UNA POSIBLE SEMÁNTICA OPERACIONAL DE ALTO NIVEL PARA LENGUAJES DE GENERACIÓN DE CONTENIDO DINÁMICO EN EL CONTEXTO DE APLICACIONES WEB

por
Ignacio Gallego Sagastume

Presentado como parte de los requerimientos
para el grado de
LICENCIATURA EN INFORMÁTICA
UNIVERSIDAD NACIONAL DE LA PLATA
LA PLATA, BUENOS AIRES, ARGENTINA
MARZO DE 2002

UNIVERSIDAD NACIONAL DE LA PLATA
FACULTAD DE INFORMÁTICA

Los abajo firmantes certifican que han leido el trabajo de grado denominado "**Una posible semántica operacional de alto nivel para lenguajes de generación de contenido dinámico en el contexto de aplicaciones web**" por Ignacio Gallego Sagastume.

Fecha: **Marzo de 2002**

Director: _____
Claudia Pons

Codirector: _____

Revisión: _____

UNIVERSIDAD NACIONAL DE LA PLATA
FACULTAD DE INFORMÁTICA

Fecha: **15 de octubre de 2002**

Autor:	**Ignacio Gallego Sagastume**
Legajo:	**01686/0**
Dirección:	**59 nº656, p.12º dpto. "E"** **CP:1900, La Plata.**
Teléfono:	**422 1927**
E-Mail:	**nick@lpsat.com**
Título del trabajo:	**"Una posible semántica operacional de alto nivel para lenguajes de generación de contenido dinámico en el contexto de aplicaciones web"**
Director:	**Claudia Pons**
Codirector:	
Comienzo:	**Marzo de 2002**
Plazo de ejecución estimado:	**6 meses**
Clasificación:	**Investigación Teórica**

Firma del autor

A mis padres Eliseo y Marta,
mis hermanas Juana y Clara,
y mis amigos Ignacio y Ramiro.

"It may take one lifetime or it may take one day,
for me to find the courage, not to walk away..."
Joey Tempest.

Contenido

Resúmen

En este trabajo se hace un análisis semántico (ver [Hen90] para introducción al tema, o [AC96] para ver definiciones alternativas y uso de este tipo de semánticas en teorías más avanzadas) de un conjunto de lenguajes seleccionados entre varios de interés en la actualidad, de los denominados de "generación de contenido dinámico" o de "generación dinámica de páginas" (para brevedad, de ahora en adelante *lenguajes gcd*). Una "posible semántica de alto nivel" es debido a que se hará abstracción de muchos detalles de implementación y además porque podrían existir otras definiciones semánticas compatibles para el mismo lenguaje (más precisas o mas cercanas a la implementación por ejemplo).

Esto permitirá al lector idóneo (estudiante de grado de ciencias de la computación, o programador usuario del lenguaje, por ejemplo) entender claramente como funcionan los mecanismos de interpretación, compilación, y generación de los documentos que se obtienen como resultado de ejecutar un programa escrito en una familia de lenguajes gcd, como así también la semántica de las distintas instrucciones y construcciones sintácticas.

Para tal propósito, se define un *formalismo* (un *dominio semántico* y una relación de reducción sobre él) que permitirá expresar características comunes de lenguajes gcd y comprender mejor su funcionamiento y sus diferencias.

Este formalismo teórico pretende ser lo suficientemente **preciso** para no generar ambigüedades en las fórmulas propuestas, pero a la vez lo suficientemente **simple**, práctico y legible para permitir una aplicación directa en grupos de desarrollo o ambientes en donde se utilicen lenguajes gcd. Podría utilizarse esta notación para dotar al grupo humano de trabajo de una base teórica común para facilitar la compresión de una nueva tecnología a utilizar, como así también definir un esquema o marco de trabajo, facilitar la comunicación, permitir que los módulos sean independientes del programador (por lograr uniformidad de estilos de programación), potenciar la eficiencia del trabajo en equipo, la coherencia de resultados, y otros posibles beneficios esperados que se discutirán a lo largo del presente documento.

Se discutirá también su posible utilidad y viabilidad de aplicación.

Introducción

Los lenguajes de generación de contenido dinámico (lenguajes gcd) surgen como respuesta a una rápida evolución en las aplicaciones web y la necesidad de generar dinámicamente documentos con contenido extraído de diversas fuentes, como bases de datos o requerimientos de consultas de usuarios en tiempo real. Estos, son básicamente *lenguajes de scripting*, esto es, lenguajes de aplicación general con gran potencia, flexibilidad, de fácil aprendizaje y capaces de relacionarse con una diversidad de tecnologías de distintas compañías. Entre estos se encuentran, entre otros y solo por nombrar algunos de los dialectos presentes en internet hoy en día, *VBScript, JScript, JavaScript, asp, php, perl, jsp y servlets*, que combinados con diferentes tecnologías de componentes, librerías gráficas e interfaces para utilizar bases de datos comerciales, componen una completa gama de herramientas para programación.

Se definirá un formalismo con el fin de capturar lo más precisamente la semántica de las principales construcciones sintácticas de estos lenguajes, con el objetivo de obtener un conjunto de reglas que servirá para programadores involucrados en proyectos de desarrollo de aplicaciones web, con el fin de lograr los objetivos antes mencionados. Se espera también que el conjunto de reglas propuestas para cada lenguaje evolucione de acuerdo a la experiencia recogida en cada proyecto hacia un marco de trabajo utilizando lenguajes de scripting gcd sobre aplicaciones web.

Se hace primero una breve introducción a los distintos tipos de semánticas que pueden estudiarse dado un determinado lenguaje. Luego se define el dominio semántico base para el estudio de lenguajes gcd, y en posteriores capítulos se aplican estas definiciones y se estudian en el marco de los lenguajes ASP y PHP.

Luego se examina el contexto y se propone un marco (o metodología) de trabajo para utilizar este tipo de lenguajes.

Capítulo 1

Semántica de lenguajes de programación

WORDS

When violets were springing
And sunshine filled the day,
And happy birds were singing
The praises of the May,
A word came to me, blighting
The beauty of the scene,
And in my heart was winter,
Though all the trees were green.

Now down the blast go sailing
The dead leaves, brown and sere;
The forests are bewailing
The dying of the year;
A word comes to me, lighting
With rapture all the air,
And in my heart is summer,
Though all the trees are bare.

John Hay (1838-1905)

1.1. Introducción

La palabra **semántica** se refiere al estudio de la significación de las palabras. En particular, cuando estudiamos la semántica de un determinado lenguaje de programación, hacemos referencia al **significado de las sentencias** o instrucciones del lenguaje; esto equivale a definir o asignar

de alguna manera las **acciones atómicas en lenguaje de máquina** que tienen efectos directos sobre el hardware a partir de cada instrucción o construcción del lenguaje de programación (ver semánticas concretas más adelante).

La **manera en que se describe la semántica** de un lenguaje de programación puede variar de acuerdo al enfoque y objetivos que se pretenda alcanzar con dicha descripción; puede hacerse mediante un manual que describa los efectos de las instrucciones del lenguaje estudiado en **lenguaje natural**, tal como en una traducción. También podemos utilizar una **notación formal** (o quasi-formal) definiendo una máquina virtual para simular o representar el hardware sobre el que se ejecuta el lenguaje, y de esta manera evitar las ambigüedades del lenguaje natural.

La pregunta que surge inmediatamente es: ¿porqué querríamos describir formalmente la semántica de un determinado lenguaje de programación? ¿Qué es lo que diferencia un programador que aprende un determinado lenguaje mediante **prueba y error**, de otro que antes de programar primero **lee** detenidamente **el manual de usuario** y otro que ha estudiado su semántica? ¿Qué ventajas tiene estudiar semántica formal?

La diferencia es la **exactitud y profundidad** del entendimiento de los algoritmos que se utilizan para resolver problemas. No es posible tener un entendimiento verdadero de un lenguaje de programación sin un **modelo mental** de su semántica.

Mientras que el programador "hacker" desarrolla este modelo mental imprecisamente y tiene una vaga percepción del **manejo de memoria, complejidad e implementación** de las sentencias del lenguaje que utiliza, el programador que lee el manual si conoce estos detalles aunque con las **ambigüedades** del lenguaje natural y con las **imprecisiones** y **defectos** de las definiciones desarrolladas por el equipo de documentación en el manual; el programador que estudia semánticas formales tendrá mayor precisión en el conocimiento de todos estos detalles.

La ventaja es que cuando comprendemos como están implementadas las distintas instrucciones de un lenguaje, incrementamos nuestra **habilidad para escribir** de manera eficiente **programas eficientes**, por conocer las acciones resultantes de una determinada operación y por lo tanto, el costo de las mismas. Además, es difícil pensar en términos de elementos que no admiten expresiones equivalentes en palabras; por esto es bueno definir **elementos abstractos** que representen estos

elementos y constituyan las bases de nuestros razonamiento. Uno de los objetivos de definir formalmente la semántica de un lenguaje es el de obtener un **marco de trabajo intelectual** para sus usuarios. Como veremos en las siguientes secciones, la semántica formal también puede servir para **probar equivalencia** de dos o más programas.

Estas son algunas de las ventajas que obtiene un usuario sobre definiciones semánticas existentes; estas definiciones pueden haber sido **construidas para un lenguaje existente** o pueden **desarrollarse como parte de la documentación** al realizarse un compilador para un determinado lenguaje. Este escenario sería el ideal para que un lenguaje sea bien comprendido, utilizado e implementado (se especifica su semántica formalmente en base a un conjunto de instrucciones de máquina y luego puede implementarse para distintas plataformas).

A continuación se hace una breve introducción a las distintas visiones que se utilizan para estudiar semántica de un determinado lenguaje, y qué se puede lograr con cada una de ellas. Para una información más detallada sobre introducción a los distintos tipos de semántica puede recurrirse a [Hen90].

1.2. Semánticas Concretas (basadas en pila)

En este tipo de semánticas se define el comportamiento de un lenguaje en base a una máquina basada en pila. Se tiene dependencia del hardware o al menos de la arquitectura de instrucciones del microprocesador donde vaya a ejecutar un lenguaje de programación. Se define un lenguaje de máquina simple (real o abstracto) que sirve para dar semántica a las instrucciones del lenguaje, y además para la construcción de intérpretes y compiladores.

La definición de instrucciones se hace en base a la máquina basada en pila (**stack-machine**) y se definen las acciones básicas que se efectúan sobre la pila (*push*, *pop*, *applyStack*, *etc.*).

Un programa para la stack-machine es una secuencia de instrucciones que manipulan la pila. Entonces, puede definirse un compilador como una función (puede hacerse esto mediante inducción estructural) del conjunto Exp de expresiones válidas del lenguaje en el conjunto de secuencias de instrucciones $SeqProg$:

$$
\begin{aligned}
&compiler && :: Exp \to SeqProg \\
&compiler(n) && = push(n) \\
&compiler(e\ op\ e') && = comp(e) \oplus comp(e') \oplus applyStack(op)
\end{aligned}
$$

De esta manera, $prog = compiler(E_1)$ denota el programa (secuencia de instrucciones) para la máquina a pila que resulta de compilar la expresión E_1.

La semántica concreta es la de más bajo nivel, la que más depende del hardware subyacente, y por eso es útil para estudiar implementaciones de un determinado lenguaje y definir compiladores e intérpretes.

1.3. Semánticas de Evaluación

Describe como evaular una expresión del lenguaje mediante una definición inductiva. El objetivo es **capturar la escencia de la evaluación en un nivel abstracto**. Esto puede ayudarnos a probar propiedades de la evaluación, tales como:

- si $e \Rightarrow n$ y $e \Rightarrow m$, entonces $n \equiv m$
- $\forall e \in Exp, \quad \exists n \in Num\ tq. \quad e \Rightarrow n$

En esta semántica, no se muestra nada acerca de cuáles son los pasos que se siguieron para llegar a un determinado resultado, como por ejemplo en: $(10 - 8) \Rightarrow 2$. Nos dice cuál es el resultado obtenido, pero no nos dice cómo se llegó a él. Para eso, se define la semántica computacional.

1.4. Semántica Operacional

En el enfoque computacional u operacional, se estudian reglas de reducción para transformar estados de programas. Los estados de los programas estan dados por las distintas "instantáneas" de la memoria de la máquina, en cualquier momento durante la ejecución de un programa. Una determinada regla, especifica **cómo** puede transformarse un estado en otro, mediante la ejecución de una acción determinada realizada en un programa. En vez de definir la evaluación $\Rightarrow$ como en caso anterior, se define la relación $\to$, que define un paso de reducción; se muestran todos los pasos

intermedios para llegar a un resultado de una computación. De esta manera, la relación $\rightarrow$ puede verse como un refinamiento de $\Rightarrow$ porque puede definirse:

$$e \Rightarrow n \;\; si \;\; e \rightarrow^* n$$

donde $\rightarrow^*$ representa la clausura transitiva de $\rightarrow$.

Mediante este enfoque se pretende captar el comportamiento dinámico de un lenguaje. Un objetivo útil que justifica la definición de semántica computacional de un lenguaje es la construcción de intérpretes para el mismo.

Estudiaremos el enfoque computacional de lenguajes de scripting mediante notaciones similares a [Gog98] o [CF97].

1.5. Semántica Denotacional

En la semántica denotacional, un lenguaje de programación se define en base a una función de valuación. Se ven las expresiones de un lenguaje como representaciones de objetos abstractos. Se provee de una interpretación matemática para las expresiones, pudiendo de esta manera definir la semántica de los elementos del lenguaje dentro de un dominio conocido. Por ejemplo, si las expresiones de un lenguaje representan números naturales, la función $I : Num \rightarrow \mathbb{N}$ de interpretación mapea la representación de los números al conjunto de números naturales. La notación utilizada es la siguiente:

$$\begin{array}{ll} e \in Exp & \\ \llparenthesis \rrparenthesis & :: Exp \longrightarrow \mathbb{N} \\ \llparenthesis n \rrparenthesis & = n \\ \llparenthesis e \;\; op \;\; e' \rrparenthesis & = op_I(\llparenthesis e \rrparenthesis, \llparenthesis e' \rrparenthesis) \end{array}$$

Con la semántica denotacional puede razonarse acerca de las propiedades de las expresiones del lenguaje, y también puede comprenderse la lógica interna. Se especifica **qué** debe hacerse, pero no cómo.

También pueden demostrarse propiedades (teoremas) acerca de las expresiones del lenguaje, como por ejemplo, las dos propiedades "e *reduce* a n si y sólo si e *denota* a n" y "toda expresión es

denotada por un natural":

$$\forall e \in Exp, \quad e \Rightarrow n \Leftrightarrow \llparenthesis e \rrparenthesis = n$$
$$\forall e \in Exp, \quad \exists k \in \mathbb{N} : \llparenthesis e \rrparenthesis = k$$

Capítulo 2

Descripción del dominio semántico

IN MEN WHOM MEN CONDEMN

"In men whom men condemn as ill
I find so much of goodness still,
In men whom men pronounce divine
I find so much of sin and blot,
I hesitate to draw the line
Between the two, where God has not."

Joaquin Miller (1841-1913)

2.1. Introducción

En este capítulo se describen los elementos que que conformarán el **dominio semántico** donde se estudidará el comportamiento de lenguajes gcd.

Estos elementos, junto con las reglas para relacionarlos (definición dede las relaciones de reducción), conformarán una descripción casi completa de una **máquina virtual abstracta** capáz de interpretar un lenguaje gcd y generar una salida como resultado. Servirá mayormente como especificación formal de la semántica del lenguaje.

Debe tenerse en cuenta que se hace una descripción abstracta a un nivel muy alto con los fines de comprender como se procesan las instrucciones de un lenguaje y por lo que no se tendrán en cuenta muchos detalles de implementación de esta máquina.

2.2. Máquina Abstracta para GCDLs

Se describen en esta sección los elementos que representan las distintas componentes comunes de los lenguajes gcd y sobre los cuáles se definirán las relaciones de reducción $\rightsquigarrow_{ASP}$ y $\rightsquigarrow_{PHP}$, como veremos en los capítulos siguientes. Sobre ellos se harán conjeturas y se construirán las reglas para poder describir la semántica de cada lenguaje. Estos elementos junto con las reglas conforman el modelo abstracto de máquina sobre la cual se procesan los scripts del lenguaje.

Existe una modelización de la **entrada**, que en el caso real la constituyen los archivos físicos de texto donde se encuentra el código fuente en lenguaje HTML y lo que llamaremos **secciones de código**, una característica común de lenguajes gcd. Cada archivo físico se procesa en base a estas secciones de código, para lo cual deberemos tener en cuenta el **contexto** de aplicación o reducción de una sección de código.

El contexto de ejecución se representa mediante tres componentes, que son, el server o host donde reside el reside el archivo que se está procesando, el nombre (o un identificador) del mismo archivo y la porción de texto (secuencia de caracteres) dentro de este archivo que falta procesar en un determinado instante de la ejecución.

También existe una modelización para la **memoria** y para la **salida** producida por la interpretación del script. La memoria está compuesta por el **ambiente de trabajo**, que contiene los nombres de variables u objetos disponibles en el **almacén**.

Veremos a continuación estos elementos en más detalle.

2.3. Sección de código

Las relaciones de reducción procesarán **secciones de código**, que son fragmentos de código encerrados entre delimitadores específicos de cada lenguaje. Los símbolos más comunmente utilizados son `<%` para apertura y `%>` para cierre.

Dentro de un mismo script de una determinada aplicación pueden encontrarse secciones de código de distintos lenguajes. El siguiente ejemplo graficará:

```
linea 1: <% strGreeting = 'Hello World!' %>
linea 2: <P><B>This is the Hello World program in ASP</B></P>
linea 3: <HR>
linea 4: <%= strGreeting %>
```

Las líneas 2 y 3 son código HTML. Una imprime una frase en negrita y la otra coloca una línea horizontal. Las líneas 1 y 4 son código ASP (por ejemplo); se define una variable de tipo string y luego se imprime en la salida.

En las reglas se utilizarán los símbolos $\trianglelefteq \mathtt{S} \trianglerighteq_{\mathtt{ASP}}$ para denotar que los caracteres de la secuencia S se encuentran dentro de una sección de código ASP (análogamente para $\trianglelefteq \mathtt{S} \trianglerighteq_{\mathtt{PHP}}$ para PHP y $\trianglelefteq \mathtt{S} \trianglerighteq_{\mathtt{HTML}}$ para HTML). Esto no significa que S contenga todos los caracteres que aparecen en esa sección de código, sino que esos caracteres deben ser interpretados como código ASP (correspondientemente PHP y HTML).

2.4. Modelización de entrada y contexto de procesamiento

La **entrada** (in) para el procesamiento o interpretación de un script la constituye un determinado archivo o conjunto de archivos almacenados en un directorio de un servidor web. Se accederá a ellos mediante una referencia desde otro archivo en la web (que puede ser un hipervínculo o una invocación desde otro archivo), ya sea externa o interna (es decir, del mismo servidor o no), o también ingresando la dirección completa del archivo en un navegador web.

Como ya mencionamos, con **contexto de procesamiento** nos referimos al archivo que se esta procesando en el momento de aplicar una regla, como así también que parte se procesó y que parte del archivo aún no, y el servidor donde está ubicado dicho archivo. Es necesario conocer este contexto, pues como veremos más adelante existen instrucciones para cambiar el contexto de procesamiento y alterar de esta manera el flujo normal de ejecución. Estos cambios pueden ser difíciles de rastrear si tenemos en cuenta que estos saltos pueden condicionarse de acuerdo al estado de las variables en

un determinado momento o a los datos ingresados por un usuario.

Para indicar el contexto de aplicación de una regla, se utilizará una tripla con la siguiente notación:

$$[H, archivo.ext, S]$$

donde H es un identificador de servidor donde se encuentra el archivo con nombre "archivo.ext" que está siendo procesado actualmente y S es el texto (secuencia de caracteres) que falta procesar dentro del mismo archivo.

2.5. Environment y Store

El **ambiente de programación** (o environment) es el entorno sobre el cual se trabaja; representa una zona de memoria donde se almacenan definiciones de variables con referencias a objetos de datos y tipos. Se lo representará con un conjunto de triplas de la forma:

$$[x : x^{\frown} :: Type]$$

en donde x es un identificador de variable (o función) válido, la segunda componente $x^{\frown}$, es una dirección (referencia o puntero) al lugar del store en donde se almacena el valor de la variable y $Type$ es el tipo del objeto almacenado. Este será un tipo regular simple (entero, cadena, etc.), tipo función $FunType$ o un tipo objeto de los soportados por el lenguaje (nativo, Java, COM, etc). De acuerdo a la información del tipo se accederá a su estructura interna.

El **almacén** (o store), es conceptualmente la sección de datos de la memoria; en él se almacenan los valores de las variables y los objetos creados por el usuario, como también los objetos manipulados por el lenguaje (como es el caso de ASP). Se compone un número finito de celdas consecutivas de dimensión fija enumeradas de 0 a n, siendo n el tamaño de la memoria. La unidad mínima de alocación es de una celda, y para tipos de datos más complejos se alocarán en grupos, en principio consecutivos. Se representa en el cálculo mediante un conjunto de pares:

$[a :< o >]$

en donde a es la dirección de la memoria mediante la cual se accede al objeto de datos $< o >$.

2.6. Secuencias y conjuntos

Se utilizanrán **secuencias** para modelar los objetos de la máquina virtual en donde importa el orden en que se almacenen sus componentes internos. Podemos pensar en las secuencias como un tipo de datos abstracto junto con operaciones como pueden ser agregar un elemento en una determinada posición, chequear pertenencia de un elemento en una secuencia y unir o combinar dos secuencias en el mismo orden.

La entrada como también la salida se modelan como una secuencia, porque es necesario que los caracteres procesados estén en un orden específico para poder interpretarse. Para los casos del environment y el store, si bien existe un orden implícito ya que ambas entidades modelan partes de una memoria física, en general podrá trabajarse con ellos como si se tratara de un conjunto en donde las únicas operaciones que se utilizan son las de agregado, pertenencia y actualización de un elemento sin importar demasiado en dónde se encuentre el mismo.

2.7. Salida

A medida que se interpreta un archivo ASP, se va generando una **salida** (out) que depende, como ya hemos mencionado, del estado de las variables locales al archivo, como también de la entrada del usuario.

Se modelará la salida con secuencias de caracteres y se utilizarán las operaciones disponibles sobre las mismas.

2.8. Operaciones

Se describen en esta sección las operaciones utilizadas en las reglas, para expresar los cambios que se van efectuando en el contexto de procesamiento, la memoria y la salida generada a medida

que se procesa un script.

2.8.1. Pertenencia

Se utilizará el símbolo $\in$ para indicar la operación de pertenencia de un elemento a una secuencia o conjunto. Por ejemplo,

$$x \in E$$

denota una expresión verdadera si el elemento x se encuentra en la expresión E, ya sea esta una secuencia o un conjunto (falsa en caso contrario).

2.8.2. Agregado de un elemento

El símbolo $\Leftarrow^+$ se utilizará para indicar el agregado de un elemento en un conjunto o en algún lugar de una secuencia. Por ejemplo,

$$Env \Leftarrow^+ [\text{"}nombre\text{"} : 45 :: String]$$

denota un ambiente que se obtiene del ambiente Env agregando el identificador "$nombre$", con su correspondiente tipo y puntero al objeto de datos real (dirección con numeración 45 del store). Para esta operación en particular, se debe pedir como precondición que el identificador agregado no se encuentre ya en el ambiente, ya que un ambiente bien formado no puede tener identificadores repetidos.

En caso de querer agregar un elemento en una determinada posición de una secuencia, podrá subindicarse la posición, como por ejemplo en la siguiente expresión:

$$S \Leftarrow_i^+ c$$

que denota el agregado del elemento c en la posición i de la secuencia S.

2.8.3. Derreferencia

El ambiente puede interpretarse también como un diccionario de identificadores con referencias (o análogamente una función de identificadores en referencias). Para acceder al elemento de la memoria identificado por "var" podríamos referirnos a

$$Store["var"]$$

que buscaría el identificador en el ambiente y luego accedería a la dirección indicada para el mismo, sabiendo también el tipo del objeto al que se accede. Como el store también puede verse como una función cuyo dominio es cierto subconjunto de números naturales, también podríamos utilizar la notación $Store[n]$ para denotar el elemento almacenado en la posición n.

