PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA

SECCIÓN DE INGENIERÍA INFORMÁTICA

INTÉRPRETE DE PÁGINAS WEB DINÁMICAS PARA EL SERVIDOR APACHE

Para optar al título de Ingeniero Informático

Presentado por

Martin Richard Kong Moreno

Pando – Lima

Diciembre - 2004

Le dedico este trabajo a mi padre y madre, quienes me apoyaron a lo largo de mis estudios. También le agradezco al profesor Viktor Khlebnikov por la ayuda brindada en el trabajo realizado.

RESUMEN

El presente trabajo tiene como objetivo explicar cómo funciona la tecnología Server-Side Scripting en la generación de páginas web dinámicas, desarrollando dos versiones de un intérprete basado en las reglas del lenguaje Pascal. La primera de estas versiones corre como un módulo DSO (Dynamic Shared Object) integrada al servidor Apache, mientras que la segunda es un intérprete independiente que genera las páginas dinámicamente interactuando con el servidor Apache vía CGI.

El primer capítulo se concentra en introducir el marco contextual del proyecto, junto con una breve explicación de lo que hace y la presentación de algunas de las tecnologías predominantes en el mercado.

El segundo capítulo trata del Apache Software Foundation (ASF), su importancia como organización para la expansión del software libre y de tal vez su contribución más importante, el Servidor HTTP Apache, una pieza esencial en el trabajo.

El tercer capítulo presenta el intérprete desarrollado, empezando por los fundamentos teóricos, continuando con su diseño y terminando con algunos aspectos muy particulares de la implementación hecha.

En el capítulo cuatro se realiza una breve explicación del entorno en el cual se desarrolló el proyecto, abarcando los tres programas más usados a lo largo de la implementación: el compilador de compiladores (Bison y Yacc), las librerías APR y una herramienta bastante potente: el APXS (APache eXtenSion tool), usada para compilar e instalar el módulo.

Para concluir, en el capítulo cinco se da una breve excursión por la estructura básica de un módulo DSO del Apache Web Server, explicando la responsabilidad de cada fase y resaltando algunos aspectos del módulo junto con las consideraciones adicionales para integrarlo al intérprete.

INDICE

INTRODUCCIÓN

En la presente tesis se desarrolla un intérprete del lenguaje de programación Pascal cuyo fin es traducir páginas web que contengan código embebido de la forma:

<~ *Instrucciones del lenguaje a interpretar* ~>

De este modo, el intérprete se desempeñará de manera análoga a las tecnologías PHP, ASP, Python y JSP, entre otros, todos éstos servidores de páginas web dinámicas.

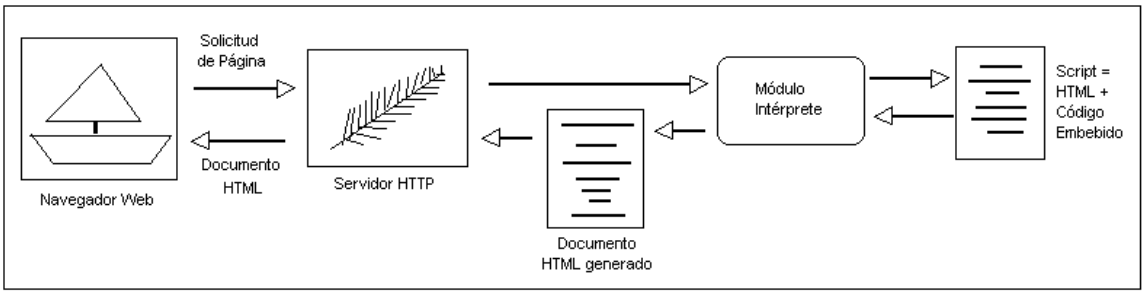
Se han implementado dos versiones del programa intérprete:

1. Como módulo integrado al servidor Apache trabajando sobre el sistema operativo GNU Linux.
2. Como programa externo independiente (*standalone*) que puede ejecutarse mediante llamadas indirectas por el servidor HTTP cuando en la cabecera del *Script* se encuentre:

   #!*dirección del programa interprete*

   En cuyo caso interpreta el resto del archivo para enviar la página web de respuesta.

Además de esto, el programa externo puede usarse en el modo de terminal o consola con fines de depuración.



Interacción entre el navegador web, servidor HTTP y módulo intérprete

El lenguaje de implementación utilizado ha sido por necesidad el Lenguaje C Estándar, debido a que la librería del Apache esta íntegramente desarrollada en dicho lenguaje y también por la facilidad de compilar el programa en las plataformas de Windows y GNU Linux. Además de esto, las herramientas generadores de compiladores que se utilizan, como Bison y Yacc, producen como código final programas fuentes en C.

Hoy en día existen una gran variedad de Lenguajes de Programación de *Scripting*. El término "*Script*" se refiere a un programa de computadora capaz automatizar trabajos, que en otros casos, realizarían los usuarios en forma interactiva. Por ejemplo, un Shell Script consiste primordialmente de comandos que serían ingresados por una terminal o consola. Otro ejemplo se da en los editores de texto, donde algún usuario podría escribir un *Script* con una secuencia de comandos de edición comúnmente usados y así evitar el trabajo de llamar cada comando por separado.

Aquellos lenguajes usados fundamentalmente para escribir *scripts* se llaman *Scripting Languages* (Lenguajes de Scripting). Muchos de estos lenguajes son bastante sofisticados y han sido utilizados para escribir programas altamente elaborados, los cuales siguen llamándose *scripts* aún cuando van más allá de automatizar una simple secuencia de comandos de usuario.

Server-Side Scripting (SSS) es una tecnología de servidores web en la cual una solicitud de usuario es resuelta ejecutando un *Script* directamente en el servidor web para generar páginas HTML dinámicas. Esta tecnología es generalmente usada para proveer sitios web interactivos que trabajan con bases de datos o con algún otro tipo de repositorio de datos. La diferencia entre SSS y *Client-Side Scripting* radica en que este último ejecuta los scripts sobre el navegador web del cliente, usualmente en JavaScript. La principal ventaja del SSS es su habilidad para adaptar la respuesta al usuario basándose en sus requerimientos, datos ingresados y/o derechos de acceso.

En los inicios de la web esto era posible gracias a una combinación de programas escritos en C, *scripts* hechos en Perl e incluso Shell Scripts, todos éstos usando la tecnología CGI (Common Gateway Interface). Estos scripts en algunos casos son ejecutados por el sistema operativo, accediendo a las variables de entorno para obtener los datos ingresados por el usuario a través de formularios HTML y devolviendo los resultados por salida estándar.

Hoy en día existen Lenguajes de Scripting totalmente diseñados y orientados al entorno web. Ejemplo de éstos son ASP (ASP es en realidad el nombre de la tecnología y los lenguajes que soporta son VBScript y JavaScript) y PHP, los cuáles son ejecutados por el servidor web o por módulos de extensión del mismo. Cualquier forma de *Scripting* puede ser empleada para construir sitios web complejos de páginas múltiples, sin embargo, la ejecución directa (caso de ASP y PHP) resulta en un costo menor debido a la inexistencia de llamadas a intérpretes externos.

La tesis presentada constituye en la opinión del autor un esfuerzo para contribuir en alguna medida en el entendimiento y desarrollo teórico-práctico de cómo funcionan e interactúan los navegadores de internet, los servidores del protocolo HTTP y los programas generadores de páginas web dinámicas, que se usan en las comunicaciones de todo tipo de servicio en la actualidad.

EL APACHE SOFTWARE FOUNDATION (ASF)

La Fundación de Software Apache, es una organización que provee soporte legal y financiero a una gran gama de proyectos de software de código abierto. La Fundación también aporta con un marco de trabajo establecido que apoya la propiedad intelectual al igual que las contribuciones financieras y que limita simultáneamente la potencial exposición legal de los contribuyentes, evitando que éstos sean objeto de demandas legales dirigidas a los proyectos de la Fundación. Mediante un proceso de desarrollo meritocrático y colaborador, los proyectos Apache ofrecen productos de software de grado empresarial y sin costo de adquisición, que atraen a extensas comunidades de usuarios. La licencia pragmática del Apache permite a todos los usuarios, individuales y comerciales, a desplegar productos Apache.

La Fundación ha sido incorporada como una corporación basada en socios, sin fines de lucro, para garantizar la continuidad de sus proyectos más allá de la participación de sus voluntarios individuales.

Historia

La Fundación fue creada en 1999 por el anteriormente conocido “Grupo Apache”, un grupo de personas que continuaron dándole soporte y mantenimiento al Servidor Web HTTPD escrito por la NCSA (National Center for Supercomputing Applications), de la Universidad de Illinois en Urbana-Champaign.

El servidor era de libre adquisición, venía con el código fuente y contaba con una licencia que permitía su modificación y redistribución. Sin embargo, los desarrolladores originales ya no estaban presentes y habían dejado a los usuarios sin soporte.

Algunos de esos usuarios empezaron a intercambiar reparaciones de código (parches o patches en inglés) e información relacionada con cómo prevenir problemas y mejorar el software. Brian Behlendorf creó una lista de correo para aquellos usuarios que desearan colaborar con las reparaciones, mantenimientos y mejoras de este software.

El nombre “Apache” fue escogido en honor a la tribu India de americanos nativos de los Apache, bien conocidos por sus habilidades en estrategias de guerra y su gran resistencia. Por otro lado, hace mofa de su nombre: “A patchy Web Server” (“A patchy” suena como “Apache” pero también significa “Parchado”), un Servidor Web parchado. Sin embargo, este no fue su origen. El grupo de desarrolladores que lanzaron este nuevo software pronto empezaron a llamarse “El Grupo Apache”.

Entre 1995 y 1999, el Servidor Web Apache HTTPD creado por el Grupo Apache se convirtió en el líder del mercado y continúa siéndolo con más del 65% de los sitios web utilizándolo (Estadística obtenida de http://www.apache.org/foundation/how-it-works.html).

Con el crecimiento de la web, el interés económico también aumentó, y el sitio web del Apache albergó a más proyectos hermanos, tales como *mod_perl*, *PHP*, *Java Apache*. La necesidad de una organización coherente y mejor estructurada que protegiera a los individuos de potenciales ataques legales se hizo cada vez más clara.

El Servidor Apache

Es una implementación robusta, usable a nivel comercial y de código fuente abierto, del Servidor HTTP para sistemas operativos modernos tales como GNU Linux y Windows, y que busca proveer los servicios HTTP en forma segura, eficiente y extensible pero siempre de acuerdo a los estándares HTTP en vigencia.

También llamado Servidor HTTP. Es un programa servidor que se encarga de recibir solicitudes de navegadores, procesarlas y enviar las páginas de respuesta de vuelta al navegador.

La comunicación entre el cliente (navegador) y el servidor HTTP se lleva a cabo gracias al protocolo HTTP (HiperText Transfer Protocol ó Protocolo de Transporte de Hipertexto).

Por ejemplo, el servidor recibe una solicitud así:

GET /mipagina.html HTTP/1.1.

Donde el primer campo corresponde al método de transferencia de datos del formulario: GET, POST, etc. El segundo, el nombre del archivo solicitado (pudiendo incluir la ruta de directorios) y finalmente, el tercer campo, la versión del protocolo HTTP. Luego, teniendo estos datos el servidor Web envía la respuesta que consiste en tres partes:

- El estado de la respuesta. Por ejemplo: http / 1.0 200 OK .
- La cabecera de la respuesta conteniendo la fecha, el tipo de servidor, tipo de contenido, etc.
- El documento HTML solicitado.

Operación del Servidor Apache

Configuración

El servidor Apache opera de acuerdo a uno o más archivos de configuración. Generalmente, al principal de éstos se le llama “httpd.conf”. En estos archivos se especifican diversas características de funcionamiento, como el número de procesos servidores, máximo de clientes que podrán atenderse simultáneamente, el puerto dónde se recibirán las solicitudes, el directorio raíz, etc.

Estas características de operación se establecen mediante órdenes o comandos llamados Directivas. Aquellas directivas situadas en el archivo de configuración principal son aplicables a todo el servidor. Sin embargo, para limitar su alcance o para aplicar configuraciones particulares solamente a ciertas partes del servidor, se dispone de secciones especiales dentro del archivo de configuración. Algunas de estas secciones son: <Directory> (para directorios), <Files> (para archivos) y <Location> (para URLs).

Handlers

Un "handler" es una representación interna del Apache de una acción a ejecutar cuando hay una llamada a un documento. Generalmente, las páginas tienen *Handlers* implícitos, basados en el tipo de documento del que se trata. Normalmente, todos los archivos son simplemente enviados por el servidor tal cuál están, pero algunos tipos de páginas se tratan de forma diferente.

