar, aumentar o modificar las características del lenguaje de programación, como si nos encontrásemos diseñando éste.

El ejecutar un código en el contexto computacional de su intérprete nos permite:

- Conocer el estado del sistema (objetos, clases, variables...): introspección.
- Acceder y modificar la estructura de sus objetos: reflectividad estructural.
- Modificar y aumentar la semántica de su lenguaje de programación: reflectividad computacional o de comportamiento.
- Modificar la sintaxis del lenguaje por la propia aplicación: reflectividad de lenguaje.

Un breve ejemplo de las tres primeras características se muestra en el archivo "`musimApp1.na`", entregado en el directorio de pruebas de la distribución del prototipo.

### A.3.4 Reflectividad de Lenguaje

La tercera y última forma de especificar un lenguaje de programación en nuestro prototipo es mediante la modificación de un lenguaje existente, previamente definido por alguno de los dos mecanismos ya mencionados. Una vez identificado el lenguaje a utilizar, utilizando el lexema "`+`", el programador puede ubicar código Python a ejecutar en el contexto de intérprete, de igual que la instrucción `reify` descrita en el punto anterior.

El código descrito se evaluará antes de la ejecución de la aplicación, significando una personalización del lenguaje de programación para la ejecución de una determinada aplicación: el lenguaje no se modifica para todo el sistema, sino que es amoldado a la aplicación concreta.

El modo en el que este código actúa para modificar la especificación de un lenguaje es accediendo al conjunto de reglas que describen el lenguaje y, por medio de la utilización de reflectividad estructural, modificar éstas para obtener la personalización del lenguaje. Un ejemplo de esta posibilidad está codificado en el archivo "`printApp.na`" del prototipo distribuido.

## A.4 Interfaz Gráfico

Para facilitar la programación de lenguajes y aplicaciones en nuestro sistema, hemos desarrollado un pequeño interfaz gráfico para nuestro prototipo. Este interfaz está construido sobre el estándar TK [Lundh99] independiente de la plataforma a utilizar.

### A.4.1 Intérprete de Comandos

La ventana principal del sistema tiene el siguiente aspecto:

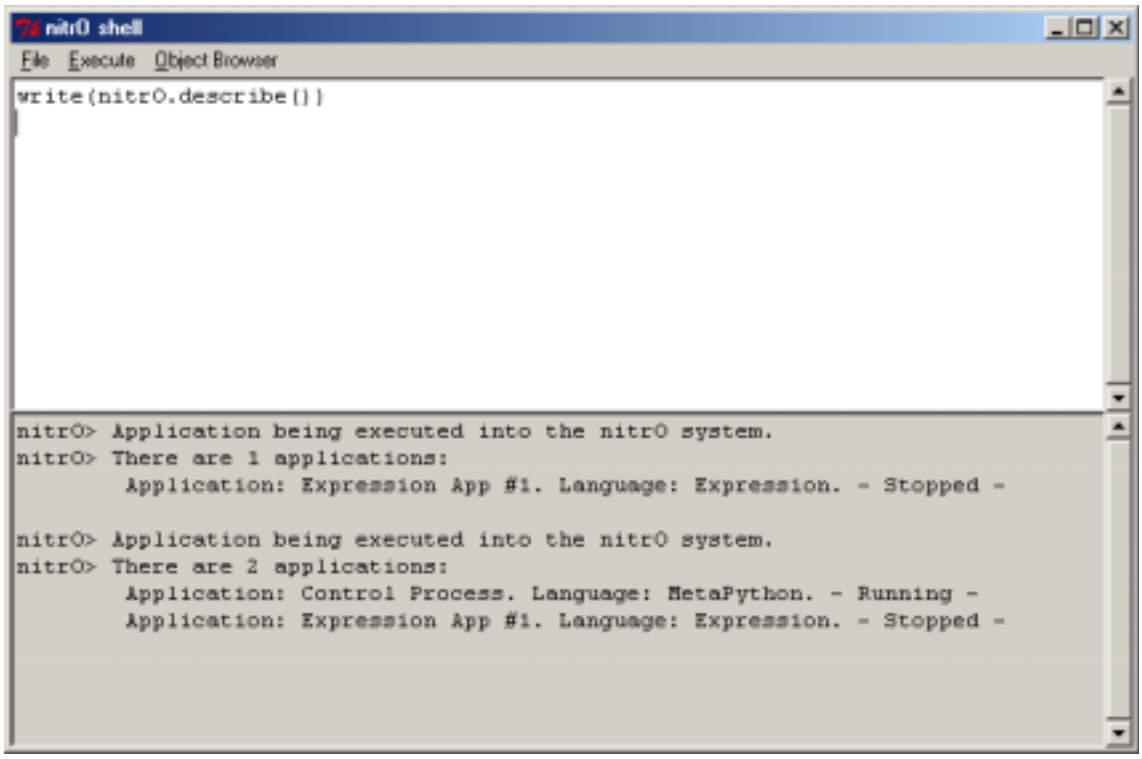


**Figura 1: Ventana principal del prototipo.**

La ventana está dividida en tres partes:

4. Menú del sistema.
5. Editor de código del intérprete de comandos.
6. Ventana de salida, en la que se visualiza el resultado de ejecutar los comandos especificados en el editor.

El usuario del sistema describe los comandos deseados en el editor y evalúa éstos seleccionando la opción `Execute` del menú del sistema. La principal cuestión que puede preguntarse el usuario es ¿qué comandos poseo para acceder al sistema?

El código susceptible de ser ejecutado es cualquier programa Python; éste será evaluado en el contexto de ejecución de nuestro sistema. Además, este contexto tendrá dos elementos añadidos para acceder al sistema:

7. La función `write`, que recibe cualquier parámetro y lo muestra en la ventana de salida. Será utilizado cuando deseemos conocer algún valor. Por ejemplo, en la Figura 1 se muestra la cadena de caracteres que describe el sistema.
8. El objeto `nitrO`, que nos da acceso a todos los elementos del sistema. Ofrece un conjunto de objetos (lenguajes y aplicaciones) así como un conjunto de métodos para trabajar sobre éstos. El conjunto de objetos y métodos existentes podrá consultarse dinámicamente, gracias al carácter introspectivo del sistema, mediante el *Object Browser* (§ A.4.3).

### A.4.2 Archivos Empleados

En la utilización nuestro sistema podemos diferenciar tres tipos de archivos:

9. Archivos de especificación de lenguajes de programación codificados mediante nuestro metalenguaje. Estos archivos poseen la extensión `ml`, siguen el metalenguaje descrito en § A.2.1, y deben ubicarse en el directorio del sistema.
10. Archivos de aplicación. Poseen la extensión `na` (*nitrO application*) y describen aplicaciones del sistema en un determinado lenguaje de programación.
11. Archivos *script* o comandos. La extensión `ns` (*nitrO script*) identifica una secuencia de sentencias a ejecutar por el intérprete de comandos del sistema.

El tratamiento del último tipo de archivos se lleva a cabo mediante el menú de archivo (*File*) de nuestro prototipo.