2.8.4. Actualización

La operación $\Leftarrow^{\checkmark}$ se utilizará para actualizar un elemento dentro del environment o store. En el caso del environment, se proveerá el identificador para saber que elemento se quiere actualizar, y en el caso del store, se proveerá la dirección del store donde se encuentra. Por ejemplo:

$$Store \Leftarrow^{\checkmark} [d : v]$$

denota que el store que se obtiene a partir de $Store$ actualizando la dirección o posición d para que contenga el valor v.

2.8.5. Unión, concatenación o combinación secuencial

Se utilizará $S \oplus Q$ para denotar la secuencia que contiene primero todos los elementos de S y luego los de Q en el mismo orden en el que aparecían en S y en Q.

Si S y Q son dos conjuntos no ordenados, $S \oplus Q$ denota la unión usual de conjuntos.

2.8.6. Acceso a objetos

Cuando necesitemos acceder a componentes internas de un objeto de datos almacenado en el store del que conocemos su estructura interna, utilizaremos la notación

$< obj > .prop$

para acceder a la componente llamada "prop" del objeto $< obj >$ almacenado en alguna locación del store.

2.8.7. Funciones de delegación

Para expresar que alguno de los elementos del dominio semántico debe ser modificado de alguna forma que no se quiere o puede describir o explicitar, o procesado de una manera compleja, se delegará el comportamiento en funciones simplemente invocándolas. En cada caso se dará una especificación de lo que debe hacer la función y de quién es la responsabilidad de implementarla, si es necesario. Algunos ejemplos podrían ser:

$removeOldDefs(Env)$

para indicar que se eliminan ciertas definiciones del ambiente,

$removeDangling(Env)$

para eliminar inconsistencias en la memoria, o
$searchInaccesibleObjects(Store)$

para recolectar en una colección elementos inaccesibles del store. Con estas funciones se modularizan y simplifican las reglas semánticas. Existen algunas funciones de delegación que se definen en la siguiente sección como operaciónes sobre la memoria.

2.8.8. gc() y newCap()

La función *newCap*() toma un tamaño (en celdas de memoria) y devuelve la dirección de inicio del bloque libre.

La función *gc*() marca los elementos no referenciados del store como libres.

La ejecución de *gc*() permitirá que la función *newCap*() tenga mas probabilidades de encontrar un lugar de memoria vacío, y de retornar valores mas pequeños de direcciones del store; recordemos que en la práctica la memoria es finita.

2.9. Tuplas

La relaciones de reducción que representarán o definirán el comportamiento de un determinado lenguaje, serán relaciones binarias sobre un subconjunto de un universo de tuplas que contienen todos los elementos antes mencionados. Las relaciones que se definirán en los siguientes capítulos para los lenguajes ASP y PHP serán respectivamente $\rightsquigarrow_{ASP}$ y $\rightsquigarrow_{PHP}$.

Se define entonces sobre un espacio de tuplas:

$(Context \times Environment \times Store \times Output)$

Context es el conjunto de todos los pares $[H, F, S]$, en donde H es el identificador del host donde reside el archivo F y S es una secuencia de caracteres contigua dentro del archivo F.

Store es el conjunto de las posibles secuencias de pares $[d :< o >]$ en donde d es un número natural que representa la dirección donde se almacena el objeto $< o >$, arbitrariamente grande.

Environment es el conjunto de todos los ambientes válidos, incluyendo como caso particular el ambiente vacío: $\emptyset$. Un ambiente se considera válido cuando no contiene identificadores repetidos, las referencias existen dentro del rango de direcciones del store y los tipos utilizados son los antes mencionados.

Output es también un conjunto de las posibles secuencias de caracteres de salida que se enviarán como respuesta al requerimiento del archivo que origina el procesamiento y que serán renderizados por un navegador web (interpretados como código HTML).

2.10. Estructura de las reglas semánticas

La estructura de las reglas sobre las que se definirá la semántica de un determinado lenguaje gcd será la siguiente:

$$RuleName \frac{\begin{array}{l} Exp_1, \\ Exp_2, \\ \\ .. \\ \\ Exp_n \end{array}}{\begin{array}{l} ([H, F, S], Env, Store, Out) \rightsquigarrow_{Language} \\ ([H', F', S'], Env', Store', Out') \end{array}}$$

en donde $RuleName$ es el nombre de la regla, y $\rightsquigarrow_{Language}$ es la relación de reducción que está siendo definida para representar el proceesamiento del lenguaje $Language$. La parte superior de la regla describe en forma declarativa las expresiones que deben cumplirse (no importa el orden en que se tengan en cuenta), es decir, evaluarse a verdaderas en caso de ser precondiciones o simplemente ejecutarse en el caso de ser sentencias, para poder transformar la primer tupla del denominador en la segunda. Para realizar las transformaciones que llevan a obtener la tupla resultado pueden aplicarse las operaciones antes definidas o también llamarse a funciones de delegación, que se deben especificar en forma conjunta con la regla.

Por ejemplo:

$$AssUpd \frac{\begin{array}{l} [\text{"}x\text{"} : N :: T] \in Env_{JScript}, \\ (e, Env, Store) \Longrightarrow v :: T \end{array}}{\begin{array}{l} ([H, F, \trianglelefteq \texttt{x} = \texttt{e} \trianglerighteq_{\texttt{ASP}}], Env_{JScript}, Store, Out) \rightsquigarrow_{ASP} \\ ([H, F, \emptyset], Env_{JScript}, Store \Leftarrow^{\checkmark} [N : v], Out) \end{array}}$$

que muestra como debe procesarse una asignación. Al producirse la reducción, la entrada queda vacía y se actualiza en el store el valor de la variable x. Veremos distintos comportamientos para una sentencia de asignación en los distintos lenguajes.

2.10.1. Casos particulares y características especiales de elementos semánticos

Cuando el ambiente no tiene ninguna característica particular a destacar, se lo llamará por convención (en las reglas) *Env*. De la misma manera, se utilizará *Store* para denotar un store sin características relevantes y *Out* para una cualquier salida definida.

Cuando se necesite hacer explícito que en el ambiente o almacén existe alguna definición o dato, se utilizarán subindicaciones. Por ejemplo,

$$Env_{@ScriptingLanguage=JScript}$$

indicará que en el ambiente *Env* se asume que el lenguaje de script por defecto es *JScript*.

En los próximos capítulos se utilizarán todos los elementos definidos en este dominio semántico.

Capítulo 3

Semántica Operacional ASP

SPEECH AND SILENCE

The words that pass from lip to lip
For souls still out of reach!
A friend for that companionship
That's deeper than all speech!

Richard Hovey (1864-1900)

3.1. Introducción

Se introduce un conjunto de reglas para describir las características más importantes del lenguaje de generación de páginas "Active Server Pages".

Con estas reglas, se pretende abstraer los principales conceptos de la interpretación de un script ASP, y capturar el comportamiento de las principales constucciones sintácticas del lenguaje. La constante evolución y la evidente presencia de bugs en los intérpretes comerciales para lenguajes gcd impiden ser dogmáticamente formal en descripción semántica. Tampoco es nuestra intención: se busca definir una relación de reducción binaria bajo dominios bien definidos mediante reglas que sean simples, claras y precisas.

Las siguientes secciones proponen y discuten distintas reglas semánticas para construcciones sintácticas del lenguaje ASP, de las que se seleccionará un conjunto minimal (en el sentido de ser el necesario y suficiente para mostrar el comportamiento de las instrucciones mas comunmente utilizadas para cualquier desafío de programación en general).

3.2. Máquina Abstracta ASP

La máquina abstracta ASP se conforma de los elementos del dominio semántico previamente descripto y de la relación de reducción $\rightsquigarrow_{ASP}$, que describiremos mediante el enunciado reglas para distintas instrucciones del lenguaje.

La relación de reducción $\rightsquigarrow_{ASP}$ reduce secciones de código ASP (encerradas por $\trianglelefteq\trianglerighteq_{\texttt{ASP}}$) y transforma tuplas del dominio semántico, haciendo modificaciones pertinentes en el estado previo.

3.3. Reglas para interpretación de un script ASP

Los scripts asp se encuentran almacenados en archivos con extensión .asp. Estos son archivos de texto que pueden contener cualquier combinación de los siguientes:

- Texto
- Tags HTML
- Scripts del lado del Server

Un archivo HTML existente puede renombrarse como .asp. Aunque no contenga funcionalidad ASP, el server lo procesa como tal (de manera más eficiente). Podemos considerar de esta manera a HTML como un subconjunto de ASP, al menos desde el punto de vista que nos concierne, es decir, al desarrollar una aplicación en ASP.

Las siguientes secciones describen la relación $\rightsquigarrow_{ASP}$ para reducir secciones de código ASP. En cada caso se explica el funcionamiento, los requisitos y consecuencias de la aplicación de cada regla.

3.3.1. Tratamiento de texto plano y secciones HTML

Dentro de un archivo ASP (script ASP) existen secciónes HTML y de texto plano (que también pueden considerarse parte del código HTML), que deberán tratarse e incluirse de alguna manera en la salida generada como respuesta.

La salida generada, que como vimos se representa con una secuencia en la cuarta componente de las tuplas, en este caso será exactamente igual al input consumido.

El *stream de caracteres* de salida que resulta luego de reducir todo el input, será enviado a un browser como requerimiento HTTP y por lo tanto los tags HTML serán procesados para mostrar el resultado como cualquier documento HTML. ASP en este caso se comporta como un pipe o tubería, sin producir ninguna transformación en los datos de entrada pasándolos para que otro proceso de software posterior se encargue de ello (en este caso ese proceso es el software incluido en el browser HTML para leer y mostrar los datos). La regla correspondiente es:

$$([H, F, \trianglelefteq \texttt{S} \trianglerighteq_{\texttt{HTML}} \oplus Q], Env, Store, Out)$$
$$\rightsquigarrow_{ASP} ([H, F, Q], Env, Store, Out \oplus \trianglelefteq \texttt{S} \trianglerighteq_{\texttt{HTML}})$$

En esta regla, no se incluye ningún antecedente pues en cualquier momento puede reducirse una porción de texto. Provee el contexto de reducción para las secciones de código ASP.

Veamos un pequeño ejemplo de aplicación de esta regla. Si se tiene la siguiente tupla:

$([H, F,$ " `<HTML>`
`<TITLE>Pagina Hello</TITLE>`
`<BODY><% ='Hello!'%></BODY>`
`</HTML>` "$], Env, Store, Out)$

en un paso de aplicación de $\rightsquigarrow_{ASP}$ (usando la regla vista), la tupla resultante es:

$([H, F,$ `<% ='Hello!'%></BODY></HTML>` $],$
$Env, Store, Out \oplus$ `<HTML><TITLE>Pagina Hello</TITLE><BODY>`$)$

aquí queda de manifiesto que no necesariamente la sección de código HTML que se denota con la expresión $\trianglelefteq$ `S` $\trianglerighteq_{\texttt{HTML}}$ debe ser cerrada, ya que el tag `<BODY>` queda abierto y se sigue por el procesamiento de "`<% ='Hello' %>...` ".

3.3.2. Directivas de procesamiento

Existe un conjunto de directivas que indican de qué manera debe procesarse un script ASP. Estas son: LANGUAGE, ENABLESESSIONSTATE, CODEPAGE, LCID, TRANSACTION. Estas directivas son opcionales y existe un conjunto de directivas definido por defecto.

La directiva que sirve para establecer el lenguaje de script por default, debe aparecer en la primer línea de un archivo ASP. La siguiente regla indica cómo debe procesarse una directiva:

$$([H, F, \trianglelefteq \texttt{@LANGUAGE} = \mathit{ScriptingLanguage} \trianglerighteq_{\texttt{ASP}}], \mathit{Env}, \mathit{Store}, \mathit{Out}) \\ \rightsquigarrow_{ASP} ([H, F, \emptyset], \mathit{Env}_{\mathit{ScriptingLanguage}}, \mathit{Store}, \mathit{Out})$$

donde aquí se utiliza la notación especial subindicando el ambiente con el lenguaje de script por default, como se indicó en la descripción del dominio semántico. En general, cualquier variable que comience con el símbolo @, es decir, cualquier directiva de procesamiento podrá ser tratada de la siguiente manera:

$$\mathit{ProcDir}\frac{\begin{array}{ll} @VAR \in & \{@LANGUAGE, \\ & @ENABLESESSIONSTATE, \\ & @CODEPAGE, \\ & @LCID, \\ & @TRANSACTION\ \} \end{array}}{\begin{array}{l} ([H, F, \trianglelefteq \texttt{@VAR} = \mathit{value} \trianglerighteq_{\texttt{ASP}}], \mathit{Env}, \mathit{Store}, \mathit{Out}) \rightsquigarrow_{ASP} \\ ([H, F, \emptyset], \\ \quad \mathit{Env} \Leftarrow^{+} [\text{"}@VAR\text{"} : N = \mathit{newCap}(\mathit{size}(\mathit{Type})) :: \mathit{Type}], \\ \quad \mathit{Store} \Leftarrow^{+} [N : \mathit{value}], \mathit{Out}) \end{array}}$$

en donde $Type$ es el tipo de la variable $@VAR$.

3.3.3. Declaración de variables

Las siguientes reglas para declaración de variables (que corresponden al caso de que el lenguaje de script default utilizado por ASP sea *JScript*), utilizan la subindicación de la sección anterior para ver como se comporta el intérprete cuando encuentra una definición de variable de tipo simple:

$$VarDec_{JScript}\frac{x \notin Env}{\begin{array}{c}([H, F, \trianglelefteq \texttt{var x} \trianglerighteq_{\texttt{ASP}}], Env_{JScript}, Store, Out) \rightsquigarrow_{ASP} \\ ([H, F, \emptyset], Env_{JScript} \Leftarrow^{+} [\text{"}x\text{"} : Nil :: NilType], Store, Out)\end{array}}$$

se utiliza en esta regla un valor nulo Nil y un tipo especial $NilType$ para indicar que el tipo de la variable no se define hasta que sea asignado algún valor del que pueda deducirse mediante el sistema de tipos de *JScript*. Se agrega de este modo un lugar para el identificador en el ambiente con referencias a tipo y valor nulas (el store no se modifica).

Para el caso de que la variable ya se encuentre definida en el ambiente, la sección de código se consume sin producir ningún efecto lateral:

$$\frac{x \in Env}{\begin{array}{c}([H, F, \trianglelefteq \texttt{var x} \trianglerighteq_{\texttt{ASP}}], Env_{JScript}, Store, Out) \\ \rightsquigarrow_{ASP} ([H, F, \emptyset], Env_{JScript}, Store, Out)\end{array}}$$

es decir, no se modifican el store o ambiente.

3.3.4. Comentarios

Esta sección muestra para el caso de los comentarios, como puede influir la subindicación del store en el proceso de reducción, dependiendo del lenguaje de script utilizado.

$([H, F, \trianglelefteq' \texttt{this is a VBScript single} - \texttt{line comment!} \trianglerighteq_{\texttt{ASP}}],$
$\quad Env_{VBScript}, Store, Out)$
$\rightsquigarrow_{ASP} ([H, F, \emptyset], Env_{VBScript}, Store, Out)$

o en JScript:

$([H, F, \trianglelefteq \texttt{//this is a JScript single} - \texttt{line comment!} \trianglerighteq_{\texttt{ASP}}],$
$\quad Env_{JScript}, Store, Out)$
$\rightsquigarrow_{ASP} ([H, F, \emptyset], Env_{JScript}, Store, Out)$

se subindicará el lenguaje de script, cuando no sea claro del contexto en el que se encuentre la reducción o ejemplo, y sólo en caso de ser necesario.

3.3.5. Asignación de variables

En el caso de que la variable no se encuentre declarada, se la registra como si se estuviera declarando, y se agrega el valor de la expresión asignada en el store:

$$AssDef \frac{\begin{array}{l}[\text{"}x\text{"} : \_ :: \_\,] \notin Env,\\ (e, Env, Store) \Longrightarrow v :: T\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \texttt{x} = \texttt{e} \trianglerighteq_{\texttt{ASP}}], Env_{JScript}, Store, Out) \rightsquigarrow_{ASP}\\ ([H, F, \emptyset],\\ \quad Env_{JScript} \Leftarrow^{+} [\text{"}x\text{"} : N = newCap(size(T)) :: T],\\ \quad Store \Leftarrow^{+} [N : v], Out)\end{array}}$$

los símbolos "_" denotan un valor cualquiera (lo que importa en esta regla es que esté el identificador "x" en el ambiente). De esta regla deducimos que la declaración de variables se justifica sólo por prolijidad o estilo de programación, ya que la declaración es automática cuando se asigna una variable de tipo simple.

En el caso de que la variable asignada se encuentre declarada en el ambiente, se hace una actualización de su valor en la dirección del store que se indique en la entrada del ambiente correspondiente al identificador de la misma:

$$AssUpd\frac{\begin{array}{c}[\text{"}x\text{"} : N :: T] \in Env_{JScript},\\ (e, Env, Store) \Longrightarrow v :: T\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \texttt{x} = \texttt{e} \trianglerighteq_{\texttt{ASP}}], Env_{JScript}, Store, Out) \rightsquigarrow_{ASP}\\ ([H, F, \emptyset], Env_{JScript}, Store \Leftarrow^{\checkmark} [N : v], Out)\end{array}}$$

podemos pensar en la función de reducción de expresiones $\Longrightarrow$ como en una relación de reducción para expresiónes asignables.

3.3.6. Programación estructurada

Se dan algunas reglas básicas para mostrar como se comporta la relación de reducción $\rightsquigarrow_{ASP}$ (y de esta manera el intérprete ASP) con las estructuras de control básicas de secuencia, condicional e iteración. Estas reglas utilizan las definiciones estándar para este tipo de construcciones, como las propuestas por [Hen90] o similarmente por [And91] para verificación de programas secuenciales.

Secuencia

La secuencia de dos comandos dentro de una sección de código ASP, o más generalmente, la secuencia de dos secciones de código ASP queda determinada por la aplicación consecutiva de $\rightsquigarrow_{ASP}$ como se muestra en la siguiente regla:

$$SeqCode\frac{\begin{array}{c}([H, F, \trianglelefteq \texttt{S} \trianglerighteq_{\texttt{ASP}}], Env_0, Store_0, Out_0) \rightsquigarrow_{ASP}\\ ([H, F, \emptyset], Env_1, Store_1, Out_1)\\ ([H, F, \trianglelefteq \texttt{Q} \trianglerighteq_{\texttt{ASP}}], Env_1, Store_1, Out_1) \rightsquigarrow_{ASP}\\ ([H, F, \emptyset], Env_2, Store_2, Out_2)\end{array}}{\begin{array}{c}([H, F, \trianglelefteq \texttt{S} \oplus \texttt{Q} \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \rightsquigarrow_{ASP}\\ ([H, F, \emptyset], Env_2, Store_2, Out_2)\end{array}}$$

cabe destacar aquí que para los lenguajes de script incluidos dentro de la distribución de ASP no existe un caracter separador de líneas, sino que en cada línea corresponde a un solo comando. De

este modo, las secuencias de caracateres S y Q incluirán líneas completas, de modo que no pueda cortarse una instrucción por la mitad.

Selección

Las siguientes dos reglas muestra como debe procederse frente a la sentencia de selección if. Se procede de manera análoga para sentencias del tipo case o switch. Recordemos que el código ASP puede estar intercalado con sentencias de texto o HTML:

$$if_{True}\frac{(B, Env, Store) \Longrightarrow (True :: Bool, Env', Store')}{\begin{array}{l}([H, F, \trianglelefteq \texttt{if(B) then} \trianglerighteq_{\texttt{ASP}} \\ \quad \oplus S_1 \oplus \\ \trianglelefteq \texttt{else} \trianglerighteq_{\texttt{ASP}} \\ \quad \oplus S_2 \oplus \\ \trianglelefteq \texttt{endif} \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \rightsquigarrow_{ASP} \\ ([H, F, S_1], Env', Store', Out)\end{array}}$$

en el caso en que la evaluación de la guarda resulte verdadera, se continua mediante la secuencia correspondiente al caso *then* y no se produce ningún efecto sobre la salida. Se procede análogamente para el caso falso con la sección correspondiente al $else$.

$$if_{False}\frac{(B, Env, Store) \Longrightarrow (False :: Bool, Env', Store')}{\begin{array}{l}([H, F, \trianglelefteq \texttt{if(B) then} \trianglerighteq_{\texttt{ASP}} \\ \quad \oplus S_1 \oplus \\ \trianglelefteq \texttt{else} \trianglerighteq_{\texttt{ASP}} \\ \quad \oplus S_2 \oplus \\ \trianglelefteq \texttt{endif} \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \rightsquigarrow_{ASP} \\ ([H, F, S_2], Env', Store', Out)\end{array}}$$

Iteración

De manera similar a las reglas presentadas para el caso de selección, se darán dos reglas para iteración. Se notan con la sintaxis de *VBScript* para la sentencia *While*. Las reglas para las demás sentencias de iteración presentadas por los dos lenguajes de script resultan similares:

$$While_{True}\frac{(B, Env, Store) \Longrightarrow (True :: Bool, Env, Store)}{\begin{array}{l}([H, F, \trianglelefteq \texttt{While(B)} \trianglerighteq_{\texttt{ASP}} \oplus S \oplus \trianglelefteq \texttt{Wend} \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \rightsquigarrow_{ASP} \\ ([H, F, S \oplus \trianglelefteq \texttt{While(B)} \trianglerighteq_{\texttt{ASP}} \oplus S \oplus \trianglelefteq \texttt{Wend} \trianglerighteq_{\texttt{ASP}}], Env, Store, Out)\end{array}}$$

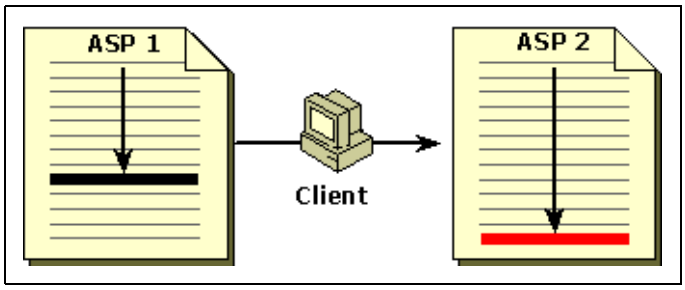
$$While_{False}\frac{(B, Env, Store) \Longrightarrow (False :: Bool, Env, Store)}{\begin{array}{l}([H, F, \trianglelefteq \texttt{While(B)} \trianglerighteq_{\texttt{ASP}} \oplus S \oplus \trianglelefteq \texttt{Wend} \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \rightsquigarrow_{ASP} \\ ([H, F, \emptyset], Env, Store, Out)\end{array}}$$

3.3.7. Control del flujo de una aplicación ASP

Existen seis formas distintas de alterar el flujo de control de ejecución de una aplicación ASP. Como estos mecanismos utilizan objetos predefinidos por ASP, solamente se mencionarán en esta sección. Mas adelante, cuando se vean los objetos ASP en detalle, se proveerán reglas.

Redirección

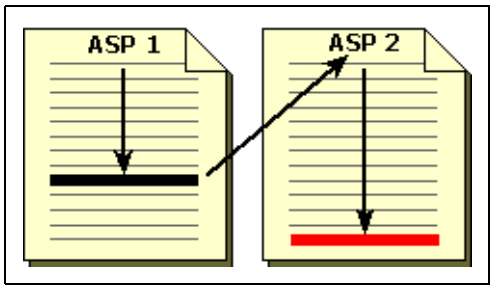
Utilizando el método ***Response.redirect*** $< url >$, puede redireccionarse una página ASP a otra, produciendo el efecto mostrado en la figura:



esto significa que cuando se lee la sentencia de redirección, se deja de procesar el archivo actual y se pasa a procesar el nuevo archivo. La redirección implica que se envía un nuevo requerimiento HTTP al server, de la misma manera que si se tratara de una nueva solicitud por parte de un browser cliente.

Transferencia

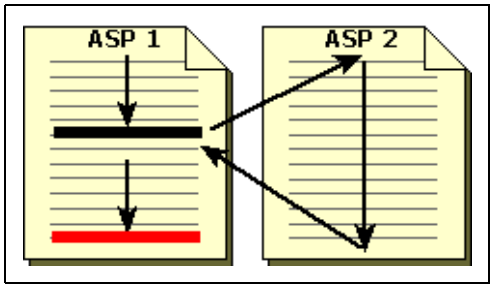
Esta forma de alterar el control es similar a la redirección, pero no necesita iniciar un nuevo requerimiento HTTP, y por lo tanto es mucho más eficiente.



Se puede lograr este efecto utilizando el método ***Server.transfer***, como veremos al proveer una regla para dicho método.