Apache ofrece a uno la posibilidad de definir *Handlers* explícitamente. Bien basándose en la extensión del archivo en la ubicación que tenga éste, se pueden especificar *Handlers* sin tener en cuenta el tipo de documento del cual se trate. Esto es una ventaja por dos razones. Primero, es una solución más elegante. Segundo, porque a un documento se le puede asignar tanto un tipo como un *handler*.

También se disponen de opciones para agregar *Handlers*. Una alternativa es compilarlos inicialmente con el servidor. Otra es incluirlos en un módulo DSO y luego cargar el módulo con la directiva "*LoadModule*".

Tipos MIME

MIME (Multipurpose Internet Mail Extensions) es un estándar para clasificar y describir los tipos de información (media) existentes. Fue creado originalmente con el fin de especificar distintos tipos de codificaciones a texto. Esto hace posible el envío de correos electrónicos.

La clasificación MIME se divide en dos partes: tipos y subtipos. Los principales tipos son: text, multipart, message, application, image, audio, video. Los subtipos dependen de cada tipo. Ejemplos de éstos son: text/text, text/html e image/jpg, image/gif.

Atención de solicitudes

El proceso de atención de solicitudes consta de varias fases:

- Fase de análisis: se encarga de mapear el URL requerido y asociarlo a un documento en el sistema de archivos local.
- Fase de seguridad: se procede a la autenticación de usuarios usando la lista de acceso del servidor, es decir, los usuarios autorizados.
- Fase de preparación: donde se confirma el tipo MIME del archivo solicitado y se realizan revisiones y ajustes adicionales previos a la atención final de la solicitud.
- Fase del handler: cuya finalidad es enviar el contenido del archivo al navegador que hizo la solicitud. Si el contenido del archivo se sirve tal cual está, entonces se trata de una página estática. Por otro lado, en esta fase también es posible modificar e incluso generar nuevo contenido de respuesta. En este último caso ya estamos hablando de páginas web dinámicas.

Extensión de la funcionalidad

El Servidor Apache posee una arquitectura modular, es decir, está compuesto de varios módulos. Algunos de ellos se incluyen en la distribución por defecto, otros hay que agregarlos manualmente ya sea por recompilación del servidor (estáticamente) o dinámicamente, mediante el mecanismo de DSOs (Dynamic Shared Objects u Objetos Dinámicos Compartidos).

Uno de los módulos más importantes es “mod_so”, responsable de dar soporte al mecanismo de DSO. Desafortunadamente, este módulo es uno de los dos módulos que no se pueden cargar dinámicamente y que tienen que ser agregados estáticamente compilándolo junto con el servidor Apache.

El uso de DSO permite extender fácilmente la funcionalidad del paquete servidor con módulos hechos por terceros, inclusive después de haber instalado el servidor.

También hay algunas desventajas adyacentes:

- No todos los sistemas operativos soportan la carga dinámica de código en el espacio de direcciones del programa.
- El servidor se vuelve aproximadamente 20% más lento al iniciar o reiniciar.
- El servidor es aproximadamente 5% más lento al atender las solicitudes.
- No todas las plataformas soportan enlaces entre librerías DSO.

El módulo

El módulo desarrollado en la tesis aprovecha el mecanismo de DSO para agregar una funcionalidad al Apache. El módulo intérprete es responsable de procesar código embebido del lenguaje de programación diseñado. Así, en un mismo documento, se puede tener secciones de código HTML mezclados con porciones de código de nuestro lenguaje de programación.

El módulo intérprete funciona de la siguiente manera:

- Primero, el servidor recibe la solicitud de una página web.
- Luego, el módulo analiza el URI recibido y lo transforma al nombre de archivo.
- Se determina la ubicación del archivo requerido (mapeo al sistema de archivos local).

- Se comprueba si el archivo solicitado corresponde con el *handler* implementado por el módulo.
- Si coincide entonces se procede a generar y enviar el contenido (generación de la página dinámica). Si no coincide se niega el servicio y entran a actuar los *Handlers* predeterminados del servidor Apache.

La generación de contenido se hace en la fase del handler, como se explicó anteriormente. Este *handler* sólo debe procesar un cierto tipo de archivos, basándose en la extensión del nombre. Para ello, modificamos el archivo de configuración principal del servidor Apache y usamos dos directivas:

- La directiva “AddType”, la cual sirve para agregar una nueva extensión de archivos y al mismo tiempo, asociarlo a un tipo MIME. En este caso, “text / html “.
- La directiva “AddHandler”, para adicionar el handler creado en el módulo y asociarlo a la nueva extensión de archivos creada con “AddType”. Este handler debe ser el mismo que se implementa en el módulo intérprete.

El *handler* en cuestión distingue el código HTML de las secciones con código embebido de nuestro lenguaje de programación. Entonces, se pueden tomar dos cursos de acción:

- Si es un bloque de código HTML, se envía el contenido tal cual aparece en el archivo solicitado.
- Si es una porción de código embebido, se invoca al analizador sintáctico (función que determina si las instrucciones del código embebido cumplen con las reglas de sintaxis establecidas por el lenguaje).

Finalmente, si no hay errores, se continúa con la etapa de interpretación (ejecución del código intermedio generado por el analizador sintáctico), generando la salida que se enviará como respuesta a la solicitud hecha por el navegador web.

EL INTÉRPRETE

Intérpretes y Compiladores

El intérprete es un programa traductor que además ejecuta el programa traducido. La diferencia entre un compilador puro y un intérprete consiste en que el primero de éstos no ejecuta el programa que recibe como entrada, el código fuente, en lugar de ello lo traduce a código máquina ejecutable (generalmente llamada código objeto), el cual es guardado en un archivo para su posterior ejecución. Es posible ejecutar el mismo código fuente ya sea directamente en un intérprete o compilando y luego ejecutando el código máquina obtenido.

En términos de eficiencia, resulta más lento interpretar un programa que correr una versión compilada del mismo. Sin embargo, la ejecución por medio del intérprete alcanza tiempos menores que el tiempo total requerido para compilar y ejecutar el mismo programa. Los intérpretes generan código intermedio y generalmente no lo almacenan. Un caso especial es el de Java, un híbrido que trabaja con código intermedio (aunque ellos lo denominan “código de bytes”), lo guardan en archivos con extensión “.class” para luego interpretarlos con su máquina virtual.

La interpretación de código deriva en tiempos mayores debido al costo inherente que implica analizar cada instrucción y luego realizar la acción deseada. La ejecución de código compilado no tiene este costo adicional porque solamente se es responsable de llevar acabo la acción. A este tipo de análisis en tiempo de corrida se le denomina “costo de interpretación”. Por otro lado, el intérprete debe mapear repetidas veces los identificadores a las direcciones de almacenamiento durante la ejecución del programa y no al momento de la compilación.

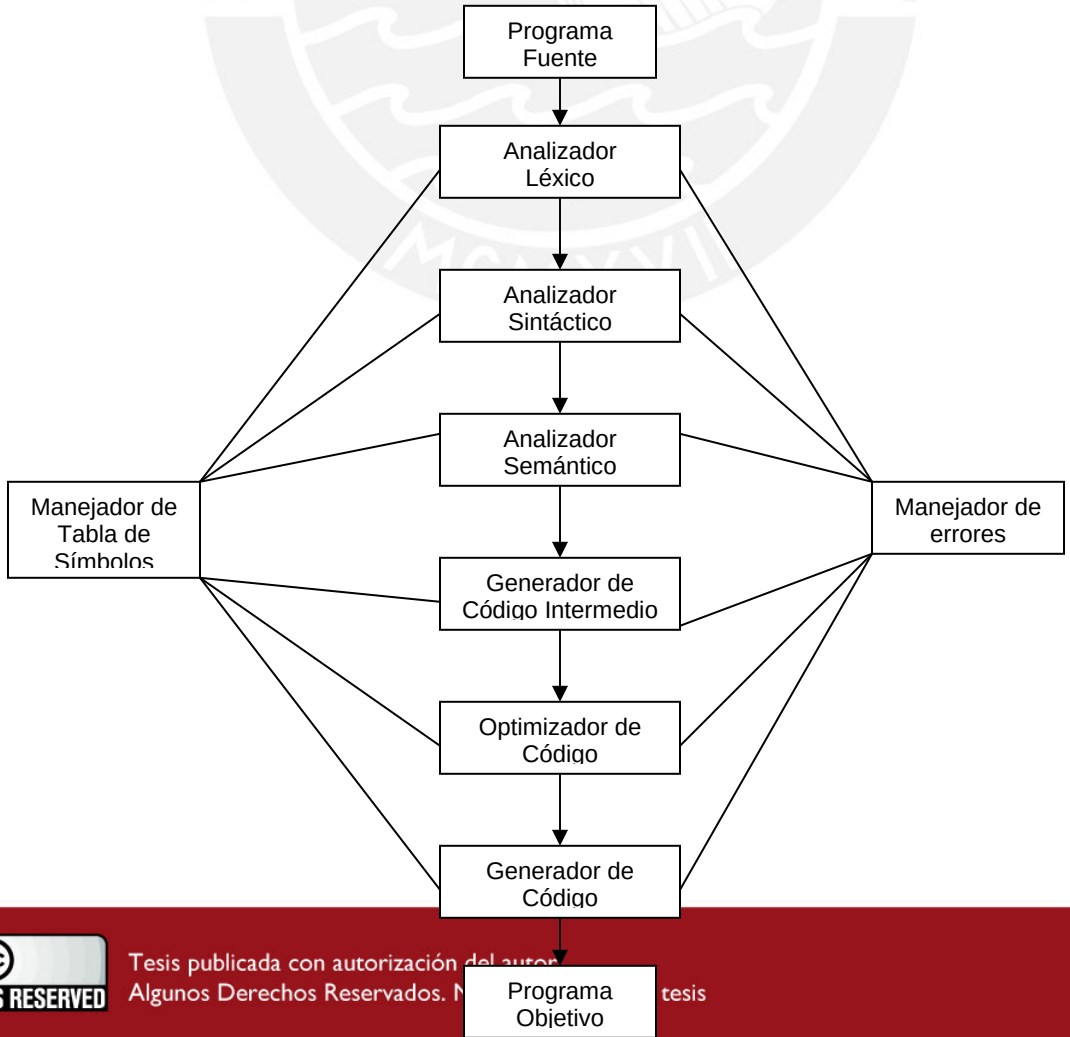
Estructura del Intérprete

Hasta ahora hemos indicado que la diferencia entre compiladores e intérpretes es que éstos últimos llegan a ejecutar el programa que reciben como entrada. No obstante, ambos comparten una misma estructura. Todo el proceso de traducir el código fuente y obtener código objeto se puede dividir en varias fases.

Una descomposición típica de un compilador consta de las siguientes fases:

- Analizador léxico
- Analizador sintáctico
- Analizador semántico
- Generador de código intermedio
- Optimizador de código
- Generador de código

A continuación se presenta el esquema mencionado:



En la práctica algunas de estas fases se pueden combinar y podría no existir un producto intermedio entre las fases agrupadas.

Las tres primeras fases realizan la función de análisis:

- Análisis léxico o Análisis Lineal: se procesa el flujo de caracteres que componen el programa fuente, de izquierda a derecha, agrupándolos en secuencias con significado llamados *tokens*.
- Análisis sintáctico o Análisis Jerárquico: también conocido como *parsing*. El objetivo de esta fase es agrupar los *tokens* del programa fuente en frases gramaticales que el compilador utiliza para generar la salida.
- Análisis semántico: se realizan revisiones buscando errores de significado, asegurando que los componentes del programa estén ordenados en forma coherente y recopilando información de tipos para la fase de generación de código.

Las siguientes tres fases llevan a cabo la función de síntesis, específicamente, de la generación de código final:

- Generación de código intermedio: algunos compiladores prosiguen con la generación explícita de una representación intermedia del programa fuente, la cual se puede considerar como un programa para una máquina abstracta. Esta representación puede tener una variedad de formas. Más adelante se explicará la estructura por la cual se ha optado para la implementación de este intérprete.
- Optimización de código intermedio: el objetivo de esta fase es intentar mejorar el código intermedio de la fase anterior, en busca de código máquina más rápido. La cantidad de tiempo que se dedica a esta etapa varía entre los compiladores, sin embargo, cabe resaltar que los compiladores optimizados destinan cantidades considerables de la compilación para este fin.

- Generación de código final: la última fase del compilador es el generar código final (o código objeto), el cual consiste por lo general en código máquina o *assembler*. Se asignan espacios de memoria para cada variable declarada en el programa. Luego, cada instrucción del código intermedio es traducida a su equivalente en instrucciones máquina.

Agrupación de fases

En la práctica es común agrupar las fases mencionadas en un *front end* y en *un back end*. El *front end* consiste en aquellas fases que dependen principalmente del lenguaje fuente y que son largamente independientes de la plataforma. Por ende, es usual incluir en el *front end* al analizador léxico, al analizador sintáctico, el manejador de la tabla de símbolos, el analizador semántico y la generación de código intermedio. También es posible realizar algunas optimizaciones a este nivel, sin embargo, aclaramos que no se realizarán optimizaciones en el intérprete. Por otro lado, se debe incluir el control de errores apropiado a cada fase.