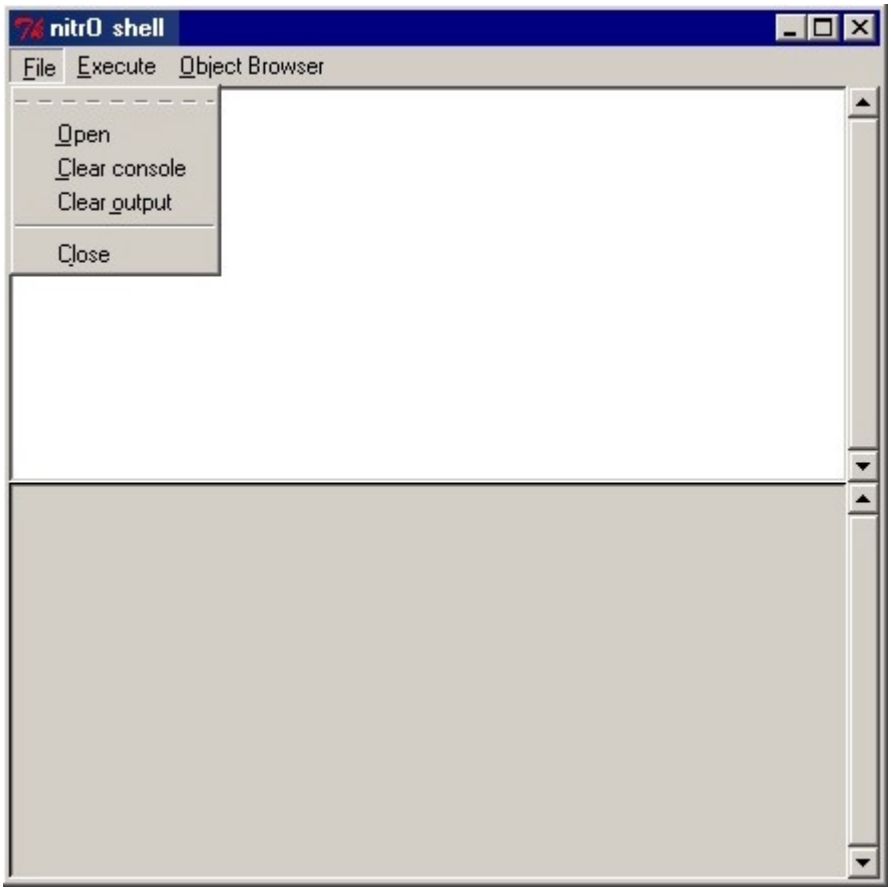


**Figura 2: Menú archivo del prototipo.**

Mediante este menú, pueden cargarse archivos de *script*, así como limpiar el editor y la ventana de salida y finalizar la ejecución del sistema.

### A.4.3 Introspección del Sistema

Todo el acceso al sistema se obtiene a través del objeto `nitrO`, que ofrece un conjunto de atributos y métodos descriptores del sistema. Sin embargo, para utilizar éstos, debemos conocer dinámicamente su existencia y funcionalidad. Para ello tenemos dos herramientas en el sistema:

12. Métodos `describe`: La mayor parte de los objetos del sistema implementan un método `describe` que nos muestra su descripción. Pasándoles este mensaje y escribiendo en la consola el resultado de su invocación –mediante la función `write`– obtendremos información dinámica de su estado.

Un ejemplo de esta utilización se muestra en la Figura 1.

13. *Object Browser*: Esta última opción del menú nos muestra el conjunto de todos los atributos y métodos del objeto `nitrO`, y por lo tanto de todo el sistema. Haciendo uso de esta herramienta, el programador podrá conocer el conjunto de aplicaciones y lenguajes existentes, su estructura y sus mensajes, y podrá programar y modificar el sistema en función de la información consultada.