Ejecutar un script ASP

Desde un script ASP puede llamarse a otro, tal como si se tratara de una llamada a procedimiento, retornando luego el control al punto luego del llamado. Este mecanismo permite modularizar aplicaciones, creando archivos ASP que realicen tareas específicas e invocándolos desde distintos puntos de una aplicación o reutilizándolos desde diferentes aplicaciones.

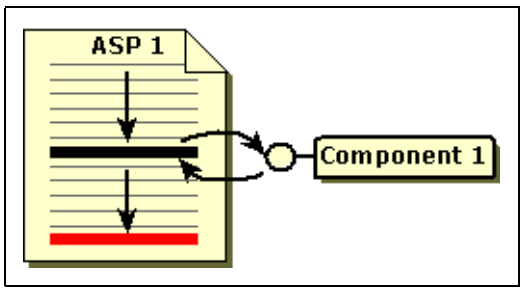


También puede ser útil en el caso de que se desee agregar funcionalidad a una aplicación existente. Se puede lograr este tipo de comportamiento utilizando el método ***Server.execute***, que se detallará más adelante.

Invocación de Componentes

Utilizando la tecnología de componentes COM (Component Object Model) puede diseñarse una aplicación para relegar la responsabilidad de codificar la lógica del problema (o también llamada "lógica del negocio") a distintos componentes reusables de software, que se invocarán desde los distintos scripts ASP. Por ejemplo pueden crearse objetos ActiveX para lograr conexión a bases de datos desde las páginas ASP.

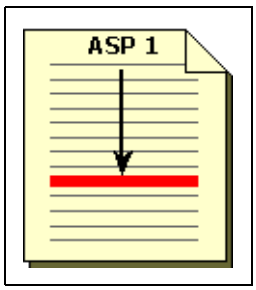
El efecto que tiene una invocación de este tipo es transferir el control al componente y luego retornar al punto siguiente al llamado en el mismo script:



Esto se conseguirá mediante la utilización del método ***Server.CreateObject*** y posteriores llamados a métodos en el objeto creado por esta sentencia.

Salida anormal de un script

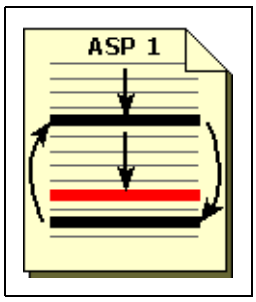
Esto puede ocurrir en casos excepcionales, en los que no quiere terminarse el procesamiento normal del script. Entonces se invoca al método ***Response.End*** que produce el siguiente efecto en el flujo del control:



Esto puede ocurrir por ejemplo cuando se detecta que el cliente no está conectado y por lo tanto no espera por respuestas. En este caso se sale del script y se cierra la sesión actual (se limpia el entorno de sesión).

Procesamiento procedural

Como veremos en la siguiente sección, tanto en *VBScript* como en *Jscript* pueden definirse procedimientos y funciones dentro de un determinado ambiente. Luego, cuando se invocan dichas declaraciones, se produce un salto con retorno en el punto llamador del script ASP.



Cabe destacar que, por lo general, estas definiciones se guardan y se ejecutan del lado del server, por ejemplo cuando se utilizan las definiciones de scripts con los tags HTML:

```
<SCRIPT LANGUAGE=JScript RUNAT=SERVER >
     function MyFunction() { ... }
</SCRIPT>
```

Además, una característica llamativa es que pueden definirse procedimientos y funciones en un lenguaje de script y luego llamarlos con una invocación en otro lenguaje de script.

Veremos reglas para procesar el flujo de control cuando estudiemos los objetos ASP.

3.3.8. Procedimientos y funciones

Las definiciones de procedimientos y funciones pueden incluirse en un script ASP en cualquier lugar, siempre que se encuentre dentro de una sección de código apropiada (sección ASP o tag `<Script>` de HTML). Puede definirse una biblioteca de funciones mediante distintos archivos ASP específicamente diseñados para proveer funcionalidad a otros scripts ASP. Estos scripts se ejecutarán en el servidor ASP, y estarán escritos en cualquier lenguaje de script soportado por ASP.

Análogamente a una definición de variables, cuando se encuentra una definición de una función o procedimiento, se almacena el nombre de la función en el ambiente con un tipo especial, por ejemplo $FunDef$, y la definición (código) con el tipo de retorno se carga en el store como una secuencia de caracteres.

Se dan a continuación reglas para definición y utilización de funciones para el caso de JScript.

$$LoadFunDef\ \frac{\begin{array}{l}[\text{"}f(formals) :: retType\text{"} : \_ :: FunDef\ ] \notin Env_{JScript},\\ \trianglelefteq\ \texttt{function f(formals) :: retType\{//codigo f...\}}\ \trianglerighteq_{\texttt{ASP}} \in F\end{array}}{\begin{array}{l}([H, F, S], Env_{JScript}, Store, Out) \rightsquigarrow_{ASP}\\ ([H, F, S],\\ Env_{JScript} \Leftarrow^{+} [\text{"}f(formals) :: retType\text{"} : N = newCap() :: FunDef],\\ Store \Leftarrow^{+} [N : \text{"}function\ f(formals) :: retType\{//codigo\ f...\}\text{"}],\\ Out)\end{array}}$$

en donde $retType$ es un tipo simple de JScript, N una dirección válida del store retornada por la función $newCap()$. Esta regla puede aplicarse en cualquier momento del tratamiento de un

archivo ASP, ya que no implica reducción (o interpretación) del código: las únicas precondiciones para aplicación de esta regla son que el nombre de la función no se encuentre ya en el ambiente y que la definición de la función se encuentre en el archivo. Teniendo esta definición para el símbolo f en el ambiente, podremos luego utilizar la referencia al código almacenado en el store para servir las llamadas a dicha función.

El hecho de guardar la definición en el store como una secuencia de caracteres tendrá algunas consecuencias a la hora de necesitar usar esta definición. Lo que se pretende hacer es que cada vez que se encuentre una invocación a una función, se busque su definición y se sustituya sintácticamente la invocación por su código (extrayéndola y copiándola del store, como en las cláusulas #define de C++) y reemplazando los valores de los parámetros formales por los valores correspondientes a la invocación.

Un problema que se presenta con esta solución es qué ocurre si $f()$ declara una variable local y cuando se necesita aplicar la función ya existe en el ambiente global una definición para el mismo nombre. Lo que se hace para evitar este tipo de problemas es renombrar las variables locales que defina la función con un prefijo (que llamaremos $FunInvocID$) que indique *a qué función y a qué invocación* pertenece (ya que puede haber varias invocaciones simultáneas recursivas de una misma función) para evitar superponer nombres de variables y de esta manera identificar los nombres unívocamente.

Teniendo en cuenta las consideraciones anteriores, damos a continuación la regla para tratar una invocación a una función ya incorporada al ambiente:

$$FunCall\frac{\begin{array}{l}[\text{"}f(formals) :: retType\text{"} : N : FunDef] \in Env_{JScript},\\ (actuals[i], Env_{JScript}, Store) \Longrightarrow v_i :: T_i,\\ [N : \text{"}f(formals) :: retType\{\text{"} \oplus S \oplus \text{"}\}\text{"}] \in Store\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \texttt{f(actuals)} \trianglerighteq_{\texttt{ASP}}], Env_{JScript}, Store, Out) \rightsquigarrow_{ASP}\\ ([H, F, \trianglelefteq \texttt{S}_\sigma \oplus \text{"}\texttt{returnFrom}\text{"} \oplus \texttt{FunInvocID} \trianglerighteq_{\texttt{ASP}}],\\ Env_{JScript}\\ \Leftarrow^{+}_{\forall x \in LV(\{S\})} [FunInvocID \oplus \text{"}\_x\text{"} : Nil :: NilType]\\ \Leftarrow^{+}_{\forall p_i \in \{formals\}} [FunInvocID \oplus \text{"}\_\text{"} \oplus p_i = formals[i] : v_i :: T_i], Store, Out)\end{array}}$$

$where$

$$FunInvocID = \quad "\#f(formals) :: retType" \oplus toString(invocNum("\#f(formals) :: retType"))$$

$$\sigma = \quad \{x/FunInvocID \oplus "\_x"\}_{\forall x \in LV(\{S\}) \cup \{formals\}}$$

σ es la sustitución sintáctica que se aplica al código de la función para reemplazar la invocación. $invocNum()$ es una función que examina el store y retorna el número de invocaciones simultáneas del encabezado de función que se pasa como parámetro, y el símbolo # es un caracter no válido como parte del nombre de una función , que se utiliza para distinguir los nombres de las variables renombradas de nuevas invocaciones a la función.

Para observar mejor como funciona la aplicación de esta regla sigamos un ejemplo. Sea la siguiente sección de código que define una definición recursiva:

```
<% function rec(t) {
var x     //variable local que enmascara "x" global
x = 'Llamado final!'
if (t >0) {
   Response.Write('llamado recursivo...')
   rec(t-1)
           } else Response.Write(x)
} %>
```

y supongamos que el ambiente contiene las siguientes definiciones globales:

$$["x" : "x"^{\curvearrowright} :: T]$$

entonces al aplicar la regla $LoadFunDef$, se agrega:

$$["rec(t)" : N = newCap() :: FunDef]$$

al ambiente, y la correspondiente definición en el store. Supongamos ahora que se procesa la siguiente sección de código ASP:

```
<% rec(1) %>
```

entonces, aplicando la regla $FunCall$, este llamado a $rec(1)$ se transforma en:

```
<% var rec(t)1_x
 rec(t)1_x = 'Llamado final!'
 if (rec(t)1_t >0) {
   Response.Write('llamado recursivo...')
   rec(rec(t)1_t -1)
                   } else Response.Write(rec(t)1_x)
 returnFrom rec(t)1 %>
```

y al mismo tiempo se agregan en el ambiente definiciones, referencias al store y tipos para las variables locales $rec(t)1\_t$ (parámetro formal inicializado en 1) y $rec(t)1\_x$ con el valor $'Llamado\ final!'$, haciendo el ambiente las veces de registro de activación para mantener estas variables durante llamados recursivos a la función.

Observamos que en el código generado se hace un nuevo llamado a la función (ya que al evaluar la condición del comando if se evalúa resultando verdadera), por lo tanto podemos aplicar nuevamente la misma regla para obtener el código (el nuevo código está entre las líneas comentadas):

```
<% var rec(t)1_x
   rec(t)1_x = 'Llamado final!' //local que enmascara x global
   if (rec(t)1_t >0) {
       Response.Write('llamado recursivo...')
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      var rec(t)2_x
      rec(t)2_x = 'Llamado final!'
      if (rec(t)2_t >0) {
          Response.Write('llamado recursivo...')
          rec(rec(t)2_t -1)
      } else Response.Write(rec(t)2_x)
      returnFrom rec(t)2
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   } else Response.Write(rec(t)1_x)
   returnFrom rec(t)1 %>
```

la evaluación $rec(t)2\_t > 0$ resulta falsa, por lo que se escribe en la salida el mensaje "`Llamado final!`" y luego se eliminan las variables locales y parámetros almacenados durante la activación de los llamados a la función, mediante 2 aplicaciones consecutivas de la regla $FunRet$, la cual procesa la marca de retorno de función ($returnFrom$) dejada por la regla $FunCall$.

A continuación se presenta la regla $FunRet$:

$$FunRet\frac{}{\begin{array}{c}([H, F, \trianglelefteq \texttt{returnFrom f(formals)N} \trianglerighteq_{\texttt{ASP}}], Env_{JScript}, Store, Out) \rightsquigarrow_{ASP} \\ ([H, F, \emptyset], removeAll("f(formals)N\_", Env_{JScript}), gc(Store), Out)\end{array}}$$

la función $removeAll()$ elimina todas los identificadores del ambiente que comiencen con el prefijo que se agregó a las variables locales en el momento de la activación de la función. Esto tiene el mismo efecto que desapilar el registro de activación correspondiente en un lenguaje basado en pila.

Luego se invoca a la función $gc()$ para que el store limpie los valores de las variables no referenciadas desde el ambiente, los cuales no se volverán a utilizar (se marcan como libres).

Otra variante para la semántica de la función $removeAll()$ sería que además de quitar los identificadores correspondientes del ambiente se encargara de liberar todas las celdas inaccesibles del store. Esta variante tiene como ventaja que en un solo paso se actualiza todo el ambiente de programación, pero es mas costosa en cuanto al tiempo de ejecución, porque $gc()$ tiene como ventaja que podría correr como un hilo de ejecución concurrente al proceso de reducción e interpretación de un archivo ASP.

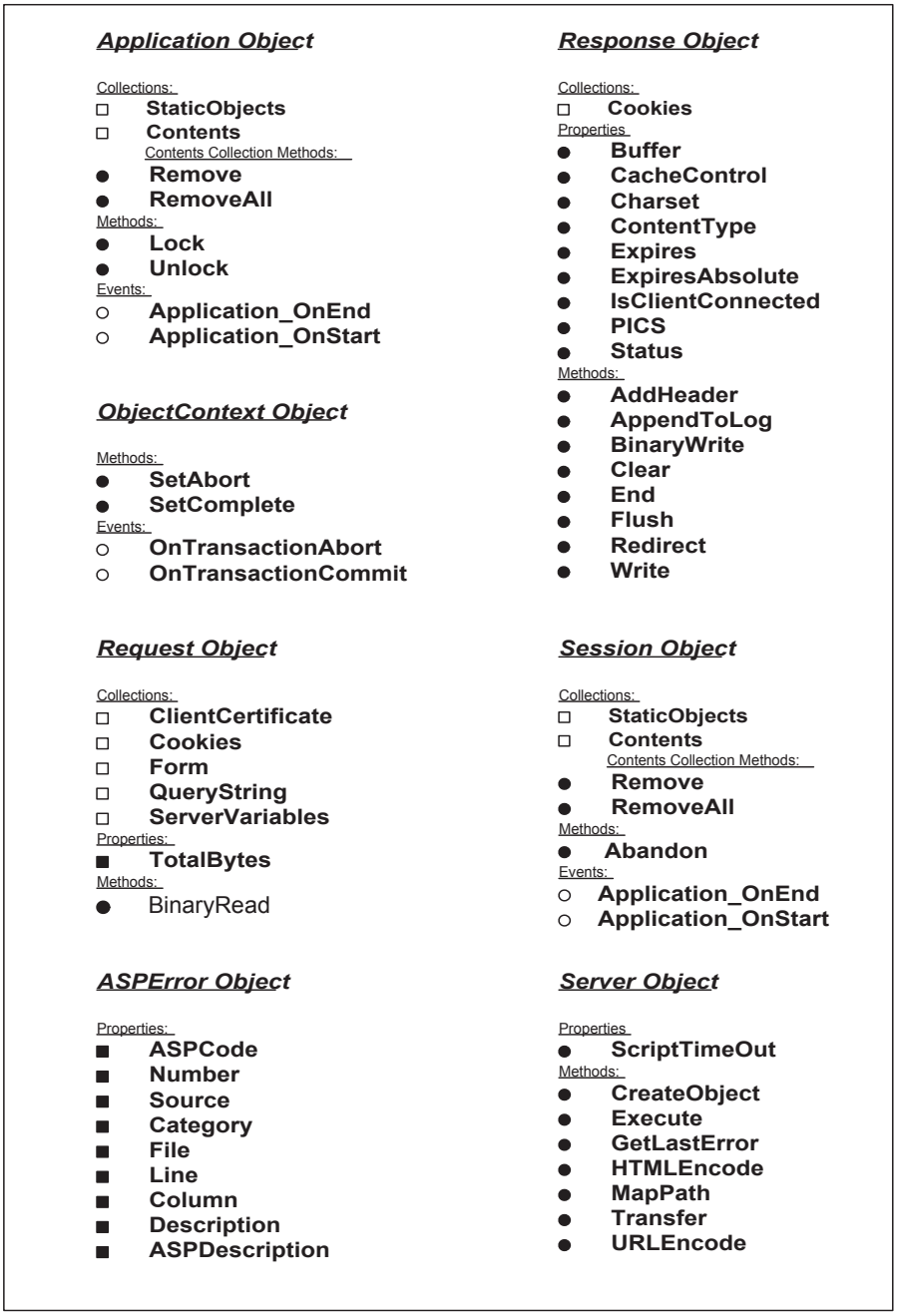
Estos aspectos escapan a la notación de la regla e involucran aspectos de implementación.

3.3.9. Modelo de Objetos integrado en ASP *(built-in objects)*

ASP dispone de un modelo integrado de objetos que permitirán realizar ciertas operaciones que conforman las características fundamentales del lenguaje, es decir, las que le dan el carácter de dinámico. Los objetos integrados en ASP harán posible que un script retorne resultados que dependerán de factores de ejecución y también de la entrada de datos del los usuarios y del estado global del ambiente. Parte del estado global será mantenido dentro de estos objetos ASP.

En el modelo formal, estos objetos se crearán en la memoria (store) y se les asignará un tipo $ObjType$, suficientemente general para tipar cualquier objeto involucrado en un script ASP (al menos será asi hasta que necesitemos cambiar esta especificación). Dado que los objetos ASP pueden ocupar distintos tamaños en memoria, cuando se necesite crear una instancia nueva de uno de ellos, se utilizará en las reglas la función $newCap(< o >)$ que asumiremos buscará una dirección de memoria libre adecuada para contener el objeto $< o >$.

ASP es el encargado de crear, mantener y cargar los valores correctos de los objetos **Request, Response, Application, Session, Server, ASPError y ObjectContext**, los cuales es indispensable conocer para desarrollar cualquier aplicación en ASP. La siguiente es una tabla para referencia rápida sobre los objetos integrados en ASP y sus correspondientes mensajes, colecciones, propiedades y eventos:



Describiremos en la siguientes secciones como es la semántica de los mensajes más importantes de cada uno de estos objetos.

Objeto *Request*

El objeto *Request* modela el requerimiento HTML (solicitud HTTP) que se crea al invocar la URL (Unique Resource Location) correspondiente al script ASP que se está ejecutando. Supongamos que $< url >$ es dicha locación, entonces la solicitud del recurso (en este caso el recurso es el script ASP) puede realizarse o generarse de varias maneras:

1. tipeando $< url >$ directamente en un navegador para internet
2. haciendo una invocación desde otro script ASP, es decir usando (por ejemplo) la instrucción: `Response.redirect`$< url >$
3. desde un documento HTML, incluyendo en el código una referencia mediante el TAG: `<A HREF=`$< url >$`>`
4. desde un formulario HTML, estableciendo que la acción del formulario sea invocar al script ASP: `<FORM ACTION=`$< url >$`>`

Además, a estas formas distintas de invocar un archivo ASP en particular, se puede agregar parámetros como parte de la invocación. Por ejemplo, cuando se tipea en un navegador de internet la dirección:

```
http://localhost/consulta.asp?nombre=Claudia&apellido=Pons
```

se solicita el archivo "consulta.asp" y se envían como parámetros los datos de nombre y apellido de la persona sobre la cuál se quiere consultar. Entonces, se limpia la zona de variables locales del store (se conservan variables de sesión y de aplicación) y se crea una nueva instancia del objeto *Request* con valores de parámetros correspondientes, como se muestra en el siguiente esquema:

```
<Request> = [ QueryString::Property:
               params[String]:ArrayElem::Property:
               params["nombre"]   = "Claudia",
               params["apellido"] = "Pons",
              ; ...
               ClientCertificate::Method:{...};
              ]
```

no nos interesa la estructura interna del objeto, pero debemos tener al menos una idea de que la estructura del objeto debe contener información sobre propiedades, métodos y colecciones para almacenar los parámetros. Por otro lado, se crea en el ambiente un descriptor para el objeto:

$$[\text{"}Request\text{"} : n = newCap(size(< Request >)) :: ObjType]$$

y se guarda el objeto $< Request >$ recientemente creado en la dirección retornada por la función $newCap()$ en el store. Es decir, se ejecuta:

$$Store' = Store \Leftarrow^{+} [n :< Request >]$$

Lo descripto se puede formalizar mediante la siguiente regla:

$$ReqCreation \frac{\begin{array}{l} queryString =< URL/file.asp > \oplus params \\ [params = \text{""} \quad || \\ \ params = \text{"?"} \oplus_{i \in (1..n-1)} (p_i \oplus \text{" = "} \oplus v_i \oplus \text{"\&"}) \oplus (p_n \oplus \text{" = "} \oplus v_n) \ ] \\ H = host(< URL >) \\ file.asp \equiv \trianglelefteq \ \mathtt{S} \ \trianglerighteq_{\mathtt{ASP}} \\ < Request >= CreateRequestObject(queryString) \end{array}}{\begin{array}{l} ([\_, \_, \_], Env, Store, Out) \rightsquigarrow_{ASP} \\ ([H, \text{"}file.asp\text{"}, S], \\ Env \Leftarrow^{+} [\text{"}Request\text{"} : n = newCap() :: ObjType], \\ Store \Leftarrow^{+} [n :< Request >], \\ Out) \end{array}}$$

en donde *queryString* representa el string del requerimiento HTTP, con lista de parámetros *params* y host H, y el archivo referenciado contiene la secuencia de caracteres S (que puede considerarse código ASP por nuestra máquina intérprete).

Esta regla es la primera en toda derivación de nuestro intérprete, dado que el primer evento que genera la ejecución de un script es una invocación al script. Además la regla puede aplicarse cada vez que desde el código se haga una invocación a otro archivo ASP, sin importar el contexto de la primera tupla, como se observa en la regla.

Se describen a continuación los principales mensajes disponibles para interactuar con *Request*.

ClientCertificate

Esta relacionado con la seguridad en la transferencia de datos, y con la validación del cliente. Pueden pedirse a este objeto datos sobre el cliente que se encuentra actualmente en el sitio web, para hacer distintos chequeos. Una regla para este mensaje sería simplemente un chequeo de una variable correspondiente a la sesión del cliente.

Cookies, Form, ServerVariables

Las cookies (galletitas) pueden utilizarse para estructurar información en forma de árbol en el equipo del cliente, para permitir almacenar datos particulares de cada cliente y por ejemplo, personalizar las aplicaciones de acuerdo a las preferencias de cada usuario. ASP provee mecanismos para almacenar y recuperar los valores de las cookies.

Los métodos $Form$ y $ServerVariables$ proveen acceso a distintas colecciones para recuperar datos. $Form$ permite saber si el requerimiento HTTP se hizo mediante un formulario HTML y permite preguntar por cada dato enviado para utilizarlo como parámetro del script ASP que se está procesando. $ServerVariables$ permite consultar mediante un conjunto predeterminado de variables, el estado del server.

QueryString

Como vimos antes en la creación del objeto *Request*, luego de la referencia a un archivo ASP pueden escribirse parámetros luego del signo "?". Luego de que se haya cargado el objeto *Request*

con los valores correctos en memoria, puede utilizarse este mensaje para consultarlos. Veamos una regla de utilización de semántica para este caso:

$$QueryString \frac{\begin{array}{l}(exp, Env, Store) \Longrightarrow (s :: String, Env', Store') \\ < req >= Store[\text{"}Request\text{"}] \\ qs =< req > . \, QueryString(s)\end{array}}{\begin{array}{c}([H, F, \trianglelefteq \texttt{Request.QueryString(exp)} \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \rightsquigarrow_{ASP} \\ ([H, F, \trianglelefteq \texttt{qs} :: \texttt{String} \trianglerighteq_{\texttt{ASP}}, Env', Store', Out)\end{array}}$$

Por lo general la evaluación de una expresión no tendrá efectos laterales, pero como estamos frente a un lenguaje imperativo (y no funcional) es posible que los tenga; en esta regla, el ambiente y el store actuales se modifican generanndo el ambiente y store Env', y $Store'$ (a partir de los equivalentes sin el apóstrofe). Este mensaje permite extraer los valores de los parámetros que se incluyeron en la invocación al script, para utilizarlos de diversas maneras (imprimirlos, tomar desiciones en base a ellos o realizar consultas sobre bases de datos). En la regla queda de manifiesto como se transforma una invocación a un método desde el código ASP en un llamado real al objeto en el store de esta semántica; además, queda implícito en la regla que este método debe utilizarse en el contexto de una expresión mayor que la englobe y utilice el valor qs retornado por $queryString()$.

Objeto *Response*

El objeto *Response* modeliza la página HTML que se genera como respuesta al requerimiento HTTP que solicita un determinado archivo ASP.

Esta respuesta es dependiente del código fuente del archivo ASP que se está ejecutando, y por lo tanto de muchos factores como del estado del entorno y de la memoria, y de los parámetros enviados a dicho script, incluidos dentro del objeto Request.