El *back end* incluye las partes del compilador que dependen directamente de la plataforma. Generalmente, estas secciones son independientes del lenguaje fuente pero dependientes de la representación intermedia. Las fases que incluye el *back end*, usualmente, son la optimización del código intermedio, la generación de código final y el control de errores respectivo.

Algunos aspectos de la implementación

- Si bien uno de los objetivos de este trabajo es obtener un intérprete integrado al Servidor Web Apache como módulo DSO, también se ha producido un intérprete externo que cuenta con las mismas características que el primero, el cual puede usarse tanto para ejecutar páginas web de modo indirecto a través del servidor Apache o del Servidor Tomcat, como con fines de depuración.

- Como mencionamos anteriormente, es posible agrupar distintas fases de compilación. Aprovecharemos esta ventaja para generar el código intermedio y administrar la tabla de símbolos, a la misma vez que se realiza el análisis semántico.
- Ninguno de las dos intérpretes, ya sea el que trabajará como módulo DSO dentro del Servidor Apache ni el intérprete externo, presentarán optimización de código.
- Para obtener las salidas de los programas se ejecutará el código intermedio generado por el *front end* del compilador. La función intérprete del módulo DSO hará uso de las librerías APR (descritas más adelante), mientras que el intérprete externo usará las librerías de C.

Diseño del intérprete

Gramáticas de Contexto Libre (Context Free Grammars, CFG)

Es una notación utilizada para especificar la sintaxis de un lenguaje. Las gramáticas de contexto libre, o simplemente gramáticas, tienen la propiedad particular de poder describir la estructura jerárquica de las construcciones de varios lenguajes de programación.

Los CFGs se componen de cuatro elementos:

- Un conjunto de tokens o símbolos terminales.
- Un conjunto de símbolos no terminales.
- Un conjunto de producciones donde cada producción consiste de un símbolo no terminal, llamado el lado izquierdo de la producción, una flecha, y una secuencia de tokens y/o símbolos no terminales, llamada el lado derecho de la producción.
- La designación de uno de los símbolos no terminales como símbolo de inicio.

El uso de un CFG nos da la oportunidad de usar herramientas como Yacc y Bison, las cuales son generadores de compiladores que trabajan en base al algoritmo LALR. Se discutirá más acerca de este algoritmo en el capítulo 4.

En la siguiente sección se presentará la gramática diseñada para el intérprete.

```
<program> ::= <opList> <instrList>

<opList> ::= <opList>  <operation> ';'  | <operation> ';' | ε

<operation> ::= FUNCTION ID '(' <varBlock> ')' ':' <type> ';' <declareBlock>
                BEGIN     <instrList> END  | PROCEDURE ID '(' <varBlock> ')' ';'
                <declareBlock> BEGIN <instrList> END

<varBlock> ::= <varBlock>  ';' VAR <varType> | VAR <varType>      | ε

<declareBlock> ::= <varBlock> ';'   | ε

<varType> ::= <declarList> ':' <type>

<declarList>::= <declarList> ',' ID  | ID

<type>          ::= ARRAY '[' <dimDef> ']' OF <type> | TYPE_INTEGER  |
                TYPE_REAL   | TYPE_BOOLEAN  | TYPE_STRING  |
                TYPE_CHAR

<dimDef> ::= <dimDef> ',' <lim>   | <lim>

<lim> ::= TYPE_INTEGER  | <indep>

<instrBlock>::= BEGIN <instrList> END
```

<instrList> ::= <instrList> <instr> ';' | <instr> ';'

<instrGroup>::= <instrBlock> | <instr>

<instr> ::= <instrAsig> | <instrIf> | <instrLoop> | <instrPrint> |
<procedure> | <instrDef> | <instrCase> | <inc> | <dec>

<instrDef>::= VAR <varType> | CONST ID EQUAL <literal>

<literal> ::= TYPE_INTEGER | TYPE_REAL | TYPE_STRING

<procedure> ::= ID '(' <varList> ')'

<instrIf> ::= IF <expr> THEN <instrGroup> <instrElse>

<instrElse> ::= ELSE <instrGroup> | ε

<instrLoop> ::= WHILE <expr> DO <instrGroup>
| REPEAT <instrList> UNTIL <expr>
| FOR <instrAsig> <direction> <expr> DO <instrGroup>

<direction> ::= TO | DOWNTO

<instrPrint>::= PRINT ‘(' <exprList> ')' | PRINTLN '(' <exprList> ')'

<exprList> ::= <exprList> ',' <expr> | <expr>

<varList> ::= <varList> ',' <expr> | <expr>

<instrCase> ::= CASE <exp.> OF <optionList> END

<optionList> ::= <optionList> <option> ‘;’ | <option> ‘;’

<option> ::= <candidateList> ‘:’ <instrGroup>

<candidateList> ::= <candidateList> ‘,’ <candidate> | <candidate>

<candidate> ::= TYPE_INTEGER | ID

<inc> ::= INC '(' ID ')'

<dec> ::= DEC '(' ID ')'

<expr> ::= <expr> '+' <term> | <expr> '-' <term> | <expr> OR <term>
| <expr> <comparator> <expr> | <expr> IN <range> | <term>

<range> ::= <expr> '.' '.' <expr>

<comparator> ::= EQUAL | SMALLER | SMALLERE | GREATER |
GREATERE | DISTINCT

<term> ::= <term> '*' <factor> | <term> '/' <factor> | <term> AND <factor> |
<factor>

<instrAsig> ::= ID ASSIGN <expr> | ID '[' <dims> ']' ASSIGN <expr>

<dims> ::= <dims> ',' <expr> | <expr>

<factor> ::= '(' <expr> ')' | '-' <factor> | ID '[' <dims> ']' | NOT <expr>
| PZKTRUE | PZKFALSE | <indep> | <literal>

<indep> ::= <function>| ID <function> ::= ID '(' <varList> ')'

Código Intermedio

En el modelo de análisis y síntesis de un compilador, el *front end* se encarga de traducir el código fuente del programa en una representación intermedia a partir del cual el *back end* pueda construir código final. Aquellos aspectos relacionados con la generación de código final se tratan de delegar al *back end* tanto como sea posible. Por otro lado, el uso de una representación intermedia trae consigo ciertas ventajas:

- Se facilita la generación de código final para distintas plataformas. Esto se logra cambiando el back end y manteniendo el mismo front end.
- Se pueden aplicar optimizaciones de código intermedio independientes de la plataforma de destino.

En el resto de la presente sección describiremos la estructura utilizada para la representación intermedia.

El código intermedio es almacenado en una tabla llamada Tabla de código. Cada entrada en esta tabla representa una instrucción en la representación intermedia. Nuestra representación estará basada en cuartetos, donde cada cuarteto se conforma de un operador (código de operación) y tres operandos.

En el siguiente contexto se entiende por “Operación binaria” a la operación que necesite de dos operandos para poder realizarse, mientras que “Operación unaria” es aquella que necesita de un operando. Las operaciones implementadas a través de este código intermedio se pueden clasificar en 7 grupos:

- Operaciones binarias con resultado: Las operaciones de este tipo almacenarán el resultado en el primer operando. La operación en sí será aplicada al segundo y tercer operando. Las operaciones implementadas de este tipo son:

BOR	OR booleano.
BAND	AND booleano.
EQUAL	Operación relacional “igual a”.
SMALLER	Operación relacional “menor que”.
SMALLERE	Operación relacional “menor o igual que”.
GREATER	Operación relacional “mayor que”.
GREATERE	Operación relacional “mayor o igual que”.
DISTINCT	Operación relacional “distinto que”.
‘+’	Operación aritmética de adición.
‘-‘	Operación aritmética de substracción.
‘*’	Operación aritmética de multiplicación.
‘/’	Operación aritmética de división.

- Operaciones binarias sin resultado: Son aquellas que necesitan de dos operandos para poder realizarse, a pesar de no contar con un resultado en concreto. Por lo general se usarán el primer y tercer operando. Operaciones implementadas:

JMPT	Operación “saltar si verdadero”. Si el operando 1 es cierto se salta a la dirección indicada por el operando 3.
JMPF	Operación “saltar si falso”. Si el operando 1 es falso se salta a la dirección indicada por el operando 3.
MOV	Operación de asignación. Copia el valor del operando 3 al operando 1.

- Operaciones unarias con resultado: La operación se aplica al operando 3 y el resultado se guarda en el operando 1. Operaciones implementadas:

UMIN	Operación “menos unario”.
BNOT	Operación “not booleano”.

- Operaciones unarias sin resultado: La operación se aplica al único operando válido, el cual es en ocasiones el operando 1 y en otras el operando 3. Operaciones implementadas:

JMP	Salto incondicional a la dirección indicada por el operando 3.
INC	Incremento unitario del primer operando.
DEC	Decremento unitario del primer operando.

- Operaciones con múltiples operandos: Son operaciones que aceptan un número variable de argumentos. Como regla general, los operandos de estas operaciones estarán almacenados en las entradas inmediatamente anteriores de la tabla de código. Operaciones implementadas:

PRT	Escritura con número variable de operandos. Sus operandos
PRTLN	Escritura con número variable de operandos. Incluye cambio de línea.
CALLF	Realiza la llamada a una función. Guarda la información pertinente en un registro de activación y lo empuja dentro de la pila de registros de activación. Véase la sección “Registros de Activación” para los detalles de la implementación.
CALLP	Realiza la llamada a un procedimiento. Guarda la información pertinente en un registro de activación y lo empuja dentro de la pila de registros de activación. Véase la sección “Registros de Activación” para los detalles de la implementación.
RARR	Lectura de una celda de arreglo y asignación de su

	valor a una variable temporal. Véase la sección de algoritmos involucrados para los detalles de la implementación.
WARR	Escritura en una celda de arreglo. Véase la sección de algoritmos involucrados para los detalles de la implementación.
MATCH	Ejecuta la acción de la sentencia CASE. Véase la sección de algoritmos involucrados para los detalles de la implementación.
IN	Realiza una revisión de rangos aplicado al operando 2. El resultado es almacenado en el primer operando. El tercer operando es el número de instrucción (entrada en la Tabla de Código) que contiene los límites del rango (código de operación RANGE).

- Operaciones sin operandos:

RETF	Operación de retorno de función. Se hace un “pop” a la pila de registros de activación.
RETP	Operación de retorno de procedimiento. Se hace un “pop” a la pila de registros de activación.

- Otros: incluye principalmente a los operandos de las operaciones con número variable de argumentos.

PSHP	Guarda un operando.
CAND	Guarda un candidato a coincidencia con la llave del MATCH.
DIM	Almacena una dimensión de arreglo.
DJMP	Usado para definir las direcciones de salto para cada opción de la sentencia CASE (implementada con la operación MATCH). Véase la sección de algoritmos

	involucrados para los detalles de la implementación.
RANGE	Almacena los límites inferior y superior de un rango en los operandos 1 y 3, respectivamente.

Tabla de Símbolos

Un compilador utiliza las tablas de símbolos para llevar cuenta del alcance y asociando información a los nombres. Cada vez que se encuentra un nombre se realiza una búsqueda en la tabla de símbolos. Mientras tanto, la tabla sólo se modifica si se encuentra un nuevo nombre (en tal caso se agrega un nuevo registro o símbolo a la tabla) o si se descubre nueva información acerca de algún símbolo (nombre).

Un buen mecanismo de tabla de símbolos debe permitir la adición de nuevas entradas, al igual que la búsqueda eficiente de algún nombre. Para esto se dispone de varias estrategias:

- Listas enlazadas con inserción ordenada y búsqueda lineal.
- Uso de arreglos: inserción, ordenación por *quicksort* y búsqueda binaria.
- Uso de tablas de *hash*.

El enfoque que tomaremos es el de las tablas de *hash*. Refiérase a la sección de algoritmos involucrados para los detalles de la implementación.

En nuestro caso, cada entrada en la tabla de símbolos consta de cuatro campos descritos a continuación:

- Nombre del símbolo.
- Tipo de dato: determina cuántos bytes ocupa en el *heap* un símbolo.
- Código de token.
- Desplazamiento con respecto a la dirección base del *heap*: espacio de memoria que ocupa.

Administración de memoria

La administración de memoria se realiza en base a *heaps*. Los *heaps* son áreas de memoria que almacenan los valores de las variables. Cada heap se compone de 3 campos:

- Un campo que indique el tamaño del heap.
- Un campo que indique el offset actual.
- Y otro campo que indica la dirección base del heap.

Cuando se declara una nueva variable se le asigna el offset actual del heap al símbolo. Luego se actualiza el offset sumándole tantos bytes como indique el tamaño del tipo de dato. En nuestro caso dispondremos de un heap por cada registro de activación. El offset asignado se guarda en el cuarto campo de cada entrada en la tabla de símbolos.