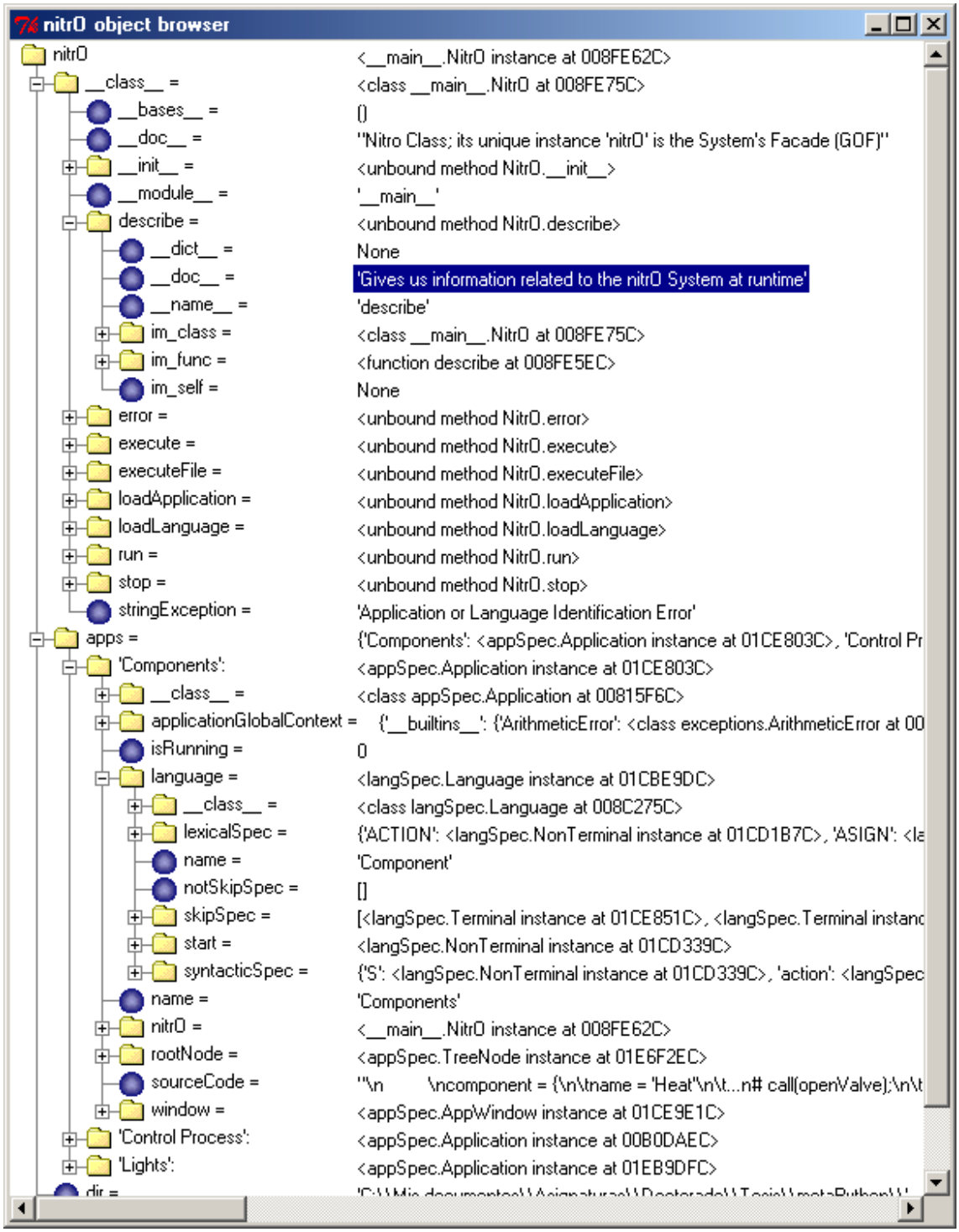


**Figura 3: Estado del sistema en tiempo de ejecución, mostrado por el *Object Browser*.**

En la Figura 3 se muestra la información dinámica del sistema. A modo de ejemplo, comentaremos parte de ésta:

- Accediendo al atributo `__class__` de `nitrO`, obtenemos todos los métodos de éste (`execute`, `executeFile`...) así como la descripción de la clase y cada uno de los mensajes, consultando sus atributos `__doc__`.
- El atributo `apps` es una lista de las aplicaciones existentes en el sistema (`Component`, `ControlProcess` y `Lights`).
- Cada aplicación posee un conjunto de atributos y métodos (su clase) que nos ofrecen información y funcionalidad de ésta. Uno de estos atributos es siempre la especificación de su lenguaje de programación mediante una estructura de objetos –atributo `language`.
- Por cada lenguaje de programación tenemos un conjunto de reglas léxicas y sintácticas libres de contexto (`lexicalSpec` y `syntacticSpec`), así como los componentes léxicos a descartar y a reconocer automáticamente (`skipSpec` y `notSkipSpec`).

Como se aprecia en este ejemplo, la información del sistema es elevada y compleja y, por tanto, una herramienta introspectiva como el *Object Browser* facilita el trabajo al programador.

### A.4.4 Ejecución de Aplicaciones

Cada vez que se ejecuta una aplicación en nuestro sistema, se crea un ventana gráfica para ésta. La utilización de la función `write` por esta aplicación, supone la visualización de la información pasada como parámetro en su ventana asociada.