Write

La principal utilidad de este objeto (y una de las características que hacen de ASP un lenguaje

de generación dinámica de páginas) consiste en poder escribir caracteres en la salida, con el mensaje *Response.write* $< exp >$. Este mensaje puede abreviarse con la directiva de salida $\trianglelefteq =< \texttt{exp} >\trianglerighteq_{\texttt{ASP}}$. Este comportamiento se modela con la siguiente regla:

$$OutputDir\frac{(e, Env, Store) \Longrightarrow (v :: T, Env', Store')}{\begin{array}{r}([H, F, \trianglelefteq = \texttt{e} \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \rightsquigarrow_{ASP} \\ ([H, F, \emptyset], Env', Store', Out \oplus toString(v))\end{array}}$$

Se destaca en la regla que la expresión resultante v puede tener cualquier tipo T, pero es en el momento de imprimirla en la salida que se hace la transformación al tipo *String*, o al tipo correspondiente a la secuencia de salida, no necesariamente *String*.

Clear

Otro de los mensajes que pueden utilizarse con el objeto *Response* es *Clear*. El comportamiento es el siguiente:

$$RespClear\frac{}{\begin{array}{l}([H, F, \trianglelefteq \texttt{Response.Clear} \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \rightsquigarrow_{ASP} \\ ([H, F, \emptyset], Env, Store, \emptyset)\end{array}}$$

es decir, elimina todo el output generado hasta el momento. Esto puede hacerse, por ejemplo, cuando se advierte en algún punto del script que debe comenzarse todo el procesamiento de generación de la salida nuevamente.

End

El mensaje *End* detiene el procesamiento del archivo ASP. El output generado hasta ese punto

es el que se envía como respuesta.

$$RespEnd\frac{}{\begin{array}{l}([H, F, \trianglelefteq \texttt{Response.End} \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \rightsquigarrow_{ASP} \\ ([Nil, Nil, \emptyset], Env, Store, Out)\end{array}}$$

el contexto $[Nil, Nil, \emptyset]$ indica que no existe un contexto luego de ejecutar este mensaje, y la secuencia vacía que no hay mas entrada para consumir. La reducción de la tupla termina con la salida actual procesada, la que se llevaba en la primera tupla.

Redirect

Redirect provee una característica de comportamiento particular: al momento de interpretar esta instrucción, se envía un mensaje de redirección al browser HTML, generando un nuevo requerimiento HTTP que puede referenciar otro archivo ASP o un archivo HTML de hipertexto.

$$Redir\frac{\begin{array}{l}existsFile(< URL >), \\ < URL > \equiv \trianglelefteq \texttt{S} \trianglerighteq_{\texttt{ASP}}, \\ [N :< Request >] \in Store, \\ < Request' >= UpdQueryString(< URL >, < Request >)\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \texttt{Response.Redirect} < \texttt{URL} > \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \rightsquigarrow_{ASP} \\ ([host(< URL >), file(< URL >), \trianglelefteq \texttt{S} \trianglerighteq_{\texttt{ASP}}], \\ CleanLVs(Env), gc(Store \Leftarrow^{\checkmark} [N :< Request' >]), Out)\end{array}}$$

Primero se chequea que la referencia sea un archivo válido existente en el server. Se simboliza con $< URL > \equiv \trianglelefteq \texttt{S} \trianglerighteq_{\texttt{ASP}}$ que el contenido del archivo que se referencia mediante $< URL >$ contiene la secuencia S, que será procesada como fragmento ASP por el intérprete. Para mostrar el comportamiento del objeto **ASPError**, podría dividirse en dos esta regla, similarmente a como se hizo en

la regla para condicionales: en el primer caso no se genera ningún error y las tuplas se transforman normalmente, y en el segundo caso, se detecta que el archivo referenciado no existe. En este caso, se elimina el contexto de ejecución para que no pueda continuarse el procesamiento, y se cargan los valores del error en el objeto ASPError (en este caso, el código 404 correspondiente al error HTTP .ªrchivo no encontrado").

La operación $CleanLVs()$ limpia las variables locales del ambiente (no así las de aplición y sesión que se retienen), y $UpdQueryString()$ actualiza el objeto *Request*, ya que a partir de la redirección el nuevo requerimiento HTML es $< URL >$. Luego se invoca al garbage collector (o función $gc()$) para recuperar los objetos de memoria inaccesibles, es decir, elementos del store que no son referenciados desde el ambiente.

Expires

La propiedad *Response.expires* define un límite temporal en minutos para el acceso a la copia en cache del browser cliente de la página que se genera como salida. Al momento de procesar una asignación a la propiedad, ASP en conjunción con el web server escribe en el encabezado HTTP de la página respuesta la hora en que la página deja de ser válida (actual más tiempo de expiración). Cuando un cliente intenta acceder a la página desde su browser, se compara su hora con la hora en el encabezado: si es menor, se accede a la versión "cacheada" y si ya expiró se solicita en archivo remoto nuevamente. Esta propiedad es interesante, por ejemplo, para modelar consultas en las que los datos cambian en intervalos regulares de tiempo. Usando esta propiedad, puede establecerse un tiempo de expiración adecuado para que un cliente no vea datos desactualizados.

Esta es la regla *Expires*, sin ningúna precondición:

$$Expires\ \frac{}{\begin{array}{l}([H, F, \trianglelefteq\ \texttt{Response.Expires} = \texttt{mins}\ \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \rightsquigarrow_{ASP} \\ \quad ([H, F, \trianglelefteq\trianglerighteq_{\texttt{ASP}}], Env, Store, \\ \qquad setHTTPTimeHeader(now() + mins) \oplus Out)\end{array}}$$

Simplemente se establece que el encabezado HTTP con una función a partir de la hora actual

y el tiempo de expiración de *mins* minutos. Puede lograrse un efecto equivalente a esta propiedad utilizando el método $Server.addHeader(name, value)$ con ciertos valores específicos de *name* y *value*.

El objeto *Server*

El objeto *Server* provee acceso a métodos y propiedades del server ASP. La mayor parte de estos métodos servirán como funciones de utilidad.

La única propiedad disponible es $ScriptTimeOut$, que establece el tiempo máximo que puede transcurrir durante el procesamiento de un script en el server antes de que sea terminado.

El procesamiento de cualquier script dentro del server, no puede demorarse mas de esa cantidad de segundos. Por lo tanto, en la aplicación de toda regla, existe un chequeo implícito de este tiempo máximo. La siguiente regla graficará lo dicho anteriormente:

$$ScriptTimeOutCheck\frac{\begin{array}{l}([H, F, \trianglelefteq \texttt{S} \trianglerighteq_{\texttt{ASP}}], Env_0, Store_0, Out_0)(\rightsquigarrow_{_{ASP}} -\{ScriptTimeOutCheck\}) \\ \quad ([H, F, \emptyset], Env_1, Store_1, Out_1), \\ (Server.scriptTimeOut > 0)\end{array}}{\begin{array}{r}([H, F, \trianglelefteq \texttt{S} \trianglerighteq_{\texttt{ASP}}], Env_0, Store_0, Out_0) \rightsquigarrow_{_{ASP}} \\ ([H, F, \emptyset], Env_1, Store_1, Out_1),\end{array}}$$

La regla correspondiente a la asignación de esta propiedad, simplemente actualiza una variable dentro del objeto server.

Los métodos disponibles para el objeto *Server* son: $CreateObject$, $Execute$, $GetLastError$, $HTMLEncode$, $MapPath$, $Transfer$ y $URLEncode$.

Se darán reglas para $CreateObject$, $Execute$ y $Transfer$, porque son los que ofrecen las características más inovadoras, relevantes, y prácticas, y que incluiremos en el conjunto minimal de construcciones ASP.

CreateObject

Con *Server.CreateObject* se crea una instancia de un componente almacenado en el server. La invocación a este método se hace suministrando como parámetro un $progID$, que es un identificador del tipo o clase del objeto a crear cuyo formato es "[Vendor.]Component[.Version]". El componente debe estar registrado en el server. Esta registración puede hacerse en un script mediante la declaración:

```
<Registration Description ="......"
              ProgID      ="someID" Version="1.00">
</Registration>
```

seguida de la definición e implementación de los métodos del componente. También puede utilizarse *Server.createObject* con un $progID$ correspondiente a componentes instalables que se distribuyen junto con el paquete de soft ASP, (como "Ad Rotator","Database Access", "File Access Component", etc.). Se utiliza la siguiente sentencia:

```
<% Set Session("adR") = Server.CreateObject("MSWC.AdRotator")%>
```

que define una variable de sesión para que almacene un objeto "AdRotator", el cual permite rotar banners con gifs animados para hacer publicidad en una página web, por ejemplo.

Puede también importarse una librería de componentes COM con el meta-tag METADATA,

```
<!--METADATA TYPE="TypeLib"
             FILE="file.dll"
             UUID="typelibraryuuid"
             VERSION="MajorVersionNumber.MinorVersionNumber"
             LCID="localeId"
 -->
```

que se declara en el archivo de configuración "global.asa", para luego instanciar componentes en dicha librería con la función *Server.createObject*.

A continuación se sugiere una regla para el método, en la cual el chequeo del $progID$ se deriva a una función auxiliar, al igual que la creación del objeto real.

$$CreObj\frac{\begin{array}{l} checkRegistration(progID, H) \\ < o >= createServerObject(\text{"}progID\text{"}) \end{array}}{\begin{array}{l} ([H, F, \trianglelefteq \texttt{obj} = \texttt{Server.CreateObject}(progID) \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \\ \rightsquigarrow_{ASP} ([H, F, \trianglelefteq\trianglerighteq_{\texttt{ASP}}], \\ \quad Env \Leftarrow^{+} [\text{"}obj\text{"} : n = newCap(size(< o >)) :: ObjType], \\ \quad Store \Leftarrow^{+} [n :< o >], Out) \end{array}}$$

en donde $< o >$ es la instancia del objeto creado; en este caso se simboliza el proceso de creación del objeto por parte de ASP mediante la función *createServerObject*, que dado el identificador de la clase retorna una instancia nueva. Se verán más adelante otras formas de crear objetos en un script.

Execute

El método *Server.Execute* ejecuta un script asp determinado. Por otra parte, provee una manera de *modularizar* las aplicaciones mediante la división de problemas complejos en módulos más pequeños, ya que puede definirse un script (correspondiente a un módulo) y luego ser invocado desde distintos puntos de una aplicación. Se evita de esta manera scripts demasiado largos que atentan contra la legibilidad.

Además de modularización, promueve el *reuso de código*, pues puede construirse una biblioteca de módulos (scripts ASP) que resuelvan tareas comunes, para que luego puedan ser reutilizadas desde distintos scripts en aplicaciones, invocándolos con el método *Server.Execute*.

La semántica es similar a un llamado a un procedimiento en muchos lenguajes de programación, como se observa en la siguiente regla:

$$
\begin{array}{l}
\trianglelefteq \texttt{Server.Execute}(\texttt{F}_2 \oplus \texttt{args}) \trianglerighteq_{\texttt{ASP}} \in F_1 \textit{ in line } l_j, \\
[args = \text{""} \quad || \\
\ args = \text{"?"} \oplus_{i \in (1..(n-1))} [p_i \oplus \text{"} = \text{"} \oplus v_i \oplus \text{"\&"}] \oplus p_n \oplus \text{"} = \text{"} \oplus v_n] \\
\ ] \\
existsInServer(H, F_2), \\
F_2 \equiv \trianglelefteq \texttt{S}_2 \trianglerighteq_{\texttt{ASP}}, \\
[N :< Request >] \in Store, \\
\hline
([H, F_1, \trianglelefteq \texttt{Server.Execute}(\texttt{F}_2 \oplus \texttt{args}) \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \rightsquigarrow_{ASP} \\
\ ([H, F_2, S_2\sigma \oplus \text{"}returnFrom : \text{"} \oplus ScriptInvocID \\
\qquad \oplus \text{"}toFile : \text{"} \oplus F_1 \\
\qquad \oplus \text{"}line : \text{"} \oplus toString(l_j + 1)], \\
\ Env \Leftarrow^{+}_{\forall x \in LV(S_2 \cup \{p_i\}_{i \in (1..n))}} [\text{"\#"} \oplus ScriptInvocID \oplus \text{"}\_x\text{"} : Nil :: NilType], \\
\ Store \Leftarrow^{\checkmark} [N : UpdQueryString(\text{"}http://H/F_2\text{"} \oplus args, < Request >)], \\
\ Out)
\end{array}
\quad ExecScript
$$

$$
\begin{array}{rrcl}
where & ScriptInvocID & = & F_2 \oplus args \oplus toString(invocNum(F_2 \oplus args)) \\
 & \sigma & = & \{x/ScriptInvocID \oplus \text{"}\_x\text{"}\}_{\forall x \in LV(S_2 \cup \{p_i\}_{i \in (1..n))}}
\end{array}
$$

La lectura parece complicada, pero es en realidad una suma de varios chequeos y acciones simples. Veamos primero las precondiciones:

- El archivo que está siendo procesado actualmente, F_1, contiene una llamada al método *Server.Execute* con parámetro $(F_2 \oplus args)$ en la línea l_j

- La secuencia denotada por $args$ puede ser una lista de parámetros de la forma $?p_1 = v_1 \& p_2 = v_2 .. \& p_n = v_n$ o puede ser la secuencia vacía (no hay parámetros)

- El archivo que se quiere ejecutar, F_2, existe en el server local H

- El objeto $built - in$ ASP *Request* se encuentra almacenado en la dirección N del Store

Enumeremos ahora los efectos de aplicar la regla una vez chequeadas las precondiciones:

- Se origina el cambio del conexto actual del procesamiento al archivo F_2 y su código S_2

- Se aplica a S_2 la sustitución σ, que antepone a las apariciones de las variables locales de S_2 un prefijo que identifica unívocamente la llamada al método $Execute$, de manera similar al caso de llamado a funciones. Esto es para que al cargarlas al ambiente no enmascaren las posibles definiciones actuales con el mismo nombre.

- Se agrega, al final la secuencia a procesar, una marca para el punto de retorno al archivo llamador ($returnFrom...$)

- Se agregan al ambiente las definiciones de las variables locales

- Se modifica el objeto *Request* en el store para que contenga el nuevo $queryString$ como propiedad.

- La salida no cambia.

La regla para retornar de una ejecución de un archivo ASP, $ExecRet$, lee la marca de retorno agregada por la regla anterior:

$$ExecRet\frac{F \equiv \trianglelefteq \texttt{S} \trianglerighteq_{\texttt{ASP}}}{\begin{array}{l}([H, F_2, "returnFrom : " \oplus ScriptInvocID \oplus \\ \quad "toFile : " \oplus "F" \oplus \\ \quad "line : " \oplus l_j], Env, Store, Out) \rightsquigarrow_{ASP} \\ ([H, F, S_{(l_j..l_n)}], \\ \quad removeAll(Env, "\#" \oplus ScriptInvocID \oplus "\_"), gc(Store), Out)\end{array}}$$

como antes, la función *removeAll* borra todas las variables locales al llamado que se hizo para ejecutar el archivo F_2, es decir, las que empiezan con el prefijo $"\#" \oplus ScriptInvocID$, y también se invoca a $gc()$ para que libere esas posiciones del *Store*.

Aquí la expresión $S_{(l_j..l_n)}$ denota la porción de S que comienza en l_j y termina al final de la secuencia S; recordemos que en estos lenguajes de scripting ($JScript$ o $VBScript$) no hay separadores de comandos, sino que se utiliza una instrucción por línea, y por lo tanto la separación de líneas y comandos resulta equivalente.

Transfer

El método *Server.transfer* transfiere el control a un segundo archivo ASP dentro del mismo server, a la vez que le envía toda la información de procesamiento que ha guardado el archivo llamador en su entorno (variables de sesión, variables de aplicación, salida, y toda la información almacenada en objetos built-in).

La siguiente, es la regla correspondiente:

$$Transfer\frac{f\_2.asp = file(< path >)}{\begin{array}{l}([H, f\_1.asp, \trianglelefteq \texttt{Server.transfer} < \texttt{path} > \trianglerighteq_{\texttt{ASP}}], \\ Env, Store, Out) \rightsquigarrow_{ASP} \\ \quad ([H, f\_2.asp, \trianglelefteq\trianglerighteq_{\texttt{ASP}}], Env, Store, Out)\end{array}}$$

La precondición (o antecedente) indica que `f_2.asp` es el archivo referenciado en en el parámetro del método $Server.transfer$, si bien la referencia a éste puede no ser explícita.

El objeto *Application*

El objeto *Application* (aplicación) sirve para almacenar información que será compartida entre usuarios de una determinada aplicación. Se proveen en él métodos para agregar y quitar objetos que serán utilizados durante toda la aplicación.

Por razones de consistencia en los datos, se proveen también métodos para asegurar exclusión mutua en su acceso. Veamos estos primero, son los métodos $Lock()$ y $Unlock()$.

Lock y Unlock

Para asegurar que varios usuarios no alteren una propiedad al mismo tiempo se dispone de métodos para bloquear y desbloquear el objeto *Application*. El siguiente fragmento de código muestra como pueden usarse *Lock* y $Unlock$ para acceder a variables almacenadas en el objeto aplicación:

```
<% Application.Lock
Application("NumVisits") =
Application("NumVisits") + 1
Application("datLastVisited") = Now()
Application.Unlock %>
```

Se dará la regla para el método $lock()$, por ser ambas reglas muy similares:

$$Lock\frac{\begin{array}{l}[N :< Application >] \in Store,\\ NOT < Application > .Locked(),\\ < Application' >= lock(< Application) >)\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \texttt{Application.Lock} \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \rightsquigarrow_{ASP}\\ ([H, F, \trianglelefteq\trianglerighteq_{\texttt{ASP}}], Env, Store \Leftarrow^{\checkmark} [N :< Application' >], Out)\end{array}}$$

Objetos estáticos y dinámicos

Se provee acceso a los componentes instanciados con la cláusula `<object>` de alcance aplicación mediante la colección *StaticObjects* (objetos estáticos). También se provee acceso a los items que fueron agregados a la aplicación desde algún script (objetos dinámicos) mediante una asignación a la colección *Contents* de la forma: **Application("aName") = aValue**.

Las siguientes reglas permiten comprender que sucede ante una asignación de este tipo:

$$\textit{AddApplConts}\frac{\begin{array}{l}NOT < Application > .Locked(),\\ "aName" \notin < Application > .Contents(),\\ [N :< Application >] \in Store,\\ T = Type(aValue),\\ < Application' >= \\ \quad Add(["aName" : n = newCap(size(T)) :: T],\\ \quad\quad < Application > .Contents())\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \texttt{Application}(\texttt{"aName"}) = \texttt{aValue} \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \rightsquigarrow_{ASP}\\ \quad([H, F, \trianglelefteq\trianglerighteq_{\texttt{ASP}}], Env,\\ \quad\quad Store \Leftarrow^{\checkmark} [N :< Application' >] \Leftarrow^{+} [n : aValue], Out)\end{array}}$$

que quiere decir que si no existe el nombre asignado en la colección, entonces se agrega su descriptor en la colección y se referencia su valor en el store. Estas variables no se declaren en el ambiente global, pues esta colección reside en el store, dentro del objeto Application, y es justificable por estar lógicamente relacionadas.

Análogamente para el caso de que exista un descriptor para el nombre asignado en la colección *Application.Contents*, se actualiza su valor:

$$UpdApplConts\frac{\begin{array}{l}NOT < Application > .Locked(),\\ [\text{"}aName\text{"} : n :: T] \in < Application > .Contents()\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \texttt{Application}(\text{"}\texttt{aName}\text{"}) = \texttt{aValue} \trianglerighteq_{\texttt{ASP}}],\\ \quad Env, Store, Out) \rightsquigarrow_{ASP}\\ \quad\quad ([H, F, \trianglelefteq\trianglerighteq_{\texttt{ASP}}], Env, Store \Leftarrow^{\checkmark} [n : aValue], Out)\end{array}}$$

en este caso, se declara (en la premisa de la regla) el descriptor, para obtener la dirección que debe actualizarse: n. No hace falta conocer la dirección donde se encuentra el objeto $< Application >$, ya que no se accede directamente a él.

De la misma manera que para la colección *Application.Contents*, se dará una regla para leer una declaración estática `<OBJECT>` y almacenar el objeto resultante en la colección *Application.StaticObjects*. Veamos solamente el caso en que el objeto se agrega a la colección por primera vez, (el otro es aún más simple):

$$AddApplSObjs\frac{\begin{array}{l}NOT < Application > .Locked(),\\ \text{"}aName\text{"} \notin < Application > .StaticObjects(),\\ [N :< Application >] \in Store,\\ < Application' >=\\ \quad Add([\text{"}aName\text{"} : n = newCap(size(< o >)) :: ObjType],\\ \quad\quad < Application > .StaticObjects)\\ < o >= createStaticObject(\text{"}SomeProgID\text{"}|\text{"}ClsId : someID\text{"})\end{array}}{\begin{array}{l}([H, F, \trianglelefteq < \texttt{OBJECT} \trianglerighteq_{\texttt{HTML}}\\ \quad \trianglelefteq \texttt{RUNAT} = \texttt{Server SCOPE} = \texttt{Application} \trianglerighteq_{\texttt{HTML}}\\ \quad \trianglelefteq \texttt{ID} = \text{"}\texttt{aName}\text{"} \trianglerighteq_{\texttt{HTML}}\\ \quad \trianglelefteq [\texttt{PROGID} = \text{"}\texttt{SomeProgID}\text{"}|\texttt{CLASSID} = \text{"}\texttt{Clsid : someID}\text{"}] > \trianglerighteq_{\texttt{HTML}}\\ \quad \trianglelefteq < /\texttt{OBJECT} > \trianglerighteq_{\texttt{HTML}}],\\ \quad\quad Env, Store, Out) \rightsquigarrow_{ASP}\\ \quad ([H, F, \trianglelefteq\trianglerighteq_{\texttt{HTML}}], Env,\\ \quad\quad Store \Leftarrow^{\checkmark} [N :< Application' >] \Leftarrow^{+} [n :< o >], Out)\end{array}}$$

El objeto *Session*

El objeto sesión permite al programador de una aplicación ASP almacenar información relacionada con una determinada sesión de un usuario en la aplicación. Un típico ejemplo de esta clase de información es el "carrito de compras", en el que un usuario acumula una compra parcial en un sitio que vende artículos de distintos tipos. Luego de que el usuario abandona el sitio, esta información no se vuelve a utilizar.

Este objeto está compuesto principalmente por dos colecciones, exactamente con el mismo criterio que en *Application*, para acceder a objetos estáticos y dinámicos: *Session.StaticObjects*() y *Session.Contents*(). Ambas colecciones son diccionarios indexados por nombre y compuestos por objetos no homogéneos. La primera contiene todos los objetos estáticos para los cuales se estableció un alcance de sesión mediante el tag `<OBJECT>` , como se muestra en la siguiente porción de código:

```
<OBJECT RUNAT =SERVER SCOPE=Session ID=MyBrowser
        PROGID="MSWC.BrowserType"></OBJECT>
```

en la cual se instancia un objeto *BrowserType*, se lo identifica con el nombre "MyBrowser" y se establece que el alcance de este objeto es solo dentro de la sesión correspondiente al momento de procesar la declaración.

La segunda colección, *Session.Contents*(), contiene todos los elementos de tipo simple u objetos que se almacenan accediendo a la colección mediante un commando de la forma
`<%Set Session("aName")=aValue %>`, que establece que el nombre de la variable "aName" contenga el valor "aValue", como en los siguientes ejemplos:

```
<%
Dim anArray(2)
    anArray(1)="second"
    anArray(2)="third"
    Session("anArray")=anArray %>
```

aquí se almacena en el nombre "anArray", un arreglo de tres elementos. En el siguiente fragmento de código se almacena un objeto "BrowserType" en la variable de sesión con nombre "MyBrowser".

```
<% Set Session("MyBrowser") =
     Server.CreateObject("MSWC.BrowserType") %>
```

De esta manera pueden guardarse objetos en la colección correspondiente a la sesión. Luego podrán eliminarse objetos o valores de tipo simple de la colección $< Session >.Contents()$ utilizando las funciones $Remove(name|index)$ o $RemoveAll()$.