Antes de asignar el offset actual a un símbolo se verifica si el offset actualizado excedería el tamaño del heap. De darse el caso se aumenta la capacidad del heap. Después de realizar la verificación se procede con los pasos mencionados en el párrafo anterior.

Registros de Activación

Es aquí donde se guarda la información necesaria para la ejecución de un procedimiento o de una función. Nos referiremos a ambos en general como “operaciones”. Para realizar una llamada a una operación cualquiera se utilizan bloques continuos de almacenamiento llamados Registros de Activación (*Activation Records*) o Marcos (*Frames*). En lenguajes como Pascal o C es práctica común el usar una pila para manejar los registros de activación. Así, una llamada a una operación implica empujar un nuevo registro en la pila (*push*), en tanto que el regresar de la llamada implica jalar el último registro empujado (*pop*).

Antes de proceder a explicar la estructura de los registros de activación introduciremos la estructura que guarda la información de las operaciones declaradas, la cual llamaremos TOperacion:

- Nombre de la operación
- Tipo de dato que devuelve (sólo usado en funciones).
- La entrada de un símbolo con el mismo nombre que la operación (sólo usado en funciones).
- La activación que realiza la llamada.
- Un arreglo con las direcciones de los parámetros en la tabla de símbolos de la operación.
- La instrucción con la cual empieza la operación.
- El número de parámetros de la operación.
- Profundidad de la operación (información vital para el uso de variables locales y globales dentro de funciones anidadas, tipo Pascal. Importante al momento de calcular el alcance de las variables).
- Número de variables temporales usadas.
- Tabla de símbolos local a la operación.
- Un offset para calcular el tamaño del heap que usará la operación.

Ahora presentaremos la estructura utilizada para los registros de activación:

- La operación. Campo que apunta a una entrada del arreglo de operaciones declaradas.
- Tipo de dato que devuelve. Redundancia para evitar el tener que ir hasta la tabla de operaciones declaradas.
- Entrada del símbolo homónimo a la operación en la tabla de símbolos local (usado sólo con funciones). Este símbolo es el que se usa para devolver el resultado de la función.
- El contador de programa antes de realizar la llamada a la operación.
- El número de parámetros de la operación.
- El número de variables temporales en la tabla de símbolos.

- La tabla de símbolos de la operación.
- Un campo llamado “acceso”. Campo que apunta a la entrada de la tabla de símbolos del registro de activación llamante, que recibirá el resultado de la función. Este campo no se utiliza con los procedimientos.
- El heap (como bloque de memoria) de la activación.

Algunos aspectos de la implementación a resaltar

- Se debe disponer de un arreglo del tipo TOperacion (no importa si es de tamaño fijo pero suficientemente grande) para poder declarar las operaciones. Este arreglo se va llenando conforme se encuentran nuevas operaciones en el programa fuente. La declaración de operaciones es responsabilidad del front end del compilador (abarca las fases de análisis léxico, análisis sintáctico y análisis semántico).
- Es importante notar que el arreglo de operaciones declaradas existe tanto en tiempo de compilación (término válido a pesar de estar desarrollando un intérprete) como en tiempo de ejecución.
- La pila de registros de activación o simplemente *Stack*, existe solamente en tiempo de ejecución.
- Siempre que se encuentre una llamada a una operación, sea función o sea procedimiento, se deberán copiar los campos pertinentes de la operación en cuestión. Esto incluye: el tipo de dato que devuelve, la entrada del símbolo homónimo, el número de argumentos de la operación, el número de variables temporales y especialmente TODA la tabla de símbolos. Los demás campos (la operación, el contador de programa previo a la llamada a la operación y el campo de acceso) deberán establecerse directamente. Además se tendrá que asignar la memoria respectiva al heap. Según nuestra representación intermedia, las operaciones (en el contexto del código intermedio) responsables de realizar este trabajo son CALLF (para llamadas a funciones) y CALLP (para llamadas a procedimientos). Las operaciones CALLF y CALLP

también son responsables de saltar hacia el código de inicio de la función o del procedimiento respectivo.

- Los argumentos necesarios para las llamadas a las funciones (y procedimientos) son insertados en la tabla de código con el código de operación PSHP (por PuSH Parameter). Así, la entrada inmediata anterior al código CALLF o CALLP siempre coincidirá con el último parámetro de la función. Luego, teniendo el número de parámetros es posible retroceder en la tabla de código y copiar los argumentos desde la tabla de símbolos que realiza la llamada hacia la tabla de símbolos de la nueva activación.
- Se prosigue con la ejecución normal del código intermedio hasta encontrar la operación RETF o RETP. Estas operaciones (refiriéndonos a operaciones de código intermedio) indican el fin del código intermedio de La Operación (en el sentido de función o procedimiento). Entonces, se reemplaza el contador de programa actual por el valor guardado en el último registro de activación y se copia el resultado de La Operación (de ser una función) a la variable indicada por el campo de acceso en el registro de activación.
- Dado que este intérprete no está pensado para soportar asignación dinámica de memoria es factible establecer el tamaño de cada heap al valor final del offset que tiene cada operación (heap de tamaño fijo).

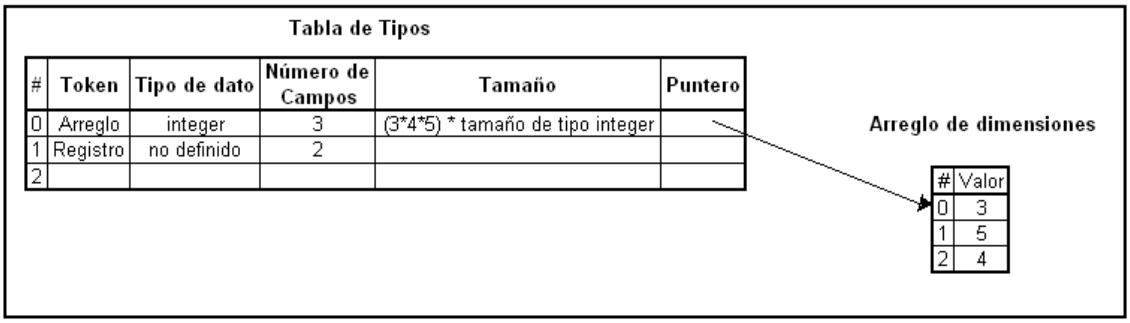
Tipos de datos definidos por el usuario

El intérprete soporte la creación de nuevos tipos de datos gracias a la estructura TType. Entre los tipos de datos que se podrían definir están los arreglos multidimensionales, registros (tipo Record en Pascal) y cadenas (*strings*) de longitud variable.

La estructura está diseñada para contener el número de campos (si fuese un registro) o dimensiones (en el caso de los arreglos) del nuevo tipo de dato. También es necesario guardar el tamaño total del nuevo tipo de dato, al igual

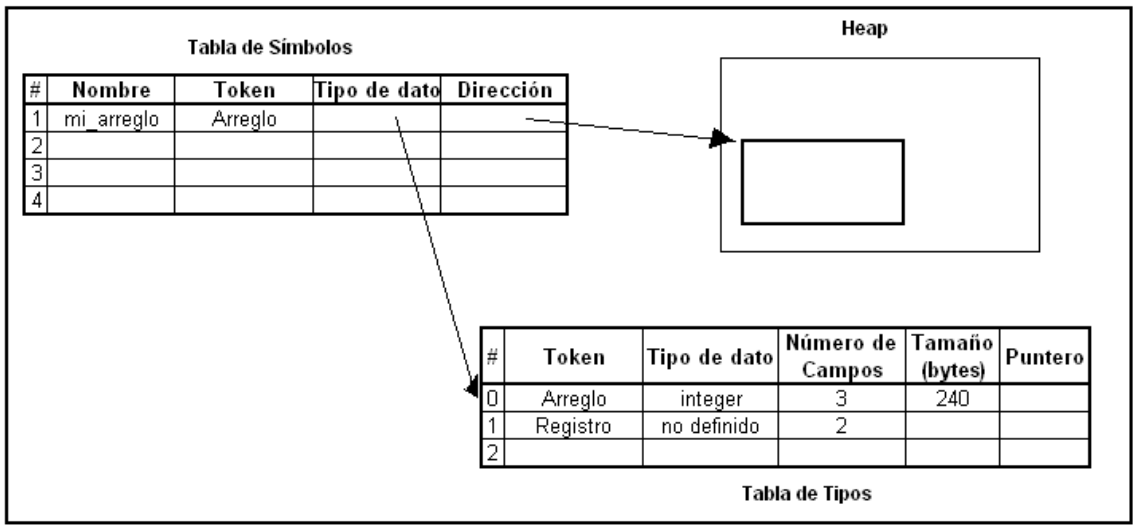
que el tipo de dato base que lo conforma (caso de las cadenas y los arreglos) y la información relacionada a cada campo/dimensión que se defina (excepto para las cadenas).

La información concerniente a los campos que debe almacenarse depende de si se trata de un registro o de un arreglo. Para los registros es necesario guardar el nombre del campo, el tipo de dato (o su ancho en bytes) y su desplazamiento con respecto a la dirección base que tenga. En cambio, para los arreglos sólo es obligatorio guardar el valor máximo de cada dimensión.



Ejemplo de Tabla de Tipos

Cuando se declara y define una variable de tipo de dato compuesto, el campo “tipo de dato” de dicho identificador en la tabla de símbolos refiere a la entrada respectiva en la tabla de tipos, mientras que el campo dirección apunta al bloque de memoria asignado.



Relaciones entre la tabla de símbolos, tabla de tipos y montículo de memoria

Una vez más se deja en claro que el intérprete soporta los tres tipos de datos compuestos mencionados anteriormente: arreglos multidimensionales, registros y cadenas de longitud fija. Sin embargo, solamente la primera de éstas ha sido implementada: los arreglos multidimensionales.

Algoritmos involucrados

Tablas de Hash

Las tablas de hash son una generalización del uso ordinario de arreglos. Un buen mecanismo de hash debe soportar, por lo menos, las operaciones de inserción y búsqueda. Opcionalmente se podría implementar la operación de eliminación. Las tablas de hash son una técnica efectiva y práctica. Su característica más resaltante es que las operaciones de diccionario requieren en promedio de un tiempo constante para poder ejecutarse.

Las tablas de hash son posibles gracias a las funciones de hash. La idea es que la función *h* asocie un valor *h(k)* (posición en la tabla) a cada llave *K* (un elemento que queramos insertar o buscar en la tabla). La función h debe devolver siempre el mismo resultado para cada *K*.

El problema aquí es la posibilidad de tener colisiones. Por colisiones se entiende que dos o más llaves podrían coincidir en una misma entrada en la tabla. A continuación presentamos una serie de técnicas que resuelven el conflicto de colisiones:

- Resolución de colisiones por encadenamiento (uso de listas enlazadas en cada entrada de la tabla)
- Método de división
- Método de multiplicación
- Hashing universal
- Direccionamiento abierto.

Utilizaremos la estrategia de direccionamiento abierto para la administración de la tabla de símbolos. El “Direccionamiento Abierto” (Open Addressing) o también “Hashing Abierto” (Open Hashing). Con esta técnica todos los elementos son almacenados en la tabla misma. De esta forma, cada entrada contiene un elemento del conjunto dinámico o un apuntador a NIL (elemento vacío). Luego, al realizar una búsqueda se examina sistemáticamente las entradas de la tabla hasta encontrar el elemento deseado o estar seguro de que éste no exista.

En contraste con la técnica de listas enlazadas, este método evita intencionalmente el uso de punteros. Su manejo de resolución de colisiones se basa en la capacidad de calcular la secuencia de entradas a revisar. El espacio de memoria que normalmente se utilizaría con las listas enlazadas se usa aquí para tener un mayor número de entradas en la tabla, dando origen a menos colisiones e incluso a un tiempo de búsqueda menor.

Para realizar inserciones se examinan sucesivamente la tabla de hash hasta encontrar una entrada vacía. En lugar de probar secuencialmente las entradas 0,1,…, m-1 (siendo m el tamaño de la tabla), la secuencia de prueba depende de la llave *k* a insertar. Entonces, se determina la secuencia de prueba extendiendo el prototipo de la función de hash a *h(k,s).* Donde *s* representa el número de

secuencia a examinar. De esta manera, la secuencia de prueba sería *{h(k, 0), h(k, 1), . . . ,h(k, m-1)}* y además se garantiza que esta sea un permutación de {0,1,. . ., m-1}.

Pseudo código de inserción

```
Insertar_hash (T, k)
i ← 0
repeat
        j ← h(k, i)
        if T[ j ] = NIL then
                T[ j ] ← k
                return j
        else
        i ← i + 1
until i = m
return devolver “tabla llena”
```

El algoritmo de búsqueda revisa las mismas entradas que el de inserción. Por tanto, la búsqueda pueda terminar en fracaso si encuentra una entrada vacía (porque la llave k debió estar en una de las entradas analizadas). Este argumento presupone que no se borrarán entradas de la tabla, como es el caso de la tabla de símbolos.