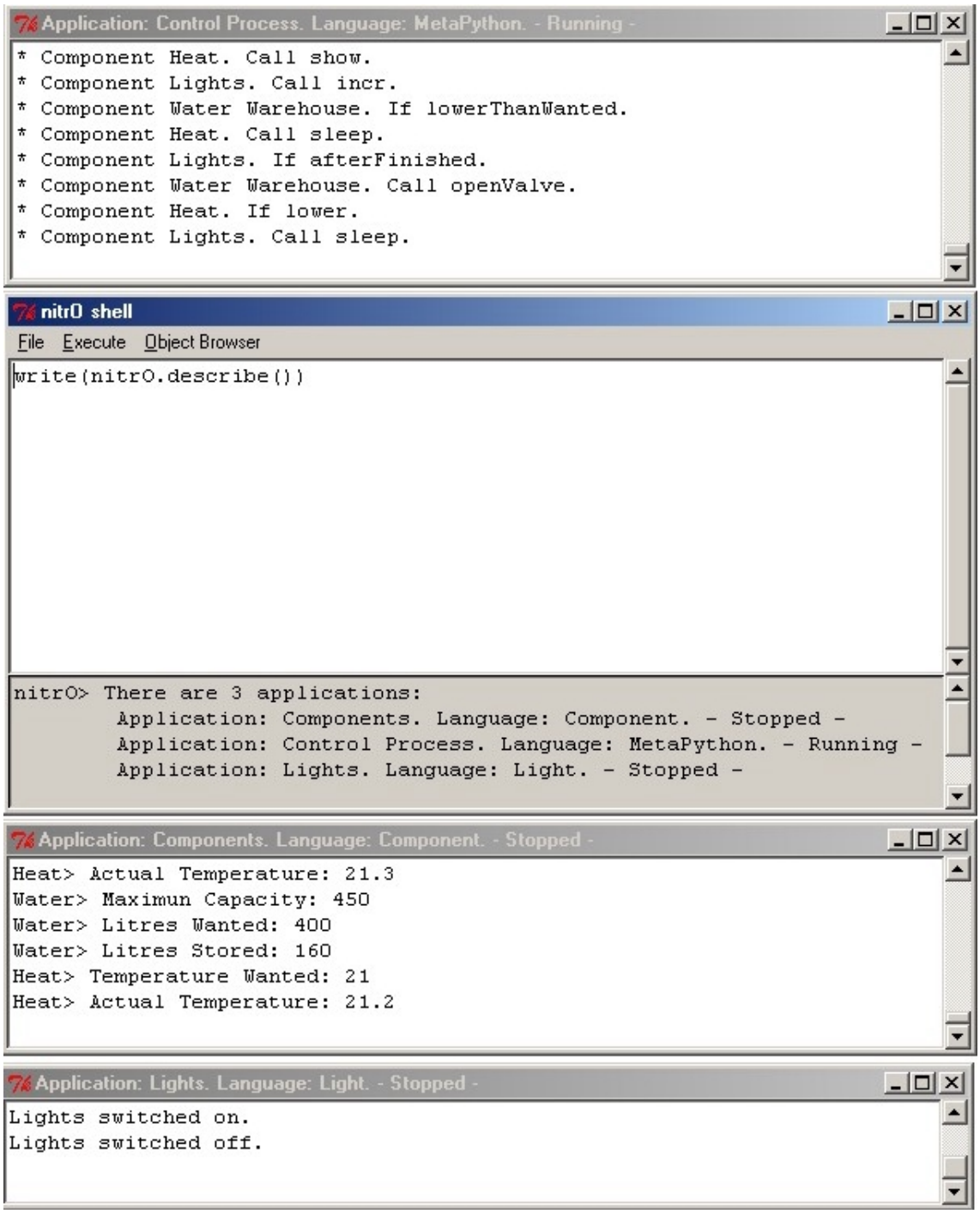


**Figura 4: Ejecución de aplicaciones en el sistema.**

En la figura anterior se muestra cómo el sistema está ejecutando 3 aplicaciones, cada una con su propia ventana de visualización. Una aplicación puede cerrarse cuando esté parada (al final de su título aparezca *Stopped*). El hecho de cerrar su ventana supone la eliminación de la aplicación asociada dentro del sistema.

Si deseamos finalizar una aplicación en ejecución, deberemos, desde el editor de comandos, enviar un mensaje `stop` al objeto `nitrO`, pasándole como parámetro el identificador de la aplicación. El título de ésta mostrará cuándo esté tratando de finalizar (*trying to stop*) y finalmente su estado de parada (*stopped*).

Cerrar la ventana principal del sistema supone cerrar el conjunto de aplicaciones existentes.