Las reglas correspondientes a este tipo de asignaciones son similares a las sugeridas para el caso del objeto *Application*:

$$AddSessConts\frac{\begin{array}{l}"aName" \notin < Session > .Contents(),\\ [N :< Session >] \in Store,\\ < Session' >= \\ \quad Add(["aName" : n = newCap(size(T)) :: T = Type(aValue)],\\ \quad\quad < Session > .Contents())\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \texttt{Session("aName")} = \texttt{aValue} \trianglerighteq_{\texttt{ASP}}],\\ \quad Env, Store, Out) \rightsquigarrow_{ASP}\\ \quad ([H, F, \trianglelefteq\trianglerighteq_{\texttt{ASP}}], Env,\\ \quad\quad Store \Leftarrow^{\checkmark} [N :< Session' >] \Leftarrow^{+} [n : aValue], Out)\end{array}}$$

la interpretación de la regla es exactamente igual al caso *Application*, excepto aquí no se consulta $locked()$. Para el caso de actualización de una variable de sesión ya existente se tiene:

$$UpdSessConts\frac{["aName" : n :: Type] \in < Session > .Contents()}{\begin{array}{l}([H, F, \trianglelefteq \texttt{Session("aName")} = \texttt{aValue} \trianglerighteq_{\texttt{ASP}}],\\ \quad Env, Store, Out) \rightsquigarrow_{ASP}\\ \quad ([H, F, \trianglelefteq\trianglerighteq_{\texttt{ASP}}], Env,\\ \quad\quad Store \Leftarrow^{\checkmark} [n : aValue], Out)\end{array}}$$

y para asignar a un objeto estático (declarado con la sentencia `<OBJECT>` de HTML) un alcance de sesión, tenemos la siguiente regla:

$$AddSessSObjs\frac{\begin{array}{l}[\text{"}aName\text{"} : \_ :: \_] \notin < Session > .StaticObjects(),\\ [N :< Session >] \in Store,\\ < Session' >= \\ \quad Add([\text{"}aName\text{"} : n = newCap(size(< o >)) :: ObjType],\\ \quad\quad < Session > .StaticObjects())\\ < o >= createStaticObject(\text{"}SomeProgID\text{"}|\\ \quad\quad\quad \text{"}Clsid : someNum\text{"})\end{array}}{\begin{array}{l}([H, F, \trianglelefteq < \texttt{OBJECT} \trianglerighteq_{\texttt{HTML}}\\ \quad \trianglelefteq \texttt{RUNAT} = \texttt{Server SCOPE} = \texttt{Session ID} = \text{"}\texttt{aName}\text{"} \trianglerighteq_{\texttt{HTML}}\\ \quad \trianglelefteq [\texttt{PROGID} = \text{"}\texttt{SomeProgID}\text{"}| \trianglerighteq_{\texttt{HTML}}\\ \quad\quad \trianglelefteq \texttt{CLASSID} = \text{"}\texttt{Clsid : someNumber}\text{"}] > \trianglerighteq_{\texttt{HTML}}\\ \quad \trianglelefteq < /\texttt{OBJECT} > \trianglerighteq_{\texttt{HTML}}], Env, Store, Out) \rightsquigarrow_{ASP}\\ \quad ([H, F, \trianglelefteq \trianglerighteq_{\texttt{HTML}}], Env,\\ \quad\quad Store \Leftarrow^{\checkmark} [N :< Session' >] \Leftarrow^{+} [n :< o >], Out)\end{array}}$$

la interpretación es similar a la regla correspondiente para el objeto *Application*.

Objeto *ASPError*

El objeto ASPError es el encargado de almacenar la información relacionada con el estado del último error de ejecución producido por el procesamiento de un script ASP. Contiene, entre otros datos, el código del error, el archivo y número de línea donde se produjo y una descripción.

Cuando el intérprete ASP detecta un error en el procesamiento de un script, carga los valores en este objeto para que estén disponibles para detección y administración desde un script. Pueden utilizarse los datos almacenados en este objeto para informar al usuario los detalles del error o tomar desiciones acorde al tipo del mismo.

$$AspError\frac{\begin{array}{c}[n :< aspError >] \in Store \\ < aspError' > = catchError(S)\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \texttt{S} \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \rightsquigarrow_{ASP} \\ ([H, F, \trianglelefteq\trianglerighteq_{\texttt{ASP}}], Env, \\ \quad Store \Leftarrow^{\checkmark} [n :< aspError' >], Out)\end{array}}$$

Esta regla agrega no determinismo en la mayoría de las otras reglas vistas anteriormente, pero sólo en un nivel. La función *catchError*() se encarga de cargar los datos del error recientemente producido. El estado del objeto indica que se ha producido un error, y queda bajo la responsabilidad del usuario del lenguaje la posibilidad de utilizar esta información o ignorarla.

3.3.10. Conectividad con Bases de Datos

Se desea encontrar en esta sección una forma de conectarse con bases de datos, que sirva para cualquier aplicación en general. Se sugiere en la documentación [Cor99] que se utilice la tecnología *ADO*, pues es fácil de usar, extensible y pueden escribirse scripts compactos y escalables para conectarse con fuentes *OLEDB*.

La forma de conectarse con una fuente de datos es creando un objeto conexión (con el alcance adecuado para cada necesidad, por ejemplo de aplicación o sesión), dando un string de conexión adecuado para cada proveedor y versión del soft. Luego, teniendo el objeto conexión almacenado en una variable, pueden ejecutarse strings SQL para interactuar con la base de datos. La siguiente es una tabla que indica distintas posibilidades para conectarse con distintas fuentes de datos creando distintos objetos conección *ADO*:

Data Source	**OLEDB Connection String**
Microsoft Access	Provider=Microsoft.Jet.OLEDB.4.0;Data Source=physical path to .mdb file
Microsoft SQL Server	Provider=SQLOLEDB.1;Data Source=path to database on server
Oracle	Provider=MSDAORA.1;Data Source=path to database on server
Microsoft Indexing Service	Provider=MSIDXS.1;Data Source=path to file

Data Source	ODBC Connection String
Microsoft Access	Driver=Microsoft Access Driver (*.mdb);DBQ=physical path to .mdb file
Microsoft SQL Server	Driver=SQL Server;SERVER=path to server
Oracle	Driver=Microsoft ODBC for Oracle;SERVER=path to server
Microsoft Excel	Driver=Microsoft Excel Driver (*.xls);DBQ=physical path to .xls file; DriverID=278
Microsoft Excel 97	Driver=Microsoft Excel Driver (*.xls);DBQ=physical path to .xls file;DriverID=790
Paradox	Driver=Microsoft Paradox Driver(*.db);DBQ=physical path to .db file;DriverID=26
Text	Driver=Microsoft Text Driver (*.txt;*.csv);DefaultDir=physical path to .txt file
Microsoft Visual FoxPro (with a database container)	Driver=Microsoft Visual FoxPro Driver;SourceType=DBC;SourceDb=physical path to .dbc file
Microsoft Visual FoxPro (without a database container)	Driver=Microsoft Visual FoxPro Driver;SourceType=DBF;SourceDb=physical path to .dbf file

Puede almacenarse una conexión una sola vez para toda una sesión de la siguiente manera:

```
<% Set Session("objConn")=
     Server.CreateObject("adodb.connection") %>
```

y si se quiere tener un alcance para todos los scripts de la aplicación, puede almacenarse en el archivo “global.asa” el string de conexión, seteando una variable con alcance de aplicación en el script correspondiente al evento *Application_OnStart*, que se dispara al comenzar la aplicación:

```
<% Application("ConnectionString") =
    "Provider=Microsoft.Jet.OLEDB.4.0;
     Data Source=C:\Data\Inventory.mdb" %>
```

y luego desde cada script que accede la base de datos, se instancia una conexión, se utiliza y luego se cierra:

```
<OBJECT RUNAT=SERVER ID=cnn
 PROGID="ADODB.Connection">
</OBJECT>

cnn.Open Application("ConnectionString")

...
cnn.execute "INSERT...."
...
cnn.execute "UPDATE...."
...
cnn.execute "DELETE...."
...

cnn.Close
```

Otra variante para manipular bases de datos desde scripts ASP, consiste en utilizar objetos *command*, que permiten preparar y precompilar las querys que se enviarán a la base de datos para reutilizarlas cuantas veces se necesiten. De esta manera se mejora la eficiencia de las consultas sql.

3.3.11. Variantes para manipulación de Objetos en scripts ASP

Existen distintas variantes para manipular objetos desde ASP. También existen distintos tipos de objetos. En esta sección se mencionará cada uno de estos tipos de objetos, y se sugerirá el uso más adecuado desde el punto de vista de la ingeniería de soft, es decir, considerando los objetivos de reusabilidad de código, legibilidad, claridad de las soluciones, un diseño modular y orientación a objetos, mantenimiento, facilidad para detección y corrección de errores, asi como también la verificación de la corrección en las soluciones. Queda a criterio del lector adoptar o no como métodos prácticos estas sugerencias.

Objetos provistos por VBScript y JScript

ASP como ya vimos hace posible trabajar con lenguajes de scripting en el lado del server; cada vez que abrimos una sección de código en un archivo asp, estamos comenzando un nuevo script en el lenguaje por defecto (a menos que comencemos la sección con un tag HTML `<script language=...>`).

Para poder trabajar con un lenguaje de script en particular, debe instalarse el motor del lenguaje en el server. En ASP se incluyen los motores de dos lenguajes de script: VBScript y JScript. Cada lenguaje tiene sus particularidades, ventajas y desventajas para trabajar con objetos.

En estos dos lenguajes, si bien los objetos soportan la construcción de un diseño medianamente complejo y apto para problemas pequeños, son definitivamente una mala opción si tenemos en cuenta los objetivos antes mencionados, ya que no soportan los conceptos fundamentales de la POO: abstracción, encapsulamiento, herencia, polimorfismo y binding dinámico.

Estos objetos son meras agregaciones de tipos de datos más simples provistos por los mismos lenguajes (producto cartesiano de tipos simples). No se provee de mecanismos de ocultamiento o de información ni encapsulamiento; sí se provee de un rudimentario mecanismo de herencia, todavía en evolución.

IIS Admin Objects

Esta clase de objetos está destinada a administrar servicios en el server desde los scripts de una aplicación, tales como restringir el acceso a archivos o manipular servicios FTP. Pueden instanciarse clases con funcionalidad y comportamientos conocidos y especificados a partir de una completa jerarquía cuyas clases se encuentran disponibles en distintas locaciones en la www, provistas por IIS. Por ejemplo:

```
<%  Dim WebServiceObj
  Set WebServiceObj = GetObject("IIS://LocalHost/W3SVC") %>
```

este par de sentencias crea en un script ASP un objeto que permite manipular distintas opciones en el servicio web que se está ofreciendo, instanciando objetos “web-server”.

Será necesario manipular estos objetos, pero como una herramienta independiente al diseño de

la aplicación y en scripts escritos con un fin específico, tal como crear un directorio virtual o cambiar permisos de acceso para un archivo. Se invocará a estos scripts cuando se necesite realizar estas tareas, pudiendo parametrizarlos para permitir una mayor reutilización.

Componentes COM vs. lenguajes específicos para POO

Los componentes COM son bloques de construcción de software que contienen código para ejecutar una tarea o un conjunto de tareas específico. Los componentes pueden combinarse con otros componentes para realizar tareas mas complejas. Por ejemplo se pueden utilizar componentes ADO (ActiveX Data Objects) para agregar conexión con bases de datos a aplicaciones web.

En la documentación [Cor99] se sugiere que la mejor manera de manejar la lógica del negocio en las aplicaciones web basadas en scripts ASP es utilizar la tecnología COM. Sin embargo, los componentes COM carecen de las propiedades fundamentales de la POO. Existen por ejemplo muchos patrones y casos de uso cononcidos que difícilmente pueden aplicarse o adaptarse a una aplicación construida con componentes COM.

Considerando esto, se propone un enfoque en el que el modelo abstracto de objetos con el que se piensa la lógica del negocio en la aplicación sea desarrollado en lenguajes como C++, Java o Smalltalk, en los que el más fácilmente pueden aplicarse patrones conocidos; este modelo puede ser manipulado por interfaces COM desde los scripts de una aplicación.

Se puede registrar clases Java como componentes COM utilizando el programa "JavaReg.exe" distribuidos por las compañías *Sun* y *Microsoft*. Luego de registrada una clase, se obtiene el identificador de programa PROGID y el identificador de clase CLSID, y podremos instanciar esta clase desde scripts ASP utilizando el tag `<OBJECT>` o el método *Server.CreateObject*, proporcionando alguno de estos identificadores para referenciar la clase registrada como ya vimos anteriormente.

Monikers Java

Un moniker es un nombre que identifica unívocamente un componente COM. Los monikers soportan la operación binding, que es el proceso de localizar el objeto nombrado por un moniker, activándolo o cargándolo en memoria si no se encuentra ya en ésta y retornar un puntero para referenciar ese objeto.

En el caso de los monikers Java, para instanciar una clase se utiliza el prefijo "java:" seguido del nombre completo de la clase. Por ejemplo para instanciar la clase *Date*, puede utilizarse las siguiente función desde ASP:

```
<%  Dim dtmDate
    Set dtmDate = GetObject("java:java.util.Date")%>
```

Esta opción sería muy útil si funcionara. Probándola exhaustivamente en el momento del desarrollo de este trabajo, la función funcionaba para clases de paquetes de la distribución standard de Java, pero no para clases definidas por el usuario. ***Esta es una muy buena justificación para el desarrollo de tecnologías basadas en semánticas formalmente definidas, ya que la especificación no se corresponde con el comportamiento real de la función.***

En un capítulo posterior, se verá un ejemplo de como utilizar objetos Java desde scripts ASP.

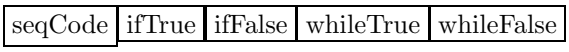
3.4. El pizarrón ASP

Como resultado final del conjunto de características analizado para el caso ASP, se dará un conjunto minimal de reglas (minimal en el sentido de que cubra las características necesarias y suficientes) con el objetivo de construir una aplicación web. A este conjunto lo llamaremos "pizarrón".

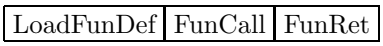
La intención es que el cuadro con todas las reglas elegidas sirva como referencia rápida sobre las técnicas de programación que se utilizarán para cada proyecto adoptadas para cada circunstancia. Por supuesto, se pretende que el grupo de desarrollo evolucione y adquiera práctica en las técnicas que le resultan mas adecuadas, eficaces y eficientes, y actualice el pizarrón acorde a su evolución este conjunto de reglas: el pizarrón debe ser dinámico. Cada fragmento dentro corresponde a una categoría de programación distinta.

Fragmento 1: Definición de variables

Fragmento 2: Estructuras de Control

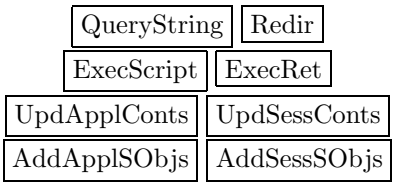


Fragmento 3: Tratamiento de procedimientos y funciones



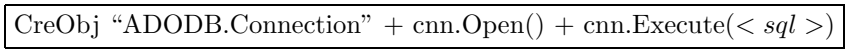
Fragmento 4: Procesamiento de salida

Fragmento 5: Objetos built-in ASP



Fragmento 7: Creación de objetos

Fragmento 6: Métodos de conexión con bases de datos



Capítulo 4

Semántica Operacional PHP

A MAN SAID TO THE UNIVERSE

A man said to the universe,
"Sir, I exist!"
"However," replied the universe,
"The fact has not created in me
A sense of obligation."

Stephen Crane (1871-1900)

4.1. Introducción

Idem capitulo anterior, ahora con el lenguaje PHP.

4.2. Máquina Abstracta PHP

La máquina abstracta PHP (de la misma manera que la máquina ASP) se compone de los elementos del dominio semántico previamente descripto y de la relación de reducción $\rightsquigarrow_{PHP}$, que los relaciona entre sí. Más precisamente, la relación de reducción $\rightsquigarrow_{PHP}$ reduce secciones de código

$\trianglelefteq\trianglerighteq_{\texttt{PHP}}$ y transforma una tupla del dominio semántico en otra.

Si es necesario introducir alguna modificación específica a las definiciones semánticas previamente realizadas, se aclarará en el momento de hacerlo.

4.3. PHP es open source y un lenguaje general

Una característica a destacar cuando estudiemos este lenguaje es que el código fuente es abierto, es decir, disponible para agregar funcionalidades o modificar las ya existentes y proponer los cambios a la comunidad PHP. Esto hace que PHP esté en constante evolución para beneficio de sus usuarios.

Otra característica a tener en cuenta es que PHP no es sólo un lenguaje para generar contenido dinámico, sino que los desarrolladores del lenguaje también incluyeron módulos con librerías con los más diversos propósitos: conectarse con una gran variedad de fuentes de datos, tratar con imágenes, archivos PDF, animaciones, e interactuar con distintas tecnologías entre las que se incluyen objetos COM y Java. PHP tiene funciones incorporadas para tratamiento de texto (strings) y distintos tipos de datos, y se encuentran muy bien documentadas en el manual del usuario y en la url `www.php.net`. Además el lenguaje es extremadamente flexible y portable, dado que existen implementaciones para distintos sistemas operativos (Windows, Linux, ...) y web servers (Apache, IIS, ...).

4.4. Reglas para interpretación de un script PHP

Se definirá en esta sección la relación de reducción $\rightsquigarrow_{PHP}$, que reduce secciones de código PHP. Estas se delimitan con "<?*php*" y "? >" por defecto, aunque pueden utilizarse también los mismos símbolos de apertura y cierre que en ASP "< %" y "% >", configurando una variable apropiadamente.

No existe declaración de variables en PHP: al realizar una asignación se infiere el tipo o se utiliza un tipo unión, como veremos en la regla $VarDec$. Las reglas para el tratamiento del texto plano (o HTML), comentarios y algunas otras resultan similares a las de ASP, hecho por el cual simplemente se las nombrará y se solo se hará un breve comentario.

4.4.1. Tratamiento de texto plano y secciones HTML

El tratamiento del texto plano y HTML es exactamente igual al caso ASP: se copian todos los caracteres (incluidos tabuladores y símbolos especiales) en la salida sin producir cambio en el contexto o ambiente.

$$TextRule \frac{}{\begin{array}{l}([H, F, \trianglelefteq \texttt{S} \trianglerighteq_{\texttt{HTML}}], Env, Store, Out) \\ \rightsquigarrow_{PHP} ([H, F, \trianglelefteq\trianglerighteq_{\texttt{PHP}}], Env, Store, Out \oplus S)\end{array}}$$

Todo lo que no esté dentro de una sección PHP, es decir texto y tags HTML, se dejará en la salida para un posterior procesamiento (renderización) por parte de un browser HTML.

4.4.2. Comentarios

Los comentarios en PHP son al estilo C, los de una sola línea se indican con doble barra "//" o con el símbolo "#", y los de múltiples líneas se delimitan con "*/" para apertura y "/*" para fin de comentario.

Las reglas, simplemete descartan el texto contenido en la sección indicada por los delimitadores de comentario.

$$SLComm \frac{}{\begin{array}{l}([H, F, \text{"//"} \oplus S \oplus \text{" } < NL > \text{"}], Env, Store, Out) \\ \rightsquigarrow_{PHP} ([H, F, \trianglelefteq\trianglerighteq_{\texttt{PHP}}], Env, Store, Out)\end{array}}$$

el símbolo $< NL >$ que aparece aquí representa el caracter de nueva línea. Para el caso de comentario multilínea se tiene:

$$MLComm \frac{}{\begin{array}{l}([H, F, \text{" } * \text{/"} \oplus S \oplus \text{"/ } * \text{"}], Env, Store, Out) \\ \rightsquigarrow_{PHP} ([H, F, \trianglelefteq\trianglerighteq_{\texttt{PHP}}], Env, Store, Out)\end{array}}$$

en donde es pertinente recordar que S es cualquier secuencia.

4.4.3. Tratamiento de variables

PHP ha evolucionado con respecto a ASP en el sentido de que ya no se deja la posibilidad al usuario sobre si debe o no definir variables: la decisión tomada es que nunca se definan variables, ya que estas se definen implícitamente cuando se les asigna un valor o una expresión, desde donde puede inferirse el tipo correspondiente.

Las variables en PHP se referencian mediante un signo dólar seguido de su nombre y son case-sensitive. Pueden asignarse valores y referencias al estilo C (con un operador que derreferencia el valor).

$$VarDec\frac{\begin{array}{l}x \notin Env,\\ (e, Env, Store) \Longrightarrow (v :: T, Env', Store')\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \texttt{\$x} = \texttt{e} \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP}\\ ([H, F, \trianglelefteq\trianglerighteq_{\texttt{PHP}}], Env' \Leftarrow^{+} [\text{"}x\text{"} : n = newCap() :: T],\\ Store' \Leftarrow^{+} [n : v], Out)\end{array}}$$

y para el caso de una variable ya definida, tenemos la regla de asignación:

$$VarAss\frac{\begin{array}{l}(e, Env, Store) \Longrightarrow (v :: T, Env', Store'),\\ [\text{"}x\text{"} : n :: T] \in Env\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \texttt{\$x} = \texttt{e} \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP}\\ ([H, F, \trianglelefteq\trianglerighteq_{\texttt{PHP}}], Env', Store' \Leftarrow^{\checkmark} [n : v], Out)\end{array}}$$

con lo que hemos aprendido hasta este punto, estas reglas deberían ser autoexplicativas.

En PHP4 se agrega la posibilidad de asignar una referencia de otra variable, es decir, de crear un *alias* explícitamente:

$$AssRef\frac{[\text{"}x\text{"} : n :: T] \in Env}{\begin{array}{l}([H, F, \trianglelefteq \texttt{\$y} = \&\texttt{\$x} \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP}\\ ([H, F, \trianglelefteq\trianglerighteq_{\texttt{PHP}}], Env\\ \Leftarrow^{+} [\text{"}y\text{"} : n :: T], Store, Out)\end{array}}$$

Variables con nombre variable

En ciertos casos, es conveniente tener nombres de variables que puedan variar en tiempo de ejecución, y PHP permite hacer esto utilizando un doble signo pesos, como se muestra en las siguientes reglas:

$$AssVarVarDef\frac{\begin{array}{l}["x" : n :: String] \in Env \\ [n : s] \in Store \\ [s : \_ :: \_] \notin Env \\ (e, Env, Store) \Longrightarrow (v :: T, Env', Store'),\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \$\$\mathtt{x} = \mathtt{e} \trianglerighteq_{\mathtt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP} \\ ([H, F, \trianglelefteq\trianglerighteq_{\mathtt{PHP}}], Env' \Leftarrow^{+} [s : m = newCap() :: T], \\ Store' \Leftarrow^{+} [m : v], Out)\end{array}}$$

cuando la variable ya se encuentra definida, se reemplaza su valor:

$$AssVarVarUpd\frac{\begin{array}{l}["x" : n :: String] \in Env \\ [n : s] \in Store \\ [s : m :: T] \in Env \\ (e, Env, Store) \Longrightarrow (v :: T', Env', Store'),\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \$\$\mathtt{x} = \mathtt{e} \trianglerighteq_{\mathtt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP} \\ ([H, F, \trianglelefteq\trianglerighteq_{\mathtt{PHP}}], Env' \Leftarrow^{\checkmark} [s : m' = newCap(T') :: T'], \\ gc(Store') \Leftarrow^{+} [m' : v], Out)\end{array}}$$

Aquí se asigna en el ambiente Env una nueva locación del store m' a la variable $\$s$ (donde s es el contenido de $\$x$); de esta manera, la locación m del store deja de ser referenciada desde el ambiente. Luego, cuando se invoca a $gc()$ se limpia dicha locación m entre todas las locaciones no referenciadas. Opcionalmente y por razones de eficiencia, podría limpiarse sólo la zona de memoria que ya no será utilizada con una operación similar a $dispose(n, size(T))$, donde n es una dirección del store y $size(T)$ es el tamaño del área que se desea limpiar.

Se utiliza la operación $\Leftarrow^{+} [m' : v]$ para agregar la nueva zona de memoria m' capáz de albergar al objeto v.

Variables Apache

Para desarrollar una aplicación web, será indispensable acceder a cierta información del entorno y del web server de alguna manera. En el caso de utilizar la dupla PHP-Apache, contamos con un conjunto de variables provistas por el lenguaje PHP exclusivamente para permitir esta comunicación con el web server.

Estas variables son creadas por el web-server Apache y pueden accederse desde los scripts PHP; el intérprete PHP se encarga de mantenerlas y administrarlas en el ambiente de trabajo para el momento en que sean requeridas. Daremos una breve descripción de algunas de ellas, ya que mediante el acceso a éstas se provee funcionalidad semejante a la provista por algunos de los mensajes a objetos predefinidos e incorporados en ASP (*objetos built-in*).

$SERVER_NAME: Es el nombre del server en el cual el script actual está corriendo.