Pseudo código de búsqueda

```
Buscar_hash (T, k)
i ← 0
repeat
        j ← h(k, i)
        if T[ j ] = k then return j
        i ← i + 1
until T[ j ] = NIL o i = m
```

return NIL

Disponemos de tres técnicas ampliamente usadas para calcular las secuencias de prueba. Estas son:

- Prueba linear.
- Prueba cuadrática.
- Doble hashing.

Elegimos la tercera de estas para implementar nuestra función hash. El doble hashing tiene la siguiente forma:

$h(k, i) = (h_1(k) + i * h_2(k))$ mod m

Donde:

- la función h_1 y h_2 son funciones auxiliares de hash.
- La primera entrada de prueba es $h_1(k)$.
- Las siguientes entradas a probar son desplazamientos en $h_2(k)$, módulo m, a partir de la primera.

Aparece un problema. El valor $h_2(k)$ debe ser un primo relativo al tamaño m de la tabla para garantizar que la tabla completa sea barrida en una secuencia de prueba. Si m y $h_2(k)$ tienen un divisor común d > 1 para algún k, entonces la búsqueda sobre dicha llave sólo barrería la 1/d parte de la tabla. Disponemos de dos enfoques para nuestro problema:

- Permitir a m ser una potencia de 2 y diseñar $h_2(k)$ de tal forma que siempre devuelva un número impar.
- Permitir a m ser un número primo y diseñar $h_2(k)$ para que devuelva cualquier número menor que m.

Adoptamos la primera de las opciones dadas para $h_2(k)$. En cuanto a la función $h_1(k)$ solamente mencionaremos que implementa el hashing por el método de multiplicación.

Indexación de arreglos

Supóngase que se define un arreglo de la siguiente manera:

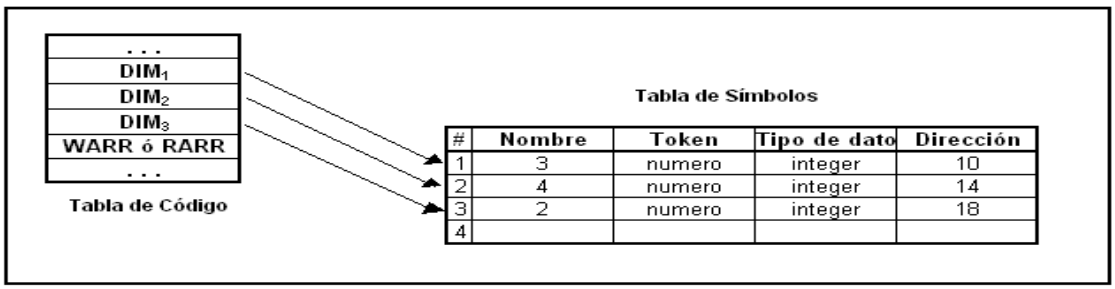
```
var mi_arreglo : array [ 3, 5, 4 ] of integer;
```

y luego se realiza la siguiente asignación:

```
mi_arreglo[ 3, 4, 2] := 100;
```

Lo que hace el intérprete primero es reservar el bloque de memoria para el arreglo. Las dimensiones máximas son 3, 5 y 4, para la primera, segunda y tercera dimensión, respectivamente. Suponiendo que el tipo de dato *integer* mide 4 bytes, entonces se le asigna a la variable *mi_arreglo* un bloque de 3x5x4x4 = 240 bytes. Además, se agrega la entrada correspondiente en la tabla de tipos.

El intérprete implementa el manejo de arreglos mediante dos operaciones de código intermedio. Éstas operaciones son WARR (Write in ARRay) y RARR (Read from ARRay). La idea general es insertar en la tabla de código las direcciones correspondientes en la tabla de símbolos, de cada dimensión del arreglo. Estas se guardan bajo el código de operación DIM y habrán tantos DIM como dimensiones tenga el arreglo. Inmediatamente después del último DIM aparece la operación que se desea realizar: WARR o RARR. Cada una de estas es responsable de calcular el *offset* (desplazamiento) relativo a la dirección base de memoria que se asignó a la variable al momento de definirla.



Ejemplo de tabla de código para ejecutar un WARR o RARR

Pseudo código para cálculo del offset

{ N es el número de dimensiones del arreglo}

offset ← 0

i ← 1

repeat

offset ← offset + (DIM_i – 1)

offset ← offset * MAX_DIM_i

i ← i + 1

until i = N

offset ← offset + (Valor DIM_N – 1)

offset ← offset * tamaño_del_tipo_de_dato

La operación MATCH

Esta operación de código intermedio implementa la evaluación y selección de instrucciones a ejecutar para la sentencia CASE de nuestro intérprete. A continuación presentamos el subconjunto de reglas que implementa la instrucción:

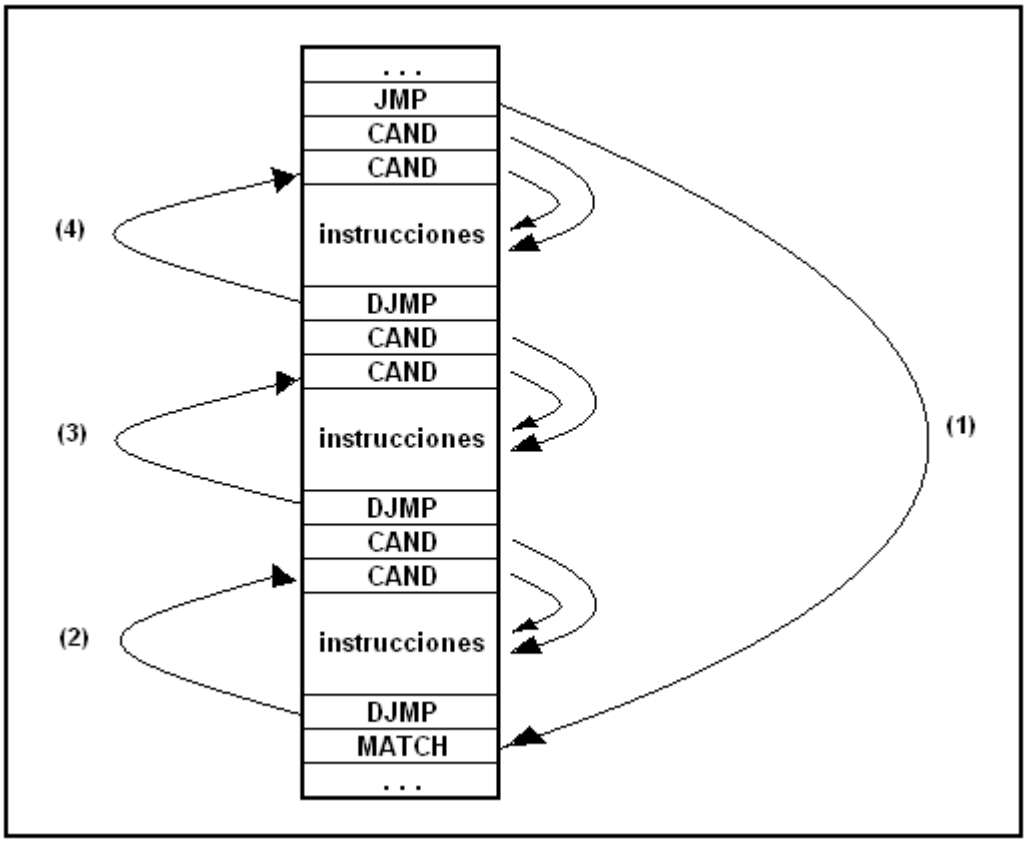
<instrCase> ::= CASE <exp.> OF <optionList> END

<optionList> ::= <optionList> <option> ‘;’ | <option> ‘;’

<option> ::= <candidateList> ‘:’ <instrGroup>

<candidateList> ::= <candidateList> ‘,’ <candidate> | <candidate>

<candidate> ::= TYPE_INTEGER | ID



Ejemplo de código intermedio para la instrucción CASE

Descripción de cada tipo de entrada

- **JMP**: Realiza un salto incondicional a la instrucción MATCH.
- **MATCH**: Busca la primera coincidencia de la llave (<expr> en la primera regla) a partir del último <candidateList> hasta el primero. Además, guarda la dirección del JMP como su segundo operando.
- **DJMP**: Almacena la dirección del último y el primer CAND de un <candidateList> en los campos de su primer y segundo operando, respectivamente.
- **CAND**: Número de entrada del <candidate> (candidato) en la tabla de símbolos.

La idea general del algoritmo consiste en seguir las direcciones hacia las CAND guardadas en DJMP. Luego, comparar cada CAND contra la llave. Si ningún

CAND de un <candidateList> concuerda con la llave se prosigue con el anterior DJMP o hasta llegar a la instrucción JMP.

A continuación se presenta el pseudo-código que implementa la operación MATCH:

```
continuar ← true
codigo ← codigo_actual – 1
cod_fin_case ← codigo_actual + 1
while ( continuar and codigo > direccion de JMP) begin
        Operando 3 de DJMP ← cod_fin_case
        cod_primer_candList ← Operando 2 de DJMP
        codigo ← Operando 1 de DJMP
        direcc_instrucciones ← codigo + 1
        while (codigo >= cod_primer_candList) begin
                candidato ← Operando 1 de CAND
                if (candidato = llave) then begin
                        codigo_actual ← direcc_instrucciones
                        continuar ← false
                end
        end
        codigo ← codigo – 1
end
if (continuar) codigo_actual ← codigo_actual + 1
```

El vector de ejecución (Display)

La idea del Display es permitir el acceso a variables no-locales a alguna función o procedimiento. El algoritmo es bastante sencillo. Este consiste en utilizar un arreglo de enteros para guardar y poder conocer el último registro de activación ejecutado para algún nivel de profundidad i, determinado en tiempo de compilación, y así poder tener acceso rápido a variables no locales.

Cuando se prepara un nuevo registro de activación, declarado en una profundidad i, se hace lo siguiente:

- Guardar el valor Display[i] en el nuevo registro de activación.
- Establecer Display[i] para que apunte al nuevo registro de activación.
- Luego, cuando este terminando la activación se reestablece el Display[i] al valor guardado en el registro de activación apuntado por el mismo Display[i].

AMBIENTE DE DESARROLLO

Generadores de Compiladores

Yacc (Yet Another Compiler-Compiler) es una herramienta que sirve para describir la estructura sintáctica de los lenguajes de programación. Yacc recibe como entrada el conjunto de reglas que conforman la gramática del lenguaje. Cada regla (o símbolo no terminal) puede estar compuesta por otras reglas o símbolos terminales (tokens).

Yacc permite insertar dentro de cada regla bloques de código fuente (por lo general en C, o cualquiera que sea el lenguaje que use dicha versión del Yacc), llamadas acciones, las cuales se ejecutan al identificar un símbolo no terminal o token en particular.

Al ejecutar el programa Yacc se obtiene un archivo fuente en C (o del lenguaje que soporte) que contiene principalmente dos funciones:

- yyparse: Función responsable de realizar el análisis sintáctico a los programas, es decir, verificar si el programa escrito en dicho lenguaje coincide con la sintaxis de las reglas que componen la gramática.
- yylex: Función encargada de desglosar el programa de entrada en unidades elementales (tokens) y proveérselos al yyparse para que éste determine qué regla de la gramática debe seguir.

Bison, al igual que Yacc, es un generador de parsers de propósito general el cual convierte la descripción gramatical de una gramática LALR de contexto libre en un programa de C capaz de realizar el análisis sintáctico de programas que cumplan con dicha gramática.

Librerías APR (Apache Portable Runtime)

El APR es un proyecto del Apache Software Foundation cuyo fin es crear y mantener librerías de software que provean una interfase predecible y

consistente a implementaciones específicas de plataforma. Su objetivo primario es proveer a los desarrolladores de software con un API que les garantice comportamientos similares de su software, si no idénticos, sin importar en qué plataforma esté hecha, liberándolos de la responsabilidad de lidiar directamente con las deficiencias inherentes de cada plataforma.

Desde otro punto de vista, se puede considerar las librerías APR como una abstracción de la función que realiza un sistema operativo. A continuación se presentan algunas de las funcionalidades que provee el APR:

- Manejo de archivos: apr_file_io.h
- Manejo de memoria: apr_pools.h
- Manejo de tablas (para las cabeceras MIME): apr_table.h
- Librerías de compatibilidad:
    - apr_compat.h
    - apr_general.h
    - apr.h
    - apr_portable.h
- Librerías relacionadas al servidor y al protocolo HTTP:
    - http_config.h
    - http_connection.h
    - http_core.h
    - http_log.h
    - http_main.h
    - http_protocol.h
    - http_vhost.h
    - httpd.h
    - http_request.h
- Manejo de hilos:
    - apr_thread_cond.h
    - apr_thread_mutex.h
    - apr_thread_proc.h

    - apr_thread_rwlock.h
- Manejo de cadenas: apr_strings.h
- Tablas de hashing: apr_hash.h

En algunos casos, lo que hace la librería es conformar una especie de envoltorio o wrapper que realiza tareas especiales para luego llamar a las librerías de C, como sucede con las funciones de apr_strings.h, apr_file_io.h y apr_pools.h (asociada a las librerías string.h, stdio.h y las funciones de manejo de memoria de stdlib.h, respectivamente). En otros, casos se trata de implementaciones necesarias para el correcto funcionamiento del servidor web y la recepción/atención de solicitudes.

La estructura request_rec

Este registro es el corazón del APR. Contiene toda la información que se podría necesitar acerca de la solicitud actual. La estructura describe una solicitud hecha al servidor, por parte de un cliente (navegador) para una página determinada. Combina información pública que usan los módulos comúnmente con información privada que sólo tendría significado para el núcleo del servidor. En la mayoría de los casos, cada conexión al cliente genera una sola estructura *request_rec*.

La información más importante disponible en esta estructura es un pequeño conjunto de caracteres describiendo atributos del objeto solicitado, esto incluye el URI (Universal Resource Identifier), el nombre de archivo, tipo de contenido y codificación del contenido (los cuales son llenados por los manejadores de traducción y de revisión de tipo que atienden la solicitud).

En orden de importancia le siguen atributos que refieren al estado de la solicitud, a los métodos de transferencia permitidos (GET y/o POST), a los métodos de lectura del cuerpo de la solicitud, el nombre del handler (manejador) del contenido, un puntero a la estructura del servidor (*server_rec*), un puntero a la

configuración de directorios (*per_dir_config*) y otro más a la configuración de la solicitud (*request_config*). Son principalmente estos campos los que se usan para atender las peticiones recibidas.

Otro de los campos comúnmente usados (aunque no en este módulo) son las tablas que transportan las cabeceras MIME de la solicitud original, las cabeceras MIME que deben ser enviadas como respuesta (a los cuales los módulos pueden agregar cualquier cabecera que deseen) y las variables de entorno para cualquier subproceso activado en el curso de atención de la solicitud. Estas tablas son manipuladas mediante las rutinas *apr_table_get* y *apr_table_set*, declaradas en la cabecera de archivo *apr_tables.h*.

Dejamos para el final uno de los atributos esenciales de la estructura: el puntero a un *Resource Pool*. A continuación se explicará la idea que envuelve el uso de *pools* en el APR.

Resource Pools

Un aspecto que ha sido bastante cuidado por el APR es la administración de la memoria. Con la finalidad de evitar fugas y la degradación del desempeño del servicio, se crea una nueva estructura llamada *apr_pool_t*, *resource pool* o simplemente *pool* (estanque o pozo de memoria). Luego, siempre que se necesite asignar memoria en tiempo de ejecución se tendrá que realizar una llamada a alguna de las funciones declaradas en *apr_pools.h*. Todas las funciones que aparecen ahí son adaptaciones de otra en C, con la diferencia de que siempre está presente un parámetro del tipo apr_pool_t, el cual identifica al *pool* padre (ancestro inmediato) o el *pool* del cual se obtendrá la memoria requerida. Así, cada variable *pool* tiene un padre, varios ancestros e incluso varios hijos.

El uso de pools se hace presente en varios aspectos del desarrollo:

- Al pedir asignación dinámica de memoria: lo que normalmente sería una llamada a *calloc* aquí se convierte en *apr_pcalloc* pasando como parámetro el *pool* padre.
- Al trabajar con archivos: para abrir cualquier archivo, ya sea en modo de lectura o escritura se asocia dicho archivo a un resource *pool*.
- Al trabajar con cadenas: APR ofrece una serie de funciones de manejo de cadenas. Como ejemplo de éstas tenemos duplicación de cadenas, copia parcial de cadenas, concatenación, etc. Todas ellas necesitan un parámetro del tipo *apr_pool_t*.

La ventaja de trabajar con los *pools* consiste en el hecho de que no hay que liberar el recurso utilizado. La estructura misma rastrea los recursos en uso. Una vez que la solicitud ha sido procesada, el *pool* es limpiado. En ese momento, la memoria asociada con él es liberada para ser reutilizada posteriormente, los archivos asociados son cerrados y se lleva a cabo cualquier otra función de limpieza asociada al *pool*. Al acabar con esto, se tiene total certeza de que todos los recursos han sido devueltos y que no existe ninguna fuga de memoria.

Un resource *pool* se define primordialmente por su tiempo de vida. A continuación se presenta la jerarquía de *pools* en la que se basa el Apache:

- permanent_pool: Es el ancestro de todos los pools y ninguna función lo recibe como parámetro.
- pconf: Subpool de *permanent_pool*. Es creado al principio de un ciclo de configuración (inicio o reinicio de servidor). Existe hasta el momento de detener o de reiniciar el servidor. Es pasado como parámetro a todas las rutinas de configuración, ya sea por medio de *cmd->pool* o como "*pool *p*".
- pchild: Subpool de *permanent_pool*. Es creado al momento de engendrar un hijo (crear un hilo) y vive hasta el momento en que éste se destruye.

- ptrans: Debería ser un subpool de pchild, pero actualmente lo es de *permanent_pool*. Es limpiado por el hijo antes de entrar al bucle de recepción de conexiones. Utilizado como *connection->pool*.
- r->pool (pool de request_rec): Para las solicitudes principales es un subpool de *connection->pool*; mientras que para sub-solicitudes es un *subpool* del *pool* de la solicitud padre. Su tiempo de vida se extiende tanto como demore la atención de la solicitud.

Una observación importante relacionada a *request_rec* y a su atributo *pool*. La memoria para el mismo *request_rec* es obtenida de *r->pool*, en otras palabras, primero se crea el *pool* de *r->pool* y lo primero en localizarse en ese *pool* es el mismo *request_rec*.

Funciones Hook

Una *Función Hook* o simplemente *hook* es, en general, cualquier función que el servidor podría llamar al procesar una solicitud. Los módulos tienen la capacidad de declarar y definir sus propias *Funciones Hook*, además de establecer en qué momento de la atención de la solicitud se llamará y el orden, con respecto a las *Funciones Hook* implementadas por otros módulos.

El APR ofrece una serie de *hooks*, donde cada una de éstas se encuentra asociada a una fase de atención de la solicitud. Aquí explicamos la función realizada por algunos de los *hooks*:

- Hook translate_name: Permite a los módulos traducir “manualmente” el campo URI del request_rec en un nombre de archivo. Si todos los módulos rechazan esta fase las reglas predeterminadas del servidor son aplicadas.
- Hook map_to_storage: Ofrece a los módulos la oportunidad de establecer el campo *per_dir_config* de *request_rec* basados en su propio contexto y responder a solicitudes sin contexto tales como TRACE que no necesitan seguridad o mapeo al sistema de archivos.

- Hook create_request: Brinda al módulo la oportunidad de crear y configurar a gusto el atributo *request_config* de *request_rec* al momento de concebir la nueva solicitud.
- Hook handler: Ofrece al módulo la posibilidad de servir el contenido pedido mediante una función personalizada que responderá antes que
- Hook type_checker: Esta rutina es llamada para determinar o establecer los bits de información relacionados con el tipo de contenido (r->content_type), idioma, etc.
- Hook fixups: Permite al módulo realizar ajustes en los campos de las cabeceras antes de servir cualquier contenido.

Manejo de cadenas

El APR posee una librería de cadenas (declaradas en *apr_strings.h*) con funciones bastante parecidas a aquellas definidas en la librería *string.h*. de C. Estas se pueden clasificar de la siguiente forma:

- Funciones de comparación

apr_strnatcmp	Realiza la comparación natural de caracteres distinguiendo mayúsculas de minúsculas.
apr_strnatcasecmp	Realiza la comparación natural de caracteres ignorando la diferencia entre mayúsculas y minúsculas.

- Funciones de copia, duplicación y concatenación de cadenas. Las funciones enumeradas a continuación tienen como primer parámetro el *pool* del cual adquieren la memoria necesaria:

apr_pstrdup	Tiene el mismo efecto que las llamadas sucesivas a *calloc* y *cpystr.*
apr_pstrndup	Tiene la misma finalidad que apr_pstrdup pero limitándose a ‘n’ caracteres.
apr_pstrcat	Función de parámetros variables que concatena

	dos o más cadenas. El último parámetro siempre debe tener el valor NULL.
apr_psprintf	Semejante a la función sprintf de C, impresión con formato en una cadena.

Otras funciones que resultan bastante útiles son la función *apr_strtok* para dividir una cadena en tokens, indicando uno o más caracteres de separación; y *apr_collapse_spaces* para remover espacios en blanco. Ninguna de estas asocia un *resource pool* a las cadenas que maneja.

Manejo de archivos

La librería de archivos del APR es bastante completa. Consta de algunas funciones estándar de lectura y escritura de archivos, inspiradas en las versiones de C y otras no tan estándar pero igualmente útiles. El APR no utiliza el tipo de dato "*FILE* * " como en C, mas bien define su propio tipo para manejo de archivos: *apr_file_t*. Aquí tenemos una agrupación tentativa de las funciones declaradas en *apr_file_io.h*:

- Funciones de creación y eliminación de archivos. Siempre enlazadas con un *resource pool* mediante el último parámetro de la función:

apr_file_open	Apertura de archivo y creación del mismo de ser necesario.
apr_file_remove	Eliminación de archivos.
apr_file_rename	Modifica el nombre de un archivo.
apr_file_copy	Realiza una copia completa de un archivo.
apr_file_append	Une un archivo a otro.
apr_file_open_stderr	Abre el archivo de error estándar.
apr_file_open_stdin	Abre el archivo de entrada estándar.
apr_file_open_stdout	Abre el archivo de salida estándar.
apr_dir_make	Creación de directorios
apr_dir_remove	Eliminación de directorios

- Funciones de lectura y escritura. Aquí no queda muy clara la razón por la cual los desarrolladores del APR decidieron colocar en algunos casos el objeto *apr_file_t* como primer parámetro y en otros como último:

apr_file_read	Similar a la función *read* de C. El archivo es el primer parámetro.
apr_file_write	Similar a la función *write* de C. El archivo es el primer parámetro.
apr_file_read_full	Tiene el mismo fin que *apr_file_read* pero asegurando que el buffer esté lleno al volver de la llamada. El archivo es el primer parámetro.
apr_file_write_full	Tiene el mismo fin que *apr_file_write* pero garantizando que todo el contenido del buffer sea escrito. El archivo es el primer parámetro.
apr_file_printf	Similar al *fprintf* de C. El archivo es el primer parámetro.
apr_file_putc	Similar al *fputc* de C. El archivo es el último parámetro.
apr_file_getc	Similar al *fgetc* de C. El archivo es el último parámetro.
apr_file_ungetc	Similar al *ungetc* de C. El archivo es el último parámetro.
apr_file_gets	Similar al *fgets* de C. El archivo es el último parámetro.
apr_file_puts	Similar al *fputs* de C. El archivo es el último parámetro.

- Otras funciones:

apr_file_eof	Determina si el archivo ya ha llegado a su fin.
apr_file_seek	Mueve el *offset* del archivo una serie de bytes de acuerdo a las constantes: APR_SET,

	APR_CUR y APR_END.
apr_file_close	Cierra el archivo pasado como parámetro
apr_file_pool_get	Obtiene el pool asociado al archivo.

Escritura de contenido

Escribir contenido se refiere al hecho de enviar datos al cliente, es decir, responder a la petición realizada por él devolviendo la página web solicitada. La escritura de contenido se realiza por el *Content Handler*, una función asociada a la fase de Handling mediante el *Hook Handler*. Antes de servir cualquier contenido es necesario enviar las cabeceras HTTP pertinentes, mas aún, es posible devolver solamente las cabeceras HTTP sin contenido posterior.

Las funciones de escritura de contenido se encuentran declaradas en *http_protocol.h*. Todas ellas presentan el prefijo “ap_r” donde la letra “r” hace referencia a un *request_rec*, parámetro obligatorio para cualquier llamada.

ap_send_http_header	Envía la línea de estado y todas las cabeceras HTTP que contengan los campos *headers_out* y *err_headers_out* de request_rec. La función agrega automáticamente unas cuantas cabeceras tales como la fecha y el servidor.
ap_rwrite	Envía una cantidad fija de bytes a partir del buffer recibido como parámetro
ap_rputs	Escribe una cadena de longitud arbitraria.
ap_rvputs	Similar a *ap_rputs* pero aceptando un número variable de cadenas a escribir. La última de estas debe tener el valor NULL.
ap_rprintf	Escritura de contenido con formato al estilo del *printf* de C.
ap_rflush	Fuerza el envío de cualquier flujo hacia el cliente.