$QUERY_STRING: Se provee acceso al string que produjo la llamada al archivo actual. A diferencia de ASP, no es un array de strings, sino que es el string mismo, y si se quiere obtener los nombres o valores de los parámetros se debe utilizar funciones para descomponer y tratar todo el string.

$REMOTE_PORT: Es el número de puerto mediante el cual el usuario se está comunicando con el server.

$SERVER_PORT: Es el número de puerto por el cual el server recibe los requerimientos HTTP. Normalmente es el puerto número 80.

$SCRIPT_FILENAME: Es el nombre completo del archivo que se está ejecutando actualmente.

Una regla para el chequeo de estas variables, sería similar a:

$$lookUpApaVar \frac{\begin{array}{rl} \$var \in & \{\ \$SERVER\_NAME, \\ & \$QUERY\_STRING, \\ & \$REMOTE\_PORT, \\ & \$SERVER\_PORT, \\ & \$SCRIPT\_FILENAME\} \\ (v :: T, Env', Store') = & lookUpApache(\$var, Env, Store), \end{array}}{\begin{array}{c} ([H, F, \trianglelefteq\ \texttt{\$var}\ \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP} \\ ([H, F, \trianglelefteq\ \texttt{v :: T}\ \trianglerighteq_{\texttt{PHP}}], Env', Store', Out) \end{array}}$$

cabe aclarar aquí que la función *lookUpApache*() es la que interactúa con el web server para solicitar y actualizar el estado de las variables del web server Apache en nuestra representación de memoria $(Env, Store)$, resultando la actualización $(Env', Store')$.

Variables PHP

Mediante distintas variables provistas por PHP se provee acceso a cookies, a los parámetros con los que se invocó al script (con los cuales dotamos a los scripts de abstracción y posibilidad de reuso), a variables de estado sobre el usuario y el server. Veamos algunas de ellas:

$argv: Es un array de argumentos pasados al script. Si el script se ejecutó desde la linea de comandos, contiene acceso a los argumentos pasados, al estilo C. Si se llamó con el método GET, esta variable contiene el query string, que en este caso no hace falta tratar con funciones sobre strings, sino que se provee acceso directo a los valores.

$PHP_SELF: El nombre del script que se está ejecutando actualmente, pero relativo al documento root.

$HTTP_COOKIE_VARS: Se provee acceso a un array asociativo de valores en el browser remoto para seguimiento e identificación de los usuarios. Se puede establecer el valor de las cookies utilizando la función *setcookie*().

$HTTP_GET_VARS, $HTTP_POST_VARS, $HTTP_ENV_VARS, $HTTP_SERVER_VARS: proveen acceso a distintos arrays para chequear el estado del sistema y de los datos pasados por el usuario.

Para el caso de *argv* la regla sería análoga a la regla vista para las variables *Apache*. Para las otras variables, que involucran una indexación por nombre en una colección de variables relacionadas, podemos representar las acciones de la siguiente manera:

$$lookUpColVar\ \frac{\begin{array}{rl} \$col \in & \{\$HTTP\_GET\_VARS, \\ & \$HTTP\_POST\_VARS, \\ & \$HTTP\_ENV\_VARS, \\ & \$HTTP\_SERVER\_VARS, \\ & \$argv, \\ & \$HTTP\_COOKIE\_VARS\} \\ 'name' \in & \$col \\ [\$col['name'] : n :: T] \in & Env \\ (Env', Store') = & updatePhpVar(\$col['name'], Env, Store), \\ [n : v] \in & Store' \end{array}}{\begin{array}{c} ([H, F, \trianglelefteq \texttt{\$col['name']} \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP} \\ ([H, F, \trianglelefteq \texttt{v} :: \texttt{T} \trianglerighteq_{\texttt{PHP}}], Env', Store', Out) \end{array}}$$

aquí se verifica primero que la colección sea una de las antes mencionadas, luego se chequea e indexa por nombre y se extrae del ambiente la dirección n del store, se actualiza el valor de la variable interactuando con el intérprete PHP (obteniendo Env' y $Store'$), y luego se obtiene el valor v correspondiente del store.

4.4.4. Programación estructurada

Existe una leve variación en la forma en que se tratan las sentencias para estructurar la programación en PHP: cualquier cosa que retorne un valor es una expresión y puede ser interpretada (al igual que en C) como distintos tipos, dependiendo de la variable en que se almacene y el contexto en que se utilice.

Por ejemplo el tipo *boolean*, es un entero y puede ser tratado como tal; pero si se lo utiliza como una guarda booleana, si es cero significa el valor $false$ y si es positivo debe interpretarse como *true*. Por supuesto, el lenguaje define las constantes *true* y $false$ como 1 y 0 respectivamente, para que nos abstraigamos del hecho de que se trata en realidad de números enteros.

$$if_{True}\frac{(e, Env, Store) \Longrightarrow (true, Env', Store'),}{\begin{array}{l}([H, F, \trianglelefteq \texttt{if(e)}\ \{\trianglerighteq_{\texttt{PHP}} \\ \quad \oplus S_1 \oplus \\ \trianglelefteq\}\texttt{else}\{\trianglerighteq_{\texttt{PHP}} \\ \quad \oplus S_2 \oplus \\ \trianglelefteq\}\ \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP} \\ ([H, F, S_1], Env', Store', Out)\end{array}}$$

aquí, a diferencia de la regla anterior vista para ASP, la expresión e puede ser una asignación, y por lo tanto, además de funcionar como guarda booleana, puede modificar el ambiente y el store, como por ejemplo en la expresión "`if (x =(1==y))...`" (recordemos que = es asignación). Análogamente para el caso falso se tiene:

$$if_{True}\frac{(e, Env, Store) \Longrightarrow (false, Env', Store'),}{\begin{array}{l}([H, F, \trianglelefteq \texttt{if(e)}\ \{\trianglerighteq_{\texttt{PHP}} \\ \quad \oplus S_1 \oplus \\ \trianglelefteq\}\texttt{else}\{\trianglerighteq_{\texttt{PHP}} \\ \quad \oplus S_2 \oplus \\ \trianglelefteq\}\ \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP} \\ ([H, F, S_2], Env', Store', Out)\end{array}}$$

Las reglas correspondientes a la sentencia de control de repetición "while" son análogas a las vistas para el caso ASP. La diferencia radica en la sintaxis tipo C: los paréntesis de la expresión booleana son obligatorios y si se escribe mas de una sentencia en el cuerpo debe incluirse una llave de apertura y otra de cierre. Cabe recordar también, que además de la condición booleana, pueden incluirse una sentencia de inicialización y otra que se ejecuta con cada iteración opcionales.

4.4.5. Funciones built-in

Existe una amplia gama de librerías de funciones definidas en el manual de PHP para incluirlas en cualquiera de nuestros scripts con distintos propósitos: acceso a datos, manipulacion de strings, imágenes, generación de salida (como *echo* y *print* o *printf*), y control de flujo de programa. Estas funciones **pueden considerarse parte del lenguaje PHP**, por estar disponibles en todo momento

y en toda versión de PHP. Puede consultarse la última definición de semántica informal de cada función haciendo una búsqueda en el sitio `www.php.net`.

Directivas para generación dinámica de salida: echo, print, printf

Las funciones *echo*, *print* y *printf* sirven para escribir en la salida que se está generando. Al igual que en ASP, existe un atajo para este tipo de directivas, que es el operador $=< exp >$:

$$OutputDir\frac{(e, Env, Store) \Longrightarrow (v :: T, Env', Store')}{\begin{array}{c}([H, F, \trianglelefteq = \texttt{e} \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP} \\ ([H, F, \trianglelefteq\trianglerighteq_{\texttt{PHP}}], Env', Store', Out \oplus toString(v))\end{array}}$$

Si se desea una regla para la función "$echo(s :: string)$", ésta es la que se obtiene reemplazando el símbolo "=" por la palabra *echo* en la regla anterior (es decir, las reglas son exactamente iguales). En cambio, la función $printf()$ provee una funcionalidad más, que es la posibilidad de incluir llamadas a función para evaluar expresiones que deben ir intercaladas en el string que se quiere mostrar. Por ejemplo:

```
printf('error num.: %s, error str.:%s', errorNum(), errorString() )
```

imprime un mensaje de error llamando a dos funciones para describir el número y el string que lo describe. Esto se hace al estilo C, dejando "huecos" de valores que se "rellenan" con los demás parámetros. Esta función está sobrecargada, porque pueden darse opcionalmente 0, 1, 2, 3 y hasta 4 parámetros de expresiones para "rellenar huecos". La regla también es similar; se incluye como precondiciones las evaluaciones de los valores de relleno y luego se concatena en la salida el string resultado. Para el caso de un solo parámetro, sería:

$$Printf\ \frac{\begin{array}{l}(e, Env, Store) \Longrightarrow (s :: String, Env', Store'),\\ s = s_1 \oplus \text{" \%}t\text{"} \oplus s_2,\\ t \in \{'s', 'i', 'f', 'd', 't', ...\},\\ f() = v :: T,\\ T = Type(t)\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \texttt{printf(e,f())} \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP}\\ ([H, F, \trianglelefteq \trianglerighteq_{\texttt{PHP}}], Env', Store', Out \oplus s_1 \oplus toString(v) \oplus s_2)\end{array}}$$

donde en este caso, la expresión $Type(t)$ retorna el tipo de PHP que corresponde a la notación " %t" de la función $printf$.

Terminación anormal de un script: exit() y die()

Cuando se detecta algún tipo de error, alguna condición particular (anormal o no), puede terminarse el procesamiento del mismo en el momento en que se llama a la función *exit*() o de igual semántica *die*(), excepto que en esta última puede incluirse un string como parámetro que representa el mensaje que se desea mostrar en la salida cuando esta condición ocurre. Por ejemplo puede utilizarse:

```
connect() or die('no se pudo conectar a la base de datos')
```

El siguiente es el esquema de la regla (no necesita precondiciones):

$$DieRule\ \frac{}{\begin{array}{l}([H, F, \trianglelefteq \texttt{die()} \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP}\\ ([H, Nil, Nil], clear(Env), gc(Store), Out))\end{array}}$$

lo que ocurre al procesar una llamada a *die*() es que se elimine el contexto de la tupla, es decir, se deje de procesar el archivo actual, y además se limpie el ambiente actual (función *clear*()) y se ejecute *gc*() para limpiar el store. La salida generada hasta ese momento permanece inalterada en

la tupla resultado, pero se deja de procesar la sección PHP actual y el archivo del contexto pasa a ser nulo. La tupla resultante es final, es decir, no puede continuarse la reducción desde ella.

4.4.6. Construcciones sintácticas para control de flujo de una aplicación

En esta sección presentamos algunas construcciones provistas por PHP, indispensables para controlar el flujo de una aplicación y para una mejor organización de los archivos y el código que contienen, lo cual genera la posibilidad de crear librerías de funciones o secciones de código reusables en diferentes aplicaciones.

Quizás ésta sea un punto débil de PHP, pues se mezclan algunos conceptos diferentes, como la semántica de la cláusula de inclusión de C `#include` o el mecanismo que algunos lenguajes llaman sustituciones macro, con un mecanismo para conseguir hacer redirecciones y llamadas de un script a otro.

Las principales construcciones para comunicar información o simplemente desviar el procesamiento de un archivo a otro son: *require*(), *require_once*(), *include*(), *include_once*().

También puede utilizarse con fines de redirección la función *header*(), que envía un encabezado HTML (en particular utilizando como parámetro "Location:URL" para simular una redirección).

Intentaremos describir y diferenciar la semántica de las cuatro funciones anteriores, mediante reglas semánticas.

$$Inc\frac{\begin{array}{l}F_2 = \trianglelefteq\ \mathtt{S}\ \trianglerighteq_{\mathtt{PHP}} \\ ([H_1, F_2, \trianglelefteq\ \mathtt{S}\ \trianglerighteq_{\mathtt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP} \\ \quad ([H_1, F_2, \trianglelefteq\trianglerighteq_{\mathtt{PHP}}], Env', Store', Out \oplus Out')\end{array}}{\begin{array}{l}([H_1, F_1, \trianglelefteq\ \mathtt{include}(\mathtt{'http://H_2/F_2\ '})\ \trianglerighteq_{\mathtt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP} \\ \quad ([H_1, F_1, \trianglelefteq\trianglerighteq_{\mathtt{PHP}}], Env', Store', Out \oplus Out')\end{array}}$$

algunos puntos a mencionar en esta regla:

- F_2 contiene la secuencia S_2 que puede ser vista puramente como código PHP
- El archivo F_2 puede residir en un server distinto H_2

- La ejecución del script F_2 en el server H_1 solamente agrega caracteres en la salida (no puede alterar o eliminar nada de la secuencia Out anteriormente generada)

veamos ahora las diferencias que existen con su regla análoga *include_once*():

$$IncOnce\frac{\begin{array}{l}F_2 = \trianglelefteq \texttt{S} \trianglerighteq_{\texttt{PHP}},\\ ([H_1, F_2, \trianglelefteq \texttt{S} \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP}\\ \quad ([H_1, F_2, \trianglelefteq\trianglerighteq_{\texttt{PHP}}], Env'', Store'', Out \oplus Out')\\ D = set\ of\ all\ declarations\ in\ S,\\ Env' = Env \Leftarrow^{+} [d : n = newCap() :: Type(d)] \ \wedge\\ \quad Store' = Store \Leftarrow^{+} [n : Val(d)], \quad (\forall d \in D\ st : d \notin Env),\end{array}}{\begin{array}{l}([H_1, F_1, \trianglelefteq \texttt{include\_once}(\texttt{'http://H}_2\texttt{/F}_2\texttt{'}) \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP}\\ \quad ([H_1, F_1, \trianglelefteq\trianglerighteq_{\texttt{PHP}}], Env', Store', Out \oplus Out')\end{array}}$$

la principal diferencia es que si $Env = Env'$ (es decir, no existen declaraciones nuevas en F_2), el input "*include_once*(...)" se consume sin producir ningún cambio en el ambiente Env o el store $Store$.

A continuación, la regla para *require*():

$$Req\frac{\begin{array}{l}S = S_1 \oplus \trianglelefteq \texttt{require}(\texttt{'http://H}_2\texttt{/F}_2\texttt{'}) \trianglerighteq_{\texttt{PHP}} \oplus S_2,\\ F_2 = \trianglelefteq \texttt{Q} \trianglerighteq_{\texttt{PHP}},\\ ([H_1, F_1, S_1 \oplus \trianglelefteq \texttt{Q} \trianglerighteq_{\texttt{PHP}} \oplus S_2], Env, Store, Out) \rightsquigarrow_{PHP}\\ \quad ([H_1, F_1, \trianglelefteq\trianglerighteq_{\texttt{PHP}}], Env', Store', Out \oplus Out')\end{array}}{\begin{array}{l}([H_1, F_1, \trianglelefteq \texttt{S} \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP}\\ \quad ([H_1, F_1, \trianglelefteq\trianglerighteq_{\texttt{PHP}}], Env', Store', Out \oplus Out')\end{array}}$$

y la regla para *require_once*()

$$ReqOnce\frac{\begin{array}{l}S = S_1 \oplus \trianglelefteq \texttt{require\_once}('\texttt{http}://\texttt{H}_2/\texttt{F}_2\ ') \trianglerighteq_{\texttt{PHP}} \oplus S_2,\\ F_2 = \trianglelefteq\ \texttt{Q}\ \trianglerighteq_{\texttt{PHP}},\\ D = set\ of\ all\ declarations\ in\ Q,\\ Env' = Env \Leftarrow^{+} [d : n = newCap() :: Type(d)]\\ Store' = Store \Leftarrow^{+} [n : Val(d)] \quad \forall d \in D\ st : d \notin Env,\\ ([H_1, F_1, S_1 \oplus \trianglelefteq\ \texttt{Q}\ \trianglerighteq_{\texttt{PHP}} \oplus S_2], Env, Store, Out) \rightsquigarrow_{PHP}\\ \quad ([H_1, F_1, \trianglelefteq\trianglerighteq_{\texttt{PHP}}], Env', Store', Out \oplus Out')\end{array}}{\begin{array}{l}([H_1, F_1, \trianglelefteq\ \texttt{S}\ \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP}\\ \quad ([H_1, F_1, \trianglelefteq\trianglerighteq_{\texttt{PHP}}], Env', Store', Out \oplus Out')\end{array}}$$

la diferencia principal entre *require* e *include*, es que cuando se utiliza la primera siempre se intentará leer el archivo especificado y en caso de no encontrarse se generará un error porque el archivo *debe* procesarse; en cambio cuando se utiliza *include*, esta podría estar en una rama de un if que no se evalúa por ejemplo, y entonces se ignora. Un defecto de la última regla, es que no se adapta a lo anterior, es decir, no se puede obligar a que se procese el *require*() obligatoriamente.

La combinación de una de las cuatro construcciones anteriores seguida de *exit*() o *die*() tiene el efecto de transferir el control a otro archivo .php, mientras que la ejecución de ellas por si solas tiene un efecto parecido al de llamada a un procedimiento, aunque sin pasaje de parámetros y pudiendo referenciar al ambiente global en el momento de la invocación.

4.4.7. Funciones definidas por el usuario

En cualquier parte de un script, pueden definirse funciones con una sintaxis muy parecida a la de C++. Las principales características de las fdu, son pasaje de parámetros por valor y por referencia, la posibilidad de incluir valores default en los parámetros y también listas de parámetros de longitud variable.

Incluiremos tres reglas para las fdu:

$$LoadFunDef\frac{\begin{array}{l}[\text{"}f\text{"} : \_ :: FunType] \notin Env,\\ \trianglelefteq \texttt{function f(formals)}\{\trianglerighteq_{\texttt{PHP}} \oplus S_f \oplus \trianglelefteq\} \trianglerighteq_{\texttt{PHP}} \in F\end{array}}{\begin{array}{l}([H,F,S], Env, Store, Out) \rightsquigarrow_{PHP}\\ ([H,F,S],\\ Env \Leftarrow^{+} [\text{"}f\text{"} : n = newCap() :: FunType],\\ Store \Leftarrow^{+} [n : \text{"}function\ f(formals)\{\text{"} \oplus S_f \oplus \text{"}\}\text{"}],\\ Out)\end{array}}$$

$$FunCall\frac{\begin{array}{l}[\text{"}f(formals)\text{"} : N : FunDef] \in Env,\\ (actuals[i], Env, Store) \Longrightarrow ([v_i :: T_i], Env', Store'),\\ [N : \text{"}f(formals)\{\text{"} \oplus S \oplus \text{"}\}\text{"}] \in Store\end{array}}{\begin{array}{l}([H,F,\trianglelefteq \texttt{f(actuals)} \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \rightsquigarrow_{ASP}\\ \quad([H,F,\trianglelefteq \texttt{S}_\sigma \oplus \text{"}\texttt{returnFrom}\text{ "} \oplus \texttt{FunInvocID} \trianglerighteq_{\texttt{ASP}}],\\ \quad Env\\ \quad \Leftarrow^{+}_{\forall x \in LV(\{S\})} [FunInvocID \oplus \text{"}\_x\text{"} : Nil :: NilType]\\ \quad \Leftarrow^{+}_{\forall p_i \in \{formals\}} [FunInvocID \oplus \text{"}\_\text{"} \oplus p_i = formals[i] : v_i :: T_i], Store, Out)\end{array}}$$

$$\begin{array}{ll} where & FunInvocID = \text{"}\#f(formals)\text{"} \oplus toString(invocNum(\#f(formals)))\\ & \sigma = \{x/FunInvocID \oplus \text{"}\_x\text{"}\}_{\forall x \in LV(\{S\}) \cup \{formals\}}\end{array}$$

$$FunRet\frac{}{\begin{array}{l}([H,F,\trianglelefteq \texttt{returnFrom f(formals)N} \trianglerighteq_{\texttt{ASP}}], Env, Store, Out) \rightsquigarrow_{ASP}\\ \quad([H,F,\trianglelefteq\trianglerighteq_{\texttt{PHP}}], removeAll(Env, \text{"}\#f(formals)N\_\text{"}), gc(Store), Out)\end{array}}$$

las cuales son muy similares a las vistas para ASP, excepto por el hecho de que no se define el tipo de retorno.

Un concepto innovador en PHP es la posibilidad de definir variables de función. Una vez que se tiene la definición cargada en el ambiente, puede utilizarse una variable de funcion (aquí llamada $func$) para referenciarla. Para comprender como funcionan, veamos la regla correspondiente:

$$FunVar\frac{\begin{array}{c}[\text{"}f\text{"} : n :: FunType] \in Env,\\ [\text{"}func\text{"} : m :: T] \in Env\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \texttt{\$func} =' \texttt{f}' \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP}\\ \quad ([H, F, \trianglelefteq\trianglerighteq_{\texttt{PHP}}], Env \Leftarrow^{\checkmark} [\text{"}func\text{"} : n :: FunType],\\ Store, Out)\end{array}}$$

y cuando se necesite invocar a la función referenciada, simplemente se llama a $func()$ y se ejecutará el código de $f()$. Cabe recordar en esta regla que la operación de actualización $\Leftarrow^{\checkmark}$ aplicada al ambiente funciona por nombre y que la misma operación aplicada al store funciona por posición de memoria. En este caso, los nombres de variable $\$func$ y $\$f$ son alias.

4.4.8. Variantes para manipulación de Objetos en scripts PHP

Objetos provistos PHP

Para PHP, una clase es una colección de variables y funciones que funcionan juntas. PHP agrega cierta funcionalidad extra a las clases que ya vimos para *JScript*: constructores, herencia, un mecanismo para definir métodos de clase, las palabras clave $this, parent, extends$, etc. Estos mecanismos son rudimentarios y distan de ser objetos realmente reusables como piezas de software independientes e inteligentes. Sin embargo, son una buena herramienta de desarrollo cuando los plazos de entrega son cortos y las interrelaciones entre los objetos de las aplicaciones son relativamente sencillas. Veamos la sintaxis que utiliza PHP, y discutamos algunas posibilidades de uso.

Una clase se define con la siguiente sintaxis:

```
<?php
class C {
  var $c_1
  var $c_2
  ...
  var $c_n
    function C() {...}//constructor
    function f_1(...) {...}
    function f_2(...) {...}
     ...
    function f_n(...) {...}
} ?>
```

Cuando se quiere instanciar un objeto, se declara una variable y se utiliza la función *new* y se accede a propiedades y funciones mediante el operador $->$:

```
<?php
   $c = new C();
   $c->c_1  = e
   $c->f_3()
 ?>
```

Las reglas que manipulan estos objetos son simples definiciones de variables y funciones como ya hemos visto, que se identifican e invocan de una manera particular, y que están agrupadas y encapsuladas dentro de una misma definición, que se da al definir una clase.

Presentamos a continuación dos reglas para contemplar esta construcción del lenguaje:

$$PhpClassDef\frac{\begin{array}{l}C(formals) \notin Env,\\ d = phpClassDescriptor(C(formals), locvars, mfuncs)\end{array}}{\begin{array}{l}([H, F, \trianglelefteq\ \texttt{class C\{C(formals), locvars, mfuncs\}} \trianglerighteq_{\texttt{PHP}}],\\ \quad Env, Store, Out) \rightsquigarrow_{PHP}\\ ([H, F, \trianglelefteq\trianglerighteq_{\texttt{PHP}}],\\ Env \Leftarrow^{+} [\text{"}C(formals)\text{"} : n = newCap(size(C())) :: PhpClass],\\ Store \Leftarrow^{+} [n : d],\\ Out)\end{array}}$$

$$ObjInst\frac{\begin{array}{l}c \notin Env,\\ D = C(formals, locvars, mfuncs) \in Store,\\ check(formals, params)\end{array}}{\begin{array}{l}([H, F, \lhd \texttt{\$c = new C(params)};\rhd_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP}\\ ([H, F, \lhd\rhd_{\texttt{PHP}}],\\ Env \Leftarrow^{+} [\text{"}c\text{"} : n = newCap(D) :: PhpObj],\\ Store \Leftarrow^{+}_{\forall i \in locvars} [n + \sum_{k=2}^{i}(size(v_k)) : \{\$c- > v_i\}],\\ Out)\end{array}}$$

en donde, cada variable local (o de instancia) de la clase C se identifica anteponiendo el nombre de la variable que contiene la instancia y el operador de referencia de métodos o variables de instancia de una clase "$- >$" seguido del nombre de la variable de instancia referenciada. Las definiciones (o el código) del constructor de la clase y de los métodos, se guardan en el descriptor de la clase en el store. De esta manera el código solo se almacena una vez para todas las instancias. Los datos que se necesitan para almacenar e identificar una instancia son sólo sus variables de instancia, y ésto es lo que se guarda al crear una, con el método $new()$.

Componentes COM

Si nos encontramos en un entorno Windows, el acceso provisto a componentes COM de PHP puede resultar muy útil. PHP provee la función COM, que permite instanciar objetos disponibles en el server desde cualquier script. Por ejemplo, podemos instanciar la aplicación Microsoft Word mediante la creación de un objeto COM:

```
$word = new COM('word.application') or die('Word no está disponible');
print 'Word cargado... versión {$word->Version}\n';
$word->Visible = 1 //trae la aplicación a primer plano
$word->Documents->Add(); //crea un nuevo documento
$word->Documents[1]->SaveAs('prueba.doc');
$word->Quit()
$word->Release()
$word = null
```

De manera similar al caso ASP, se puede instanciar una conexión ADO para bases de datos:

```
$conn = new COM('ADODB.Connection') or die('No se puede iniciar ADO');
$conn->Open('Provider=SQLOLEDB; Data Source=localhost;');
$rs = $conn->Execute('SELECT * FROM table'); //devuelve un recordset
```

La regla para esta función provista por el lenguaje crea una variable para almacenar un *descriptor* con los **nombres de propiedades y métodos** disponibles y **referencias a** direcciones de memoria (ajenas a la manejada por PHP) que contienen el **código de dichos métodos** y **valores de las propiedades de la instancia**. Por ejemplo:

$$ObjCreCOM\frac{\begin{array}{l}[\text{"}v\text{"} : \_ :: \_] \notin Env,\\ checkCOMDeclaration(className, H),\\ D = createComDescriptor(className)\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \$\mathtt{v} = \mathtt{new\ COM(className)} \trianglerighteq_{\mathtt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP}\\ \quad([H, F, \trianglelefteq\trianglerighteq_{\mathtt{PHP}}],\\ \quad\ Env \Leftarrow^{+} [\text{"}v\text{"} : n = newCap(size(D)) :: ComObj],\\ \quad\ Store \Leftarrow^{+} [n : D], Out)\end{array}}$$

en esta regla, se chequea que la clase COM que se quiere instanciar exista en el server, y luego se almacena en el store el descriptor para esa clase COM, mediante una función interna a PHP llamada $createCOMDescriptor()$. Cabe destacar que la memoria alocada para almacenar **la instancia** propiamente dicha **se encuentra en el server** pero es administrada por una aplicación ajena a PHP, **aunque PHP debe mantener el descriptor** de la instancia.

Otra diferencia importante con respecto los objetos PHP que vimos anteriormente es que **se**

necesita un descriptor por cada instancia de un objeto COM; esto es porque el descriptor debe contener referencias a los valores de las propiedades, que son propias de cada instancia (no así el código de los métodos, que se comparte para todas las instancias de una clase).

El descriptor permite chequear en tiempo de ejecución que los métodos invocados estén en la interface del objeto y que los parámetros estén bien tipados, agilizando así la invocación de un método. De otra manera, PHP debería interactuar con una aplicación que realizara estos chequeos.

Cuando se desee invocar en las instancias uno de los métodos que una clase posee, se utiliza el descriptor almacenado en la regla anterior de la siguiente manera:

$$MetInvCOM\frac{\begin{array}{l}[\text{"}v\text{"} :: n : ComObj] \in Env,\\ [n : D = desc[props,\ funcs]] \in Store,\\ m(formals) :: T \in funcs,\\ check(actuals,\ formals),\\ addr = pointerAddress(D, m(actuals)),\\ e :: T = callCOM(addr),\end{array}}{\begin{array}{c}([H, F, \trianglelefteq \texttt{\$v}- > \texttt{m(actuals)} \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP}\\ ([H, F, e :: T], Env, Store, Out)\end{array}}$$

en esta regla se chequea que la definción de la variable se encuentre en el ambiente, luego se extrae el descriptor del store, se controla que el método invocado exista y que los parámetros enviados se correspondan en tipo y cantidad con los formales, se extrae la dirección de memoria que contiene el código del método y luego se invoca a la función *callCOM* que se encarga de invocar el método almacenado en esa dirección. Posteriormente, se recoge el valor devuelto por esta función que es el resultado de la expresión dentro de la sección de código PHP.

Cómo instanciar objetos Java

De manera similar a la instanciación de objetos COM, podemos instanciar objetos Java mediante una función provista por PHP: *Java*.

Simplemente, se invoca a la función nombrando la clase (que debe ubicarse en algún directorio nombrado dentro de la variable $CLASSPATH$ de PHP), y se crea un descriptor con los métodos y propiedades de la clase nombrada. El siguiente código graficará la simplicidad de lo anterior:

```
<?php
$system = new Java('java.lang.System');
 print 'OS='.$system->getProperty('os.name');
```

La siguiente regla, permite mostrar este comportamiento:

$$ObjCreJava\frac{\begin{array}{l}[\text{"}v\text{"} : \_ :: \_] \notin Env,\\ byteCodes = lookFor(className),\\ desc[props, funcs] = phpDescriptor(byteCodes)\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \ \$\texttt{v} = \texttt{new Java(className)} \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP}\\ \quad([H, F, \trianglelefteq\trianglerighteq_{\texttt{PHP}}],\\ \quad Env \Leftarrow^{+} [\text{"}v\text{"} : n = newCap(size(desc[])) :: JavaObj],\\ \quad Store \Leftarrow^{+} [n : desc[props, funcs]],\\ \quad Out)\end{array}}$$

$$\begin{array}{l}donde\ props = \{p_1, p_2, .., p_n\},\\ \quad funcs = \{f_1(), f_2(), .., f_n()\}\end{array}$$

la mayor parte del trabajo de esta operación se delega a $lookFor()$, que busca entre las clases registradas en el ambiente, lo que implica la búsqueda del archivo ".class" en el sistema de archivos, y luego mediante la función $phpDescriptor(bytecodes)$ se obtiene un descriptor PHP para la clase Java a partir de la interpretación de los bytecodes de la definición de la clase. Este descriptor servirá de interface con la *jvm* para invocar los métodos de la instancia.

Una vez que se ha almacenado la descripción de una clase en el ambiente PHP, pueden invocarse métodos con la siguiente sintaxis:

```
$javaObj->doSomething(param)
```

la regla semántica que describe lo que hace PHP en este caso, es la siguiente:

$$MetInvJava\frac{\begin{array}{l}["v" :: n : JavaObj] \in Env,\\ [n : D = desc[props,\ funcs]] \in Store,\\ m(formals) :: T \in funcs,\\ check(actuals,\ formals),\\ addr = pointerAddress(D, m(actuals)),\\ e :: T = callJVM(addr),\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \$\mathtt{v}- > \mathtt{m(actuals)} \trianglerighteq_{\mathtt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP}\\ ([H, F, e :: T], Env, Store, Out)\end{array}}$$

aquí lo nuevo de la regla con respecto a las anteriores es la interacción con la $JavaVirtualMachine$, mediante la función $callJVM()$ que se necesita para ejecutar el método solicitado sobre la instancia activa.

La última regla relacionada con la interacción con Java es la de acceder a propiedades públicas sin la utilización de un método que provea su encapsulamiento.

$$PropAccJava\frac{\begin{array}{l}["v" :: n : JavaObj] \in Env,\\ [n : D = desc[props,\ funcs]] \in Store,\\ p :: T \in props,\\ addr = pointerAddress(D, p),\\ e :: T = callJVM(addr),\end{array}}{\begin{array}{l}([H, F, \trianglelefteq \$\mathtt{v}- > \mathtt{getProperty}('\mathtt{p}') \trianglerighteq_{\mathtt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP}\\ ([H, F,\ e :: T], Env, Store, Out)\end{array}}$$

esta regla debería ser comprensible a partir de las anteriores.

4.4.9. Conectividad con Bases de Datos

En el manual de PHP se propone una manera de ligar scripts con distintas bases de datos. Por ejemplo, utilizando la librería disponible para $MySQL$, se propone utilizar las funciones:

```
mysql_connect(),
mysql_select\_db(),
mysql_query(),
mysql_fetch_array() o mysql_fetch_object(),
mysql_close()
```

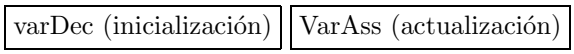
para permitir interactuar con tablas *mySql* desde scripts php. Se puede diseñar una **librería de funciones para persistencia** de objetos utilizando $mysql\_fetch\_object()$, para **"levantar" un registro de una tabla a un objeto PHP**, tratando sus campos como las variables de instancia del objeto. Luego, puede **"bajarse" el objeto a disco con una sentencia insert** apropiada.

Orientándonos hacia aplicaciones grandes, proponemos leer objetos Java o C++ a variables objeto en scripts PHP, para que el modelo abstracto conceptual de la aplicación se encargue de las interacciones con la base de datos (utilizando alguna técnica de persistencia transparente al usuario), abstrayéndose el usuario programador de esas rutinas y concentrándonos en la lógica de los scripts y utilizando los objetos de la aplicación instanciados. Un framework para persistencia transparente de objetos puede encontrarse en [Val02], para el caso de C++.

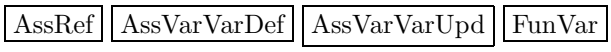
4.5. El pizarrón PHP

Al igual que en el capítulo anterior, se presenta una tabla a manera de resúmen de las características estudiadas:

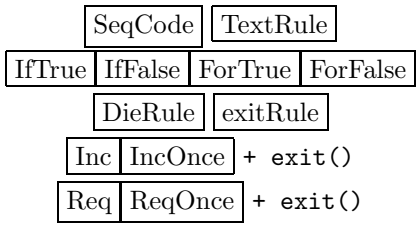
Fragmento 1: Tratamiento de Variables



Fragmento 1': Asignaciones especiales

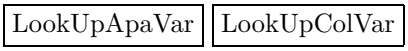


Fragmento 2: Flujo de Control

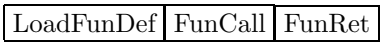


Fragmento 3: Procesamiento de salida

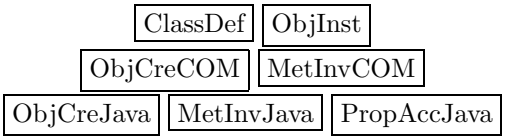
Fragmento 4: Acceso a variables de entorno



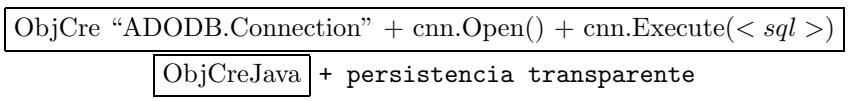
Fragmento 5: Tratamiento de procedimientos y funciones



Fragmento 6: Manipulación de objetos



Fragmento 7: Métodos de conexión con bases de datos



Capítulo 5

Marco de trabajo para aplicaciones sobre internet

WISHING

Of all amusements for the mind,
From logic down to fishing,
There is n't one that you can find
So very cheap as "wishing".
A very choice diversion too,
If we but rightly use it,
And not, as we are apt to do,
Pervert it, and abuse it.

John Godfrey Saxe (1816-1887)

5.1. Introducción

Se pretende utilizar el formalismo presentado previamente, así como los lenguajes que en él se describen, en el contexto de desarrollos de aplicaciones web.

El dominio del problema se modelará con diseños orientados a objetos y se utilizarán bases de datos para lograr persistencia de los estados de estos objetos de una sesión de usuario a otra, o de

una ejecución de la aplicación a la siguiente.

Se tomará como referencia la arquitectura orientada a objetos propuesta en [SREL99], pero con posibles modificaciones. Esta consta básicamente de 4 capas distintas:

- **Web server**, conteniendo los scripts de generación dinámica (en algún lenguaje aceptado por este server como pueden ser ASP, PHP, JSP o CGI/ISAPI, etc.)
- **Server de Navegación** que indica la forma en que se relacionan los nodos mediante links en el documento hipermedial involucrado en la aplicación y que permite el acceso a los mismos en diferentes contextos (por ejemplo en un sitio de arte un contexto podria ser "visita guiada" o "colección de Goya")
- **Modelo Conceptual** de la aplicación, que contiene el comportamiento de la aplicación expresado en algún lenguaje de programación orientado a objetos
- **Capa de Persistencia** para los objetos del modelo, que almacenan la información de todos los elementos de la aplicación en una base de datos.

La modificación básica de este modelo se hace en el server de navegación, que en vez de ser una capa independiente estará integrada en los scripts dinámicos en el web server. La eliminación de esta capa, no implica que no se haga un diseño navegacional, sino que este diseño se modelará en los distintos scripts dinámicos almacenados en el web server. Esto se logra haciendo corresponder cada nodo del modelo navegacional con un script dinámico, como se muestra en el ejemplo.

Esta modificación depende básicamente del tamaño de la aplicación: si la aplicación es realmente grande y requiere necesariamente acceso a los nodos en diferentes contextos, deberán utilizarse uno o varios servers para administrar el acceso a los nodos del modelo navegacional.

5.2. Una aplicación pequeña: La Biblioteca

Estudiaremos un ejemplo de aplicación en donde podemos utilizar la arquitectura descripta en la sección anterior. Pensemos en una aplicación sobre una biblioteca, que se utilizará para administrar préstamos de libros a sus socios que corre sobre una intranet con distintas terminales cliente desde

donde se dispararán los distintos requerimientos HTTP que responderá el servidor web. El lenguaje utilizado para programar la interface de las terminales puede ser cualquier lenguaje de generación dinámica, como por ejemplo ASP. Acotaremos un poco el problema mediante la siguiente descripción.

Descripción del problema: Una biblioteca presta libros a sus socios, si estos cumplen con las condiciones requeridas (que pueden variar de biblioteca en biblioteca, por ejemplo el pago de una cuota, o ser alumno regular en caso de una facultad) y si no tiene préstamos cuyo plazo para la devolución haya expirado.

Cuando se efectúa un préstamo, la biblioteca guarda registro de la fecha, el socio, el libro y el plazo máximo para la devolución.

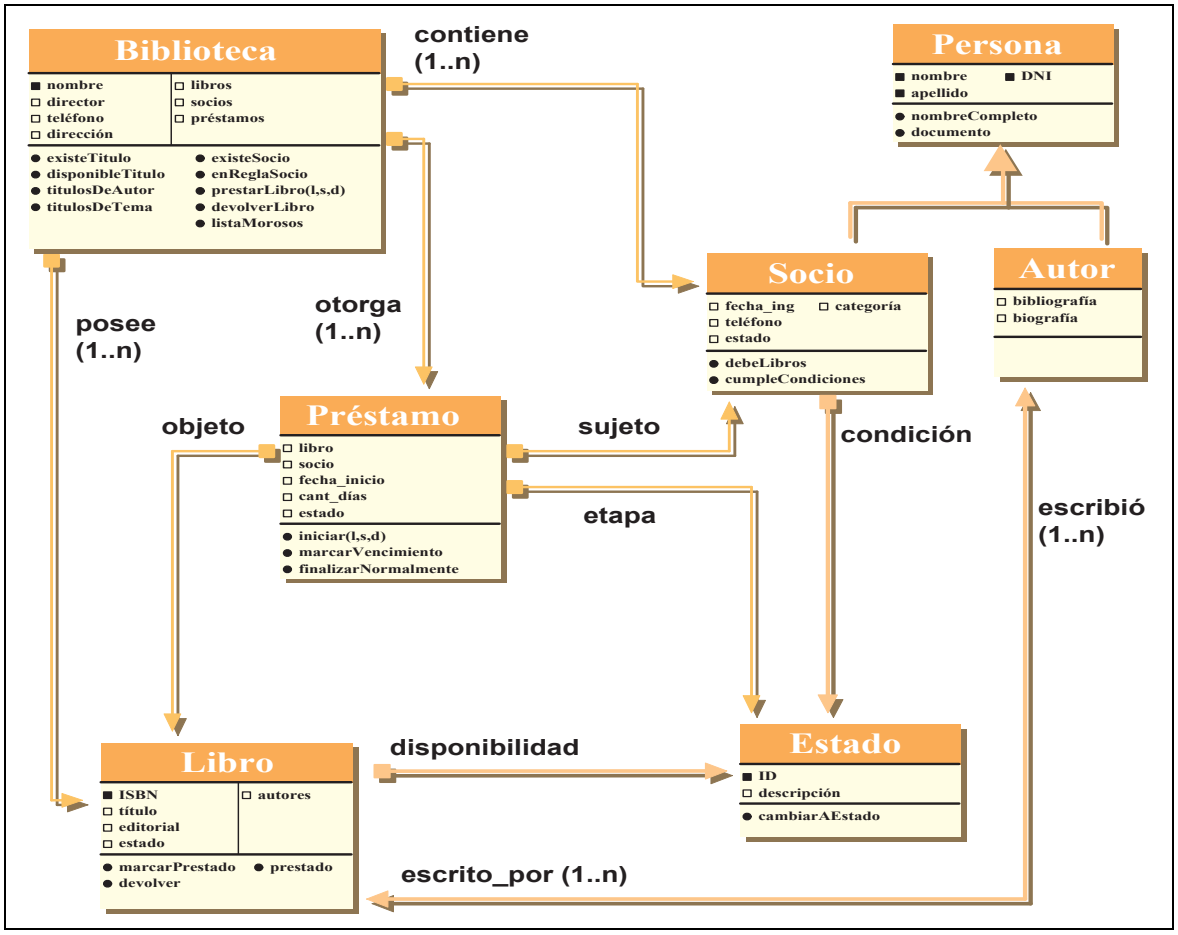
En caso de que expire este plazo máximo, se marca el préstamo como vencido, el socio como deudor y el libro como provisionalmente extraviado. En caso de que la devolución se realice normalmente se marca el préstamo como finalizado y se guarda para posteriores consultas, y se registra el libro nuevamente como disponible.

Los socios pueden clasificarse de acuerdo a un índice de confiabilidad o categoría, el cual desciende en caso de que no devuelva los libros solicitados a la biblioteca dentro del plazo estipulado. Luego pueden aplicarse políticas de prestación de libros de acuerdo a la categoría de cada socio (por ejemplo, limitando la cantidad de días para aquellos que hayan sobrepasado la fecha límite más de 3 veces).

Además se guardan registros de distintos datos de los libros, como autores, editoriales y títulos, para permitir distintas consultas sobre disponibilidad o existencia en la biblioteca de un ejemplar. Podrá realizarse periódicamente un listado conteniendo todos los socios que adeudan libros.

5.2.1. Diseño conceptual: diseño de objetos

Supongamos que se tiene el siguiente modelo conceptual del problema (formulamos esto aunque el modelo no refleje perfectamente el problema descripto en la sección anterior):



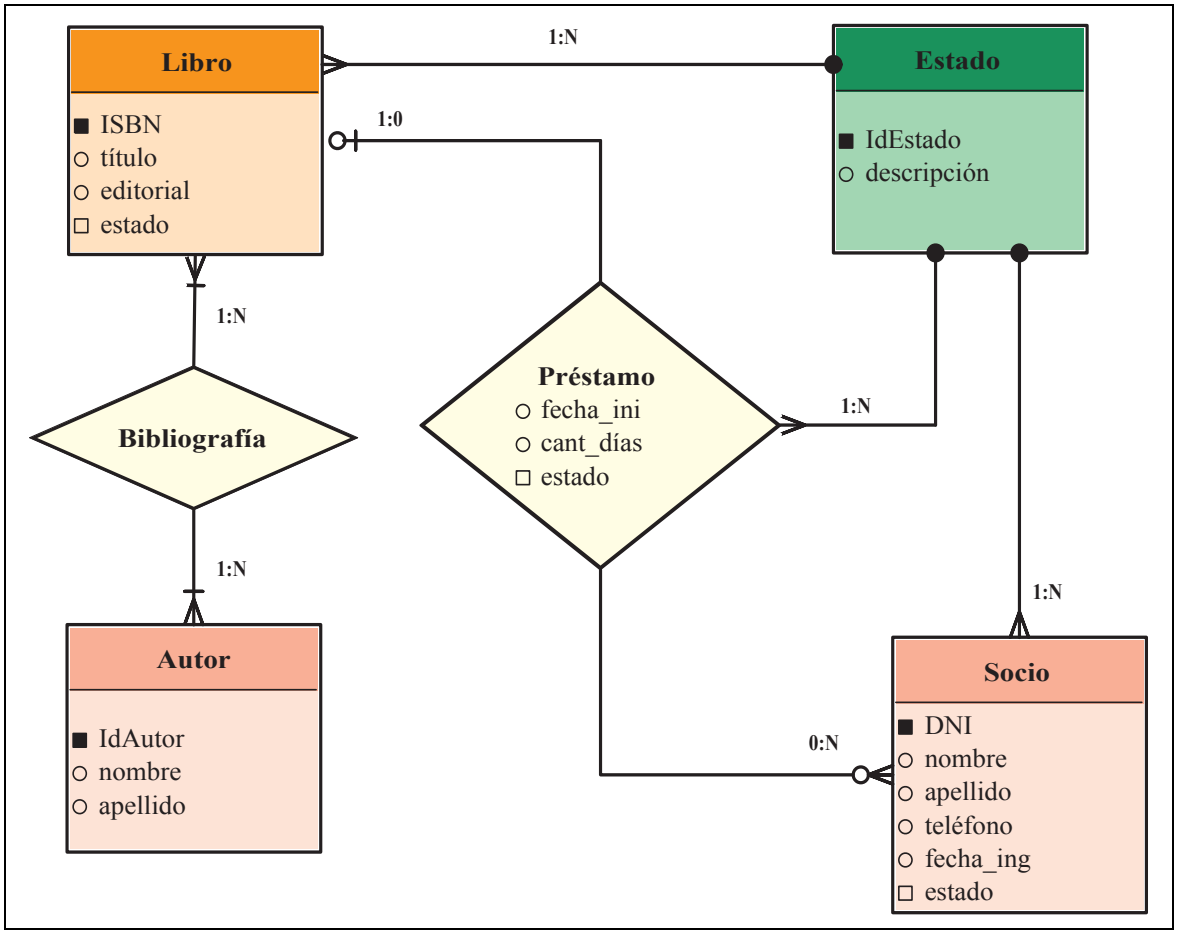
Lo que expresa el siguiente modelo conceptual de objetos podría traducirse de la siguiente manera. La **Biblioteca** contiene un conjunto de socios y un conjunto de libros que prestará. Un **Libro** es escrito por uno o varios autores y un **Autor** a su vez escribe uno o más libros, que pueden o no estar en la biblioteca. Observar que ésta relación es bidireccional, lo que significa que ambas clases contienen atributos para referenciar a la otra: la clase **Autor** contiene por ejemplo una colección de libros llamada "bibliografía", y la clase **Libro** contiene una colección donde mantiene sus autores. Los socios y los autores son subclases de **Persona** que abstrae características comunes como nombre, apellido o DNI (que sirve como identificador).

La biblioteca conoce además cuales son los préstamos otorgados actuales y pasados, mediante una colección llamada "préstamos".

Un **Préstamo** almacena su duración y su fecha de inicio e involucra un socio, un libro y un estado. Un **Estado**, desde el punto de vista de un préstamo se refiere a la etapa del préstamo: puede ser iniciado, finalizado, vencido, o finalizado con demora. Pero a su vez, cuando se referencia un estado desde el punto de vista de un socio, se está hablando sobre si cumple los requesitos para ser socio (ser alumno regular en el caso de una facultad, o estar al día con el pago de la cuota por ejemplo) y además si debe algún libro actualmente. Por otra parte, cuando se habla del estado de un libro, significa disponible, prestado o en peligro de extravío (cuando ya debería haber sido devuelto) y extraviado.

5.2.2. Modelo E-I para persistencia de objetos

De la misma manera que en la sección anterior, presentamos el siguiente modelo entidad-interrelación de bajo nivel que describe la estructura de las tablas de datos físicas, que es suficiente como para almacenar en forma precisa y organizada (si bien no podemos afirmar también que lo hace en forma normalizada) todos los datos necesarios para administrar la aplicación.



En este esquema se tienen cuatro tablas o entidades y dos interrelaciones que además de sus atributos "arrastrarán" los identificadores de las entidades que relacionan.

Es meritorio aclarar que un **Libro** es escrito por uno o varios autores, todos ellos presentes en la entidad **Autor**; a su vez, un **Autor** escribe uno o varios libros, pero algunos de estos libros pueden no figurar en la entidad **Libro**. Dicho de otra forma, el conjunto de libros existente en la entidad **Libro** es todo el conjunto de libros existentes en la biblioteca, mientras que el conjunto de autores contiene solamente aquellos que son referenciados por algún libro de la biblioteca (en su lista de autores).