Lectura de datos enviados por el cliente

El comportamiento predeterminado del Servidor Apache es leer todas las cabeceras de la solicitud. Estas se separan del cuerpo del mismo al encontrar el retorno de carro/nueva de línea. El Apache utiliza la información de las cabeceras para establecer los diversos valores de las estructuras *connection_rec*, *server_rec* y especialmente de *request_rec* (Nota: *request_rec* presenta un puntero a cada una de las dos primeras estructuras mencionadas).

A partir del retorno de carro comienza el *request body*. Esta sección del *request* contiene los datos enviados por los formularios (vía método POST) así como cualquier documento subido al servidor (vía método PUT).

Al leer los datos del cliente se presentan dos problemas. El primero de ellos es la posibilidad de perder la conexión con el cliente antes de que este termine de enviar todos los datos. En este contexto, la solución es establecer un *timeout* para que el servidor no espere indefinidamente. El segundo problema es el tipo de dato *chunked* de HTTP 1.1, en donde la data es transferida en pequeños bloques siempre precedida de la cantidad de bytes enviados. El problema aquí es que no habrá una cabecera de *Content-length* en el *request* que diga cuántos bytes hay que leer.

Entonces, las soluciones a los problemas expuestos son:

- Establecer un timeout haciendo una llamada a la función *ap_hard_timeout* previa a cualquier lectura de data.
- Seleccionar una de las tres pólizas de lectura disponibles. Por lo general, se utilizará REQUEST_CHUNKED_ERROR.

Actualmente el APR soporta tres pólizas de lectura:

- REQUEST_NO_BODY: Se retorna el código de error HTTP 413 “HTTP request entity too large” si se encuentra cualquier contenido en el request body. Es decir, no se esperan datos por parte del cliente.
- REQUEST_CHUNKED_ERROR: Se acepta el envío de datos por parte de los navegadores, sin embargo, se devolverá el error HTTP 411 “HTTP length required” en cualquier intento de transmitir datos del tipo chunked.
- REQUEST_CHUNKED_DECHUNK: Se aceptan ambos tipos de datos: del tipo *chunked* y el normal (un solo bloque). Si es del tipo *chunked* Apache integrará los bloques recibidos del cliente.

El APR provee las siguientes funciones para la lectura del *request body*, las tres muy necesarias:

ap_setup_client_block	Antes de leer los datos del cliente se debe realizar una llamada a esta función. Esto le indica al Apache que se está listo para iniciar la lectura y establece el estado interno de lectura para fines de control (manipulando los campos *clength*, *remaining* y *read_length* del objeto *request*). La función tiene dos argumentos: el request y la póliza de lectura.
ap_should_client_block	Esta función devuelve valores booleanos (según el estándar de C) e indica si es factible proceder con la lectura. Al utilizar el protocolo HTTP/1.1, una llamada a *ap_should_client_block* le envía un código 100 “Continue” al navegador, diciéndole que es momento de empezar a transferir sus datos. Su único parámetro es el *request* del cliente.
ap_get_client_block	Esta es la función que se encarga realmente de la lectura. Recibe como parámetros el *request* original, un buffer donde devolver los datos leídos

	y un entero que representa el tamaño del buffer. Si se está manejando un bloque de lectura único (nonchunked data), la lectura no debe excederse de la cantidad de bytes indicados por *Content-length*, de lo contrario la función podría bloquearse indefinidamente.

Una alternativa a usar el método de transferencia POST es el método GET. Ambos métodos son capaces de enviar los mismos datos. Sin embargo, la gran diferencia radica en que el método GET es mucho menos seguro debido a que los datos son concatenados al URL solicitado y enviados juntos. Luego, el Apache recibe estos datos en el campo *the_request* del objeto *request*.

El APache eXtenSion tool (APXS)

El APXS es una herramienta de apoyo para construir e instalar módulos de extensión del Servidor Apache HTTP. Esto se logra generando un Módulo Dinámico Compartido (DSO por sus siglas en inglés) a partir de uno o más archivos fuente u objeto los cuales pueden ser cargados posteriormente al Apache en tiempo de ejecución mediante la directiva "*LoadModule*" implementada en el módulo *mod_so*.

Para poder hacer uso de este mecanismo de extensión la plataforma debe soportar la característica DSO. Además, el binario *httpd* del Apache debe ser construido junto con el módulo *mod_so*. De no darse este caso, el *apxs* no podrá construir el módulo DSO correctamente.

El MÓDULO

Estructura de un módulo Apache

Los módulos Apache se componen de una estructura compleja. A pesar de ello, se puede distinguir una organización por secciones donde cada una es responsable de un trabajo particular. La estructura que presentamos a continuación es sólo un intento de encontrar una distribución lógica a las diferentes secciones.

- Sección 1: Inclusión de cabeceras
- Sección 2: Definición de nuevas estructuras. Declaración y definición de variables globales al módulo.
- Sección 3: Funciones de configuración.
- Sección 4: Handlers.
- Sección 5: Funciones auxiliares.
- Sección 6: Funciones de definición de comandos para el módulo.
- Sección 7: Registro de funciones hook.
- Sección 8: Estructura de directivas para el módulo.
- Sección 9: Definición del módulo para configuración.

Ahora procederemos a explicar con mayor detalle el rol que desempeña cada sección.

Sección 1 - Inclusión de cabeceras

Como todo programa escrito en C, se necesita incluir una serie de archivos de cabecera. Las cabeceras que por lo menos deben incluirse son:

- httpd.h
- http_config.h
- http_core.h
- http_log.h
- http_main.h
- http_protocol.h
- http_request.h
- http_connection.h

En nuestro caso también se hará uso extensivo de las librerías: apr_strings.h, apr_file_io.h y apr_pools.h.

Sección 2 - Definición de nuevas estructuras. Declaración y definición de variables globales al módulo

En esta sección de definen las estructuras específicas al módulo. Ejemplo de estas son la estructura de configuración del servidor, la estructura de configuración de directorios, la estructura de configuración del *request*, etc. También se declaran aquí aquellas variables globales al módulo. La más importante de estas es la variable que representa la configuración del módulo y se declara de la siguiente forma:

```
module AP_MODULE_DECLARE_DATA mi_modulo
```

Sección 3 - Funciones de configuración

El Apache distingue dos niveles de configuración: a nivel de servidor o mejor dicho de módulo servidor y a nivel de directorios. Por otro lado, también hace diferencia entre las dos fases de configuración para cada nivel: creación y fusión. Así, el Apache primero crea las configuraciones de servidor y de directorios y luego procede a fusionar las configuraciones a cada nivel. Esta combinación de fases por nivel da origen a cuatro funciones:

Fase de creación:

- A nivel de servidor:

```
void * crear_configuracion_de_servidor (apr_pool_t * pool,
                                        server_rec * ptrServidor)
```

  Se crea la estructura de configuración del servidor para el módulo. Esta tarea se hace una vez para el servidor principal y adicionalmente para cada host virtual. Es responsabilidad de esta función el separar memoria para la configuración por servidor y devolver un puntero hacia la estructura de configuración. El segundo parámetro es un puntero al servidor mismo. El prototipo de esta función es como sigue:

- A nivel de directorio:

```
void * crear_configuracion_de_directorio (apr_pool_t * pool,
                                          char * strDirectorio)
```

Está función es llamada una vez por módulo, con el parámetro strDirectorio puesto a NULL durante la inicialización del servidor y posteriormente para cada sección <Directory>, <Location> o <File> que encuentre en los archivos de configuración con strDirectorio apuntando al directorio configurado (aunque sea una sección de ubicación o archivo).

Si un módulo no soporta configuraciones por directorio, cualquier directiva aplicada a una sección <Directory> sobrescribirá la configuración del servidor principal a menos que se tomen precauciones. La forma de hacer esto es usando el atributo *req_overrides* apropiadamente.

Fase de fusión:

- A nivel de servidor:

  void * fusionar_configs_de_servidor (apr_pool_t * pool,
  void * conf_base,
  void * nueva_conf)

  Después de leer los archivos de configuración, esta función es llamada una vez para cada host virtual, con el parámetro *conf_base* apuntando a la configuración del servidor (solamente para este módulo) y *nueva_conf* apuntando a la configuración del host virtual. Esto brinda la oportunidad de heredar opciones no establecidas para el host virtual, heredándolos del servidor principal. La función devuelve un puntero a la configuración fusionada.

- A nivel de directorio:

  void * fusionar_configs_de_directorio (apr_pool_t * pool,
  void * conf_base,
  void * nueva_conf)

  Esta función es llamada una vez por cada host virtual y no por cada directorio como se podría pensar. Se le pasa la configuración del directorio raíz del servidor, en otras palabras, la configuración creada con la llamada a la función *crear_configuracion_de_directorio* (fase de creación a nivel de directorio) con el parámetro *strDirectorio* establecido a NULL.

Cuando una solicitud es atendida esta función combina primera las secciones <Directory> y luego los *.htaccess*, en forma alternada, y bajando por la jerarquía de directorios a partir del directorio raíz. Luego prosigue con las secciones <File> y finalmente con las secciones <Location>.

A diferencia de la función de la fase de fusión a nivel de servidor, está función es llamada al ejecutarse el servidor, con diferentes parámetros, dependiendo de la página requerida. Por esta razón es importante copiar adecuadamente los valores recibidos por el parámetro *nueva_conf* si se piensa aplicar cambios.

Sección 4 - Handlers

Como mencionamos anteriormente, los hooks son funciones que tratan una fase de atención de la solicitud. Ejemplos de éstos son *translate_name, create_request y content_handler*, entre otros.

Sección 5 - Funciones auxiliares

Cuyo fin es apoyar a otras funciones, principalmente a los *Handlers*.

Sección 6 - Funciones de definición de comandos para el módulo

El Apache permite definir nuevas directivas que establecen el comportamiento del módulo. Dichas directivas se utilizan en el archivo de configuración principal del Apache (httpd.conf). Las funciones de esta sección deben llevar a cabo las tareas de las nuevas directivas. El prototipo de estas funciones es como sigue:

```
static const char * funcion_comando(cmd_params * comando, void * config)
```

Sección 7- Registro de funciones hook

Todo *handler* que se implemente en el módulo debe ser registrado y asociado a un *hook* del servidor para que pueda funcionar correctamente. El Apache agrupa todo el registro de *hooks* en una sola función cuyo único parámetro es un *apr_pool_t*, así:

```
static void registrar_hooks(apr_pool_t *p) {
```

```
        ap_hook_nombre_de_hook(nombre, predecesores, sucesores,
    posicion)
        // demás registro de hooks
    }
```

Donde:

- nombre, es el nombre del *handler* que se desea asociar al *hook.*
- predecesores, es la lista de módulos cuya llamada al *hook* debe realizarse antes que el módulo actual.
- sucesores, similar a la lista de predecesores pero para módulos a llamarse posteriormente.
- posicion, es la posición relativa de este módulo. Debe tener uno de los siguientes valores: APR_HOOK_FIRST, APR_HOOK_MIDDLE, APR_HOOK_LAST.

Sección 8 – Estructura de directivas para el módulo

La definición de nuevas directivas se lleva a cabo estableciendo los campos de la estructura *command_rec*, declarada en *http_config.h*. En esta sección se dispone de un arreglo de *command_rec*, así:

```
    command_rec comandos[]
```

Cada entrada del arreglo permite definir una nueva directiva para el módulo. Para ello es necesario indicar el nombre de la directiva, la función encargada de manejarla (definida en la sección 6), cuáles directivas deberían ser aplicadas para poder permitir el comando, la forma en que se interpretarán los argumentos de la directiva y un mensaje para mostrar en caso de error.

Sección 9 - Definición del módulo para configuración

La definición del módulo se hace llenando los campos de la variable *module mi_modulo* declarada en la sección 2, de esta forma:

```
    module AP_MODULE_DECLARE_DATA mi_modulo =
    {
        STANDARD20_MODULE_STUFF,
```

```
        crear_configuracion_de_directorio,
        fusionar_configs_de_directorio,
        crear_configuracion_de_servidor,
        fusionar_configs_de_servidor,
        registrar_comandos,
        registrar_hooks
}
```

Algunos aspectos de la implementación

- El módulo implementado interviene en las siguientes fases:
  - Creación de la configuración del servidor.
  - Fusión de las configuraciones de servidor.
  - Creación de configuraciones por directorio.
  - Fusión de configuraciones por directorio.
  - Fase *translate name*.
  - Fase *map to storage*.
  - Fase *create request config*.
  - Fase *content handler*
- La llamada a la función yyparse se realiza en la fase del *Content handler*. Es aquí donde se abre el archivo cuyo nombre fue traducido en *translate name* y *map to storage* para poder procesar el código embebido.
- Normalmente sólo se necesita realizar una llamada a la función yyparse para realizar toda la compilación del programa fuente. En nuestro caso, es posible tener varias secciones alternadas de código HTML con código de nuestro lenguaje de programación. Por ello, se hace necesario llamar a la función yyparse cada vez que se encuentre un nuevo bloque de código embebido. No es común realizar varias llamadas al yyparse, sin embargo, esto es viable debido a que todas las estructuras del *front end* permanecen intactas entre llamada y llamada.

RESUMEN Y CONCLUSIONES

Las dos versiones desarrolladas del intérprete: el módulo y la versión CGI proveen a los sitios web con contenido dinámico bajo la tecnología de Server-Side Scripting, permitiendo implementar páginas web como portales, foros de discusión, buscadores, etc.

La funcionalidad de ambas versiones es básica, aún así supera con creces el objetivo académico y puede ser usado como fuente de estudio y como base de futuras tesis. Éstas ofrecen un pequeño subconjunto del lenguaje Pascal en un entorno web, incluyendo instrucciones de control típicas como el if-then, if-then-else y case; instrucciones de iteración como el repeat, for y while; operaciones aritméticas y booleanas; uso de tipos de datos básicos como integer, real, boolean, string y el tipo de dato complejo array (arreglos); funciones y procedimientos predefinidos como write, writeln, inc y dec; y finalmente soporte de funciones y procedimientos definidos por el usuario hasta del tipo recursivo.

Siendo el intérprete la parte esencial del trabajo, la comunicación con el servidor HTTP presentó dos versiones distintas. La versión CGI del intérprete es la más simple de las dos debido a que el servidor Apache se encarga de todas las fases de atención, delegando al programa la única tarea de generar el contenido a enviar. Además la comunicación con el servidor es relativamente fácil, vía entrada y salida estándar y con variables de entorno.

Por otro lado, el módulo intérprete es algo más complejo. Depende en gran parte del APR (Apache Portable Runtime) lanzado recientemente, por ende, no existen muchos proyectos que lo utilicen. No obstante, se espera que con el transcurrir del tiempo sean más y más los proyectos que recurran al APR.

Este trabajo no hace uso de todo el potencial de las librerías APR, solamente se ha explotado una fracción de las mismas, en particular las relacionadas con los

resource pools, el manejo de cadenas, lectura y escritura de archivos y algunas nociones de compatibilidad.

El módulo intérprete interviene solamente en unas cuantas fases de atención de solicitudes, éstas son: translate_name, map_to_storage, create_request_config y la fase del content_handler. Trabajos futuros podrían dar mayor importancia a otras de fases como la autenticación de usuario, revisión de tipos, revisión de acceso, configuraciones por directorio, soporte de sesiones y incluso filtrado de contenido.

La mayor desventaja del módulo intérprete es que se basa en la carga dinámica de código, con DSOs en plataformas Unix y derivados y DLLs en Windows. El sistema operativo debe soportar este tipo de enlace y carga dinámica de código, de lo contrario el módulo se vuelve obsoleto.

La versión CGI no tiene este problema, puesto que es un programa externo y el único requisito es tener un compilador de C disponible en la plataforma en el que se desee usar.

La mayor desventaja del módulo es probablemente también su mayor atractivo. La capacidad de cargar código dinámicamente permite extender fácilmente la funcionalidad del servidor Apache con módulos hechos por terceros, sin tener que recompilar todo el servidor. Como mencionamos anteriormente, el responsable de la carga de otros módulos en el servidor Apache es el módulo mod_so. El problema ahora es que éste es uno de los dos módulos que tienen que ser agregados estáticamente, es decir, recompilando el servidor y habilitándolo explícitamente.

La tecnología CGI (Common Gateway Interface) aún está vigente. No por nada, lenguajes de Scripting bastante populares como Perl y PHP ofrecen las dos

versiones de sus intérpretes: aquella que distribuyen como módulo integrado al Apache y, aparte, la versión independiente ejecutable vía CGI.

.

Al igual que el módulo, el compilador puede mejorarse notablemente. Algunas de las funcionalidades que podrían agregarse son:

- Manejo de registros.
- Lectura y escritura de archivos.
- Funciones de acceso a bases de datos populares como MySQL, PostGRES, Oracle, etc.
- Manejo de imágenes y gráficos.
- Soporte de objetos.
- Optimización del código intermedio generado.

Una optimización importante que podría hacerse en el intérprete es separar la fase de compilación de la fase de ejecución, guardando el código intermedio generado para luego sólo cargarlo y ejecutarlo, tal y como lo hacen los JSPs. Esto mejoraría los tiempos de atención, considerando que resulta más rápido ejecutar un programa compilado que interpretarlo directamente.

Finalmente, el presente trabajo sienta las bases para futuras investigaciones tanto en el área de compiladores e intérpretes como en las hasta ahora poco exploradas librerías APR.

EJEMPLOS

Ejemplo 1: Impresión de tabla con el módulo intérprete vía método GET.

```
<html>
	<head><title>Página de ejemplo vía método GET del módulo PZK</title></head>
	<body bgcolor="yellow" text="blue" link="cyan" vlink="red">
		<form action="mod-get-res.pzk" method="GET" enctype="application/x-www-form-urlencoded">
			<h3>
				<br>
				<br>
				<~
					write('<div align="center">');
					write('<img src="bigPUCP.gif" width="114" height="114" border="0" alt="">');
					write('</div><br>');
				~>
				Este ejemplo imprime una tabla con las dimensiones ingresadas como datos<br><br>

				<div align="center">
					<table>
						<tr height="50">
							<td>Filas (<=10):</td>
							<td><input type="text" name="f" size="30" maxlength="30" ></td>
						</tr>
						<tr height="50">
							<td>Columnas (<=10):</td>
							<td><input type="text" name="c" size="30" maxlength="30"></td>
							<td><input name="send" value="send" type="submit"></td>
						</tr>
					</table>
				</div>
			</h3>
```

```
            </form>
        </body>

</html>
```

Ingreso de datos

```
<html>
        <head>
        <title>
                Página producida por el módulo PZK vía método GET
        </title>
        </head>
        <body bgcolor="yellow" text="blue" link="cyan" vlink="red">
                <div align="center">
                        <h1> PUCP <br></h1>
                        <img src="bigPUCP.gif" width="114" height="114" border="0" alt="">

                        <~
                                const N = 10;
                                var a:array[N,N] of integer;
                                var cont,filas,cols:integer;
                                var i,j:integer;
                                cast(filas,f);
                                cast(cols,c);
                                cont:=1;
                        ~>

                        <h2>    Ingeniería Informática <br><br> </h2>

                        <~
                                writeln('<table width="800" border="5">');
                                writeln('Dato "Filas" leido:',f,'<br>');
                                writeln('Dato "Columnas" leido:',c,'<br>');

                                for i:=1 to filas do
                                        for j:=1 to cols do begin
                                                a[i,j]:=cont;
```

```
                                        inc(cont);
                                end;
                        ~>

                        Tabla generada<br>
                        <h3>
                        <~
                                for i:=1 to filas do begin
                                        write('<tr>');
                                        for j:=1 to cols do
                                                write('<td width="100">',a[i,j],'</td>');
                                        write('</tr>');
                                end;
                        ~>

                        </table>
                        Fin de ejemplo <br>
                        <A href="main.pzk">Regresar a página de inicio</A>
                        </h3>
                </div>
        </body>
</html>
```

Página generada con los datos ingresados

Ejemplo 2: Cálculo de la función de Fibonacci con la versión CGI del intérprete y vía método POST.

```
#!/var/www/pzk
<html>
        <head><title>Página de ejemplo vía método POST del programa PZK</title></head>
        <body bgcolor="yellow" text="blue" link="cyan" vlink="red">
                <form action="cgi-post-res.pcgi" method="POST" enctype="application/x-www-form-
urlencoded">
                        <h2>
                                <br>
                                <br>
                                Este ejemplo calcula el número fibonacci del dato ingresado <br>
```

```
                    </h2>
                    <h3>
                        <div align="center">
                            Dato:<input type="text" name="n" size="30"
maxlength="30">
                            <input name="send" value="send" type="submit">
                        </div>
                    </h3>
                </form>
            </body>

</html>
```

Ingreso de datos

```
#!/var/www/pzk
<html>
    <head><title>Página producida por el programa PZK vía método POST </title></head>
    <body bgcolor="yellow" text="blue" link="cyan" vlink="red">

        <div align="center">
            <h1>
                PUCP <br><br>
            <h1>

            <img src="bigPUCP.gif" width="114" height="114" border="0" alt="">


                <~
                    function fibo(var n:integer):integer;
                    begin
                        if (n<=1) then
                            fibo:=1
                        else
                            fibo:=fibo(n-2)+fibo(n-1);
                    end;
```

```
                ~>

        <h2>    <br><br>Ingeniería Informática <br><br> </h2>

        <h3>
                <~
                        var nint,res:integer;
                        if (not isnull(n)) then begin
                                cast(nint,n);
                                res:=fibo(nint);
                                writeln('Fibonacci(',nint,')=',res);
                                write('<br> <br>');
                        end else writeln('Ingrese nuevamente el dato');
                ~>

        Fin de ejemplo <br><br>

        <A href="main.pzk">Regresar a página de inicio</A>
        </h3>
    </div>
</body>
</html>
```

Página generada con los datos ingresados

GLOSARIO DE TÉRMINOS

La Internet

La Internet es la red de computadoras interconectadas accesibles globalmente que transfieren datos mediante intercambio de paquetes por medio de un Protocolo de Internet estandarizado (IP, Internet Protocol). Está compuesta de otras miles de redes más pequeñas: comerciales, académicas, gubernamentales, organizaciones entre otras. La Internet sirve de transporte a varios medios de información y servicios tales como correo electrónico, servicio de mensajería instantánea (chat), sitios web y otros documentos disponibles por la WWW. Debido a que éste es de lejos, el internet (con "i" minúscula) más extenso del mundo, se le llama simplemente La Internet (con "I" mayúscula).

La WWW

La World Wide Web, o simplemente "La Web", es un repositorio de información accesible por red, compuesta de software y de una serie de protocolos y convenciones. Gracias al hiper-texto (texto con enlaces) y a una serie de técnicas multimedia, cualquier persona es capaz de realizar búsquedas, de diversos tipos de documentos, mediante un visualizador o navegador.

Hipertexto

Texto con enlaces, parecido a las notas de pie de página y a las notas del autor en los libros convencionales. La computadora puede seguir los enlaces de hipertexto permitiendo a la persona escapar del esquema secuencial de lectura, haciéndolo tan fácil como cambiar de página en un libro.

Los documentos a los cuales se tienen acceso en la Web se conocen como páginas web, documentos de tipos variados e incluso de mezclas de tipos (entre texto, imágenes y sonido). Estos documentos son escritos bajo un formato especial llamado HTML (HiperText Markup Language ó Lenguaje Marcador de Hipertexto) haciendo posible su publicación en la Web.

De acuerdo a su contenido, las páginas web se pueden clasificar en dos categorías:

- Páginas Web con contenido estático o páginas estáticas
- Páginas Web con contenido dinámico o páginas dinámicas

Una página web presenta contenido estático cuando siempre devuelve el mismo contenido llamada tras llamada, sin variar nunca. En cambio, una página web presenta contenido dinámico cuando éste varía dependiendo de ciertos parámetros. Entonces decimos que la página es dinámica.

Las páginas estáticas se escriben en código HTML puro. Por otro lado, para generar páginas dinámicas se usan distintas técnicas como por ejemplo CGI (Common Gateway Interface, programas externos que se ejecutan por medio del servidor web, recibiendo datos de él y generando salida) o motores que interpretan código embebido de algún lenguaje de programación dentro del código HTML (técnica conocida como Server-Side Scripting). Es éste

URIs y URLs

Uniform Resource Identifier (URI), es un elemento del protocolo de internet que consiste en una cadena de caracteres conformados bajo una sintaxis en particular. La cadena refiere a un nombre o a una dirección que pueden ser usados para referir tanto a recursos abstractos como físicos. Los URLs (*Uniform Resource Locators*), también conocidos como direcciones web, son direcciones estandarizadas de algún recurso de la Internet.

BIBLIOGRAFIA

- Alfred V.Aho, Ravi SEIT y Jeffrey D.Ullman [1988]. “Compilers Principles, Techniques and Tools”. Addison Wesley.

- Thomas Cormen, Charles Leiserson y Ronald Rivest [2000]. “Introduction to Algorithms”. The MIT Press.

- Nicklaus Wirth [1976]. “Algorithms + Data Structures = Programs”. Prentice Hall.

- Brian Kernighan & Dennos Ritchie [1985]. “El lenguaje de programación C”. Prentice Hall.

- Leon Atkinson & Zeev Suraski [2004]. “Core PHP programming”. Prentice Hall.

- Falkner, Galbraith, Irani, Kochmer, Panduranga, Perrumal, Timney, Kunnumpurath