La tabla de estados **Estado** visto como un conjunto de tuplas abstracto, es un conjunto de datos similar a:

$\{(1, "libro\_disponible"), (2, "libro\_prestado"), (3, "socio\_regular"),$
$(4, "socio\_moroso"), (5, "prestamo\_iniciado"), (6, "prestamo\_finalizado"),$
$(7, "prestamo\_vencido"), (8, "prestamo\_finalizado\ con\ demora")\}$

y cada uno de estos estados será referenciado por los objetos (tablas) correspondientes de libros, préstamos y socios utilizando el identificador del estado.

Considerando este modelo de persistencia (diseño E-I) para los datos de los objetos, debemos responder algunos puntos:

- 1] ¿Cómo se hace para cargar estos datos en los objetos Java o C++ del modelo de objetos conceptual del problema? es decir, ¿cómo hace un objeto para recuperar su estado al crearse en memoria, (es decir al iniciarse una sesión, por ejemplo)?
- 2] ¿Cómo y cuándo se almacenan los los datos de una sesión en la base de datos, haciendo persistentes las modificaciones?
- 3] ¿Cómo se opera desde los scripts dinámicos con los objetos del modelo conceptual?

El punto 1] debe ser respondido por el sistema de persistencia de objetos que se utilice. En este caso, se desea utilizar una herramienta similar a la desarrollada por [Val02] para C++, que solamente exige que subclasifiquemos las clases del modelo conceptual de una jerarquía que provee las interfaces necesarias para que redefinamos los métodos **load()** y **save()** que hacen posible la persistencia transparente. Recordemos que en C++ se dispone de herencia múltiple, lo que significa que la jerarquía del modelo conceptual no se vería desformada, sino enriquecida.

Para el caso de persistencia transparente en Java, se disponen herramientas comerciales para tal fin; para mayor información puede buscarse en internet.

La respuesta del punto 3] se basa en la discusión anterior sobre cómo manipular objetos desde los scripts dinámicos.

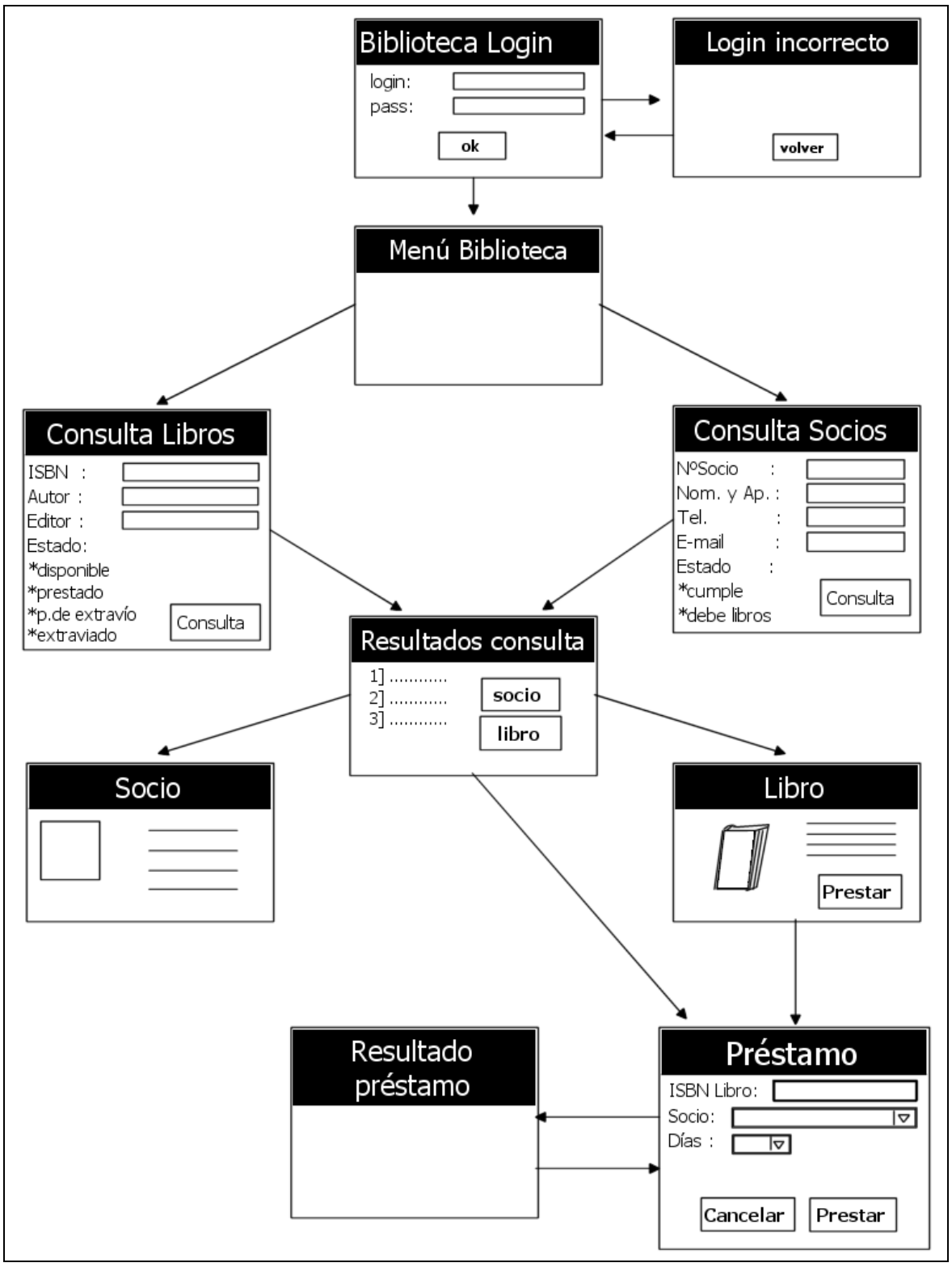
5.2.3. Modelo navegacional, organización de archivos y directorios virtuales

La interface de nuestra aplicación se compone de documentos o páginas estáticas y dinámicas, y queremos lograr que cada una se corresponda con un nodo en el diseño de navegación, para lograr una solución simple y clara.

Para los nodos con contenido estático, el archivo .asp o .php es simplemente código html; los nodos que dependen de factores, como por ejemplo el contexto desde donde se accedan o de variables que se conocen en tiempo de ejecución para mostrar su contenido, se representan con archivos .asp o .php parametrizados. Los parámetros que se utilizarán serán la sesión (un identificador único), o un conjunto mínimo de datos que permitan distinguir todos los atributos necesarios para mostrar el nodo correctamente. Si se utilizan sesiones PHP o ASP, pueden guardarse los datos de usuario dentro de variables de sesión, y utilizarse más parámetros cuando se necesita mostrar una consulta sobre la base de datos que dependa de fechas, nombres o estados.

El *caso de uso* principal de la aplicación de ejemplo, se da cuando un socio llega a la biblioteca y solicita al encargado un libro, conociendo exactamente sus datos o preguntando por un tema, autor o editorial. Se da siguiente secuencia de pasos, en el esquema navegacional:

1. **Login:** El encargado ingresa en el sistema introduciendo sus datos para validación
2. **Consulta libros:** Realiza una consulta de acuerdo a los datos brindados por el socio
3. **Resultados Consulta:** Como respuesta, se obtiene una lista eventualmente unitaria de libros que concuerdan con los criterios ingresados, especificando la disponibilidad de cada uno. El socio decide cual de los títulos disponibles es el que desea e informa al encargado (si el socio no le satisface ninguno de los títulos disponibles se retira y la operación termina). El encargado selecciona el libro de la lista resultante, ingresando en el formulario de préstamo
4. **Préstamo:** Por la forma en que se ingresa en este nodo, en el formulario de préstamo ya se tiene como dato el identificador del libro (ISBN). Luego, se selecciona el socio de una lista desplegable y lo mismo se hace con la cantidad de días del préstamo.

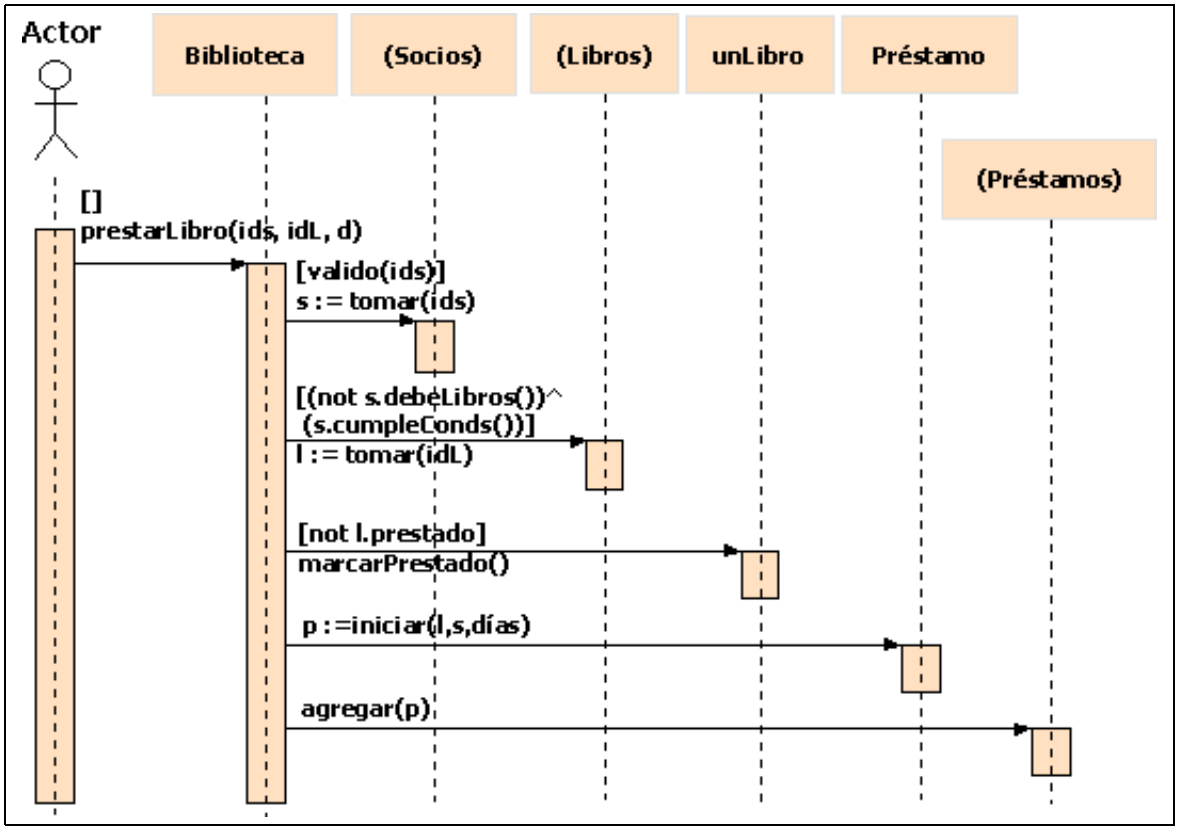

Biblioteca Login
login:
pass:
ok
Login incorrecto
volver
Menú Biblioteca
Consulta Libros
ISBN :
Autor :
Editor :
Estado:
*disponible
*prestado
*p.de extravío
*extraviado
Consulta
Consulta Socios
NºSocio :
Nom. y Ap. :
Tel. :
E-mail :
Estado :
*cumple
*debe libros
Consulta
Resultados consulta
1] .....
2] .....
3] .....
socio
libro
Socio
Libro
Prestar
Resultado préstamo
Préstamo
ISBN Libro:
Socio:
Días :
Cancelar
Prestar

5.2.4. Detalle de un préstamo

La clase **Biblioteca**, como se mostró en el modelo de objetos, contiene en su interface el método **prestarLibro(socio, libro, dias)**. Este mensaje es disparado al presionar un botón sobre la interface en el formulario de préstamo, en el nodo correspondiente del diseño de navegación. Este evento ejecuta código que chequea que se hayan ingresado todos los datos del formulario (selección del socio e ingreso de la fecha) y luego envía el mensaje *prestarLibro(...)* a la biblioteca. Para permitir esta comunicación entre el diseño de navegación y el modelo de objetos, se incluye en el script el código para importar la clase en un objeto en la memoria local del intérprete del lenguaje gdc. La forma más segura mediante PHP es integrar el ambiente de servlets para importar directamente las clases Java; utilizando la extensión para Java de PHP el código se vería similar a:

```
<?php
$isbnLibro = $HTTP_POST_VARS['lib']
$idSocio   = $HTTP_POST_VARS['soc']
$dias      = $HTTP_POST_VARS['d']
 ...
$bib = new Java('java.biblioteca.Biblioteca');
$bib.prestarLibro($isbnLibro, $idSocio, $dias);
?>
```

A partir de este mensaje se desprende la cadena de mensajes que se observa en el siguiente gráfico de tiempo UML:


Actor
Biblioteca
(Socios)
(Libros)
unLibro
Préstamo
(Préstamos)
[]
prestarLibro(ids, idL, d)
[valido(ids)]
s := tomar(ids)
[(not s.debeLibros())^
(s.cumpleConds())]
l := tomar(idL)
[not l.prestado]
marcarPrestado()
p :=iniciar(l,s,días)
agregar(p)

Capítulo 6

Conclusiones

"If this words are helpful for you,
then put them into practice.
If not, then there's no need for them."

Dalai Lama.

6.1. Introducción

En este capítulo se hace un balance sobre el conocimiento adquirido y sobre la utilidad y posibilidad de aplicación real de este "semi-formalismo". Las evaluaciones sobre objetivos alcanzados, son a criterio del autor, por lo cual no dejan de ser subjetivas. Invitamos al lector a que saque sus propias conclusiones, y escriba sus críticas y comentarios al respecto.

Se hace primero una evaluación sobre implementación de las reglas semánticas.

6.2. Consideraciones sobre implementación de prototipos

Utilizando la idea de [VM02] para implementación de un prototipo basado en el trabajo en desarrollo (working draft) [DFFM02] sobre la semántica formal del lenguaje para consulta sobre

documentos XML de la organización W3C WWW Consortium, XQuery 1.0, implementaremos una regla de la semántica operacional sugerida para PHP, de manera de sugerir cómo debería mapearse una especificación de este tipo a una implementación real para diseñadores e implementadores de lenguajes gcd. La regla más significativa para cualquier lenguaje gcd (en este caso PHP), es la de imprimir texto en la salida, por lo que elegimos ésta como modelo. La recordamos aquí:

$$\textit{OutputDir} \quad \frac{(e, Env, Store) \Longrightarrow (v :: T, Env', Store')}{([H, F, \trianglelefteq = \texttt{e} \trianglerighteq_{\texttt{PHP}}], Env, Store, Out) \rightsquigarrow_{PHP} ([H, F, \trianglelefteq \trianglerighteq_{\texttt{PHP}}], Env', Store', Out \oplus toString(v))}$$

A diferencia de [VM02] en la cual se utilizó el meta-ambiente ASF+SDF para mostrar cómo sería la implementación de un prototipo, la regla *OutputDir* se implementó en el lenguaje funcional *Hugs98* basado en *H*askell98 standard (para mayor información recurrir a `http://haskell.org/hugs`).

Se implementaron sólo las functiones, tipos de la semántica y del núcleo del lenguaje PHP necesarios y la función `output_rule` cuyo código se muestra a continuación:

```
output_rule :: Tuple -> Tuple

output_rule ((host, file, phpCode), env, str, out) =
let (v, env',str') = (evalCode phpCode env str)
in  ((host, file, emptyCode), env', str', out ++ phpTypeToString(v))
```

en donde la función `evalCode` corresponde a la evaluación de la expresión e en la regla, aunque en la función se evalúa toda la sección de código `phpCode`, en lugar de sólo la expresión que debe imprimirse. Esto es una decisión de implementación, por practicidad en el lenguaje Hugs: si quisiéramos hacer corresponder exactamente la implementación con la regla, deberíamos implementar, además de la función `evalCode`, una función que extrajera del pedazo de código `phpCode` la expresión e a imprimir. Eso sería más costoso.

Para ver como funciona la función `output_rule` tomamos una tupla como ejemplo:

```
(("sol.info.unlp.edu.ar","printSample.php",[PHPPrint "Hola Claudia!"]),
    [],[],"")
```

la cual es tipada por el intérprete hugs como:

```
::(([Char],[Char],[PHPSent]),
     [([Char],Int,SemType)],[(Int,SemType)],[Char])
```

Este tipo es equivalente por sinónimos de tipo (`type` de Haskell) a:

```
::((Host,ScriptFile,PHPCode),
     [(Id,Location,SemType)], [(Location,SemType)], Output)
```

y más reducidamente a:

```
::(Context, Environment, Store, Output)
```

según las definiciones de estos tipos en la implementación. Este tipo es exactamente igual al sugerido en el dominio semántico definido en el capítulo 2.

Ahora, si se aplica la función antes mencionada, se obtiene la tupla resultado:

```
(("sol.info.unlp.edu.ar","printSample.php",[PHPPrint "Hola Claudia!"]),
     [],[],"Hola Claudia!")
```

en donde el único efecto es haber concatenado el string que se deseaba imprimir en la última componente de la tupla.

La implementación de cualquiera de las otras reglas se hace en base a las operaciones definidas

en la semántica, que en lenguaje Hugs son fácilmente correspondidas por operaciones sobre listas.

Se deja como ejercicio para el lector la implementación de estas operaciones para permitir implementar cualquier otra regla de la semántica de PHP o ASP, sobre los tipos ya definidos en la implementación de esta regla: `PHPCode, PHPSent, PHPType, SemType, Environment, Store, Output`, etc.

6.3. Balance positivo: objetivos alcanzados

- Se ha presentado de manera tan amena como fué posible un cálculo (o dominio semántico) con una notación "quasi-formal" para describir el comportamiento de intérpretes para lenguajes gcd
- Se han definido dos relaciones de reducción para transformar las tuplas del dominio semántico
- Se ha mostrado el funcionamiento de una máquina virtual básica frente a construcciones sintácticas de los lenguajes ASP y PHP, lo que permite comparar y estudiar aspectos de estos lenguajes. Las reglas que definen la semántica resultaron de fácil lectura para cualquier persona con conocimientos básicos de álgebra
- Se ha sugerido como implementadores de intérpretes de lenguajes gcd podrían utilizar esta formalismo como especificación de la semántica de sus instrucciones y cómo el equipo de documentación podría utilizar la notación propuesta para adjuntar a una descripción en lenguaje natural de las instrucciones del lenguaje
- Se ha mostrado cómo podrían organizarse los scripts y cómo relacionar los lenguajes gcd con diferentes tecnologías (bases de datos y lenguajes orientados a objetos entre otras), para modelar eficientemente aplicaciones comerciales, dando un marco de trabajo para aplicaciones sobre internet
- Se ha avanzado un pequeño paso hacia la implementación real de una de las reglas semánticas, lo que puede originar ideas más realistas respecto de la utilización de este dominio semántico para lenguajes gcd

6.4. Balances no positivos u objetivos pendientes

- **Aplicación real** Se había mencionado como algunos de los objetivos que el formalismo podía ser utilizado como herramienta conceptual niveladora para ambientes de desarrollo. La experiencia con alumnos ingresantes (1º o 2º año de la carrera de informática) fue absolutamente negativa, a causa de su falta de interés.

  Como respuesta a este balance negativo, se propone como herramienta para desarrollo en grupos más avanzados en carreras de ciencias de la computación (por ejemplo, analistas de computación o alumnos de licenciatura avanzados). La idea es que los integrantes del grupo podrán decidir con uniformidad de criterios en base al conjunto de reglas de un determinado lenguaje qué estructuras del son convenientes en lo que respecta a eficiencia en la utilización de memoria (espacio) y complejidad de los algoritmos (tiempo), como también practicidad y claridad (legibilidad del código) y uniformidad de estilos.

- **Respecto del lenguaje JSP** Se determinó que este formalismo no es tan claramente aplicable al estudio uno de los lenguajes considerados en un principio, JSP, por ser éste básicamente orientado al uso de objetos e integración con tecnologías Java (Servlets, Tags JSP, JavaBeans, EnterpriseJavaBeans, etc). En este caso, el lenguaje base en que se escriben los scripts de generación dinámica es Java, el cual es obviamente mucho más sofisticado que los lenguajes VBScript y JScript utilizados por ASP o PHP. Otra razón que lo hace inevitablemente distinto es que la interpretación de los mismos *se hace una sola vez para generar los bytecodes* que serán utilizados por la máquina virtual de Java, a diferencia de los scripts ASP y PHP que *se interpretan cada vez que se requiere generar salida*.

  Por esto, la conformación de las tuplas sería inadecuada o al menos necesitaría ser reestructurada para contemplar la generación de bytecodes para un posterior procesamiento. Esta reestructuración se propone como trabajo futuro pendiente.

6.5. Trabajo futuro

- Reestructuración de esta semántica para el estudio del lenguaje JSP, como se menciona más arriba.

- Estudio de las propiedades de las relaciones de reducción sobre tuplas $\rightsquigarrow_{ASP}$ y $\rightsquigarrow_{PHP}$.
- Estudio de qué propiedades matemáticas pueden demostrarse sobre programas en torno a las definiciones semánticas, definiendo las reglas en forma más precisa y, consecuentemente, menos legible. Esto cambiría el enfoque de los objetivos del trabajo, aunque no deja de ser una rama de exploración futura con mayores intereses académicos.
- Mayor riqueza en la definición de los tipos de la semántica, con el fin de orientar el trabajo al estudio de teorías de tipos.

Agradecimientos

A **Claudia Pons**. Como diría mi amigo Tomás: “Su saber y su virtud están por encima de mis elogios”. Ella me comprende aunque yo la confunda con mis palabras.

A **María Laura Ponisio**, por estar cerca en tiempos difíciles. Para ella una frase de Blas Pascal que siempre decía mi abuela: “El corazón tiene razones que la razón no conoce”.

Y en último lugar a los más importantes: **mis padres**, por darme esta pequeña oportunidad, la vida.

Universidad Nacional de La Plata, Buenos Aires, Argentina.

7 de diciembre de 2001 - 16 de octubre de 2002.

Bibliografía

[AC96] Martin Abadi* and Luca Cardelli**, *A theory of objects*, Addison-Wesley Publishing Company, 1996, *Bell Labs. **Microsoft Corporation.

[And91] Gregory R. Andrews, *Concurrent programming, principles and practice*, Addison-Wesley Publishing Company, Menlo Park, CA 94025, 1991, The University of Arizona, USA.

[CF97] Adriana Compagnoni* and Maribel Fernández**, *An object calculus with algebraic rewriting*,
*University of Edinburg, Department of Computer Science, Edinburg, U.K.
**École Normale Supérieure, Paris, France, 1997.

[Cor99] Microsoft Corporation, *Microsoft internet information services 5.0 documentation*, (from on-line IIS version 5.0 documentation), 1997-1999.

[DFFM02] Denise Draper, Peter Fankhauser, Mary Fernadez, and Ashok Malhotra, *Xquery 1.0 formal semantics*, Working draft, W3C World Wide Web Consortium, march 2002.

[Gog98] Healfdene Goguen, *A note on typed operatinal semantics*,
Department of Computer Science, University of Edinburgh, U.K., 1997-1998.

[Hen90] Matthew Hennessy, *The semantics of programming languages*, John Wiley & Sons Ltd., New York, USA, 1990, University of Sussex, UK.

[SREL99] Daniel Schwabe*, Gustavo Rossi**, Luiselena Esmeraldo*, and Fernando Lyardet**, *Engineering web applications for reuse*,
*Departamento de Informática, PUC-Rio, Brazil
**LIFIA, Facultad de Informática, UNLP, La Plata, Argentina, 1999.

[Val02] Abel Valente, *Framework para persistencia de objetos en c++ para bases relacionales*, UNLP, La Plata, Argentina, 2002.

[VM02] Sandra S. Venske* and Martin A. Musicante**, *Uso de semántica operacional na prototipaçao de uma linguagem de consulta para xml*,
*Universidade Estadual do Centro-Oeste, Dept. de Análise de Sistemas, Guarapuava, PR, BRASIL, Cx. Postal 730, 85015-430 y Universidade Federal do Paraná, Dept. de Informática, Curitiba, PR, BRASIL, Cx. Postal 19081, 81.531-970.
Email: sandra@inf.ufpr.br
**Universidade Federal do Paraná, Dept. de Informática, Curitiba, PR, BRASIL, Cx. Postal 19081, 81.531-970.
Email: mam@inf.ufpr.br, 2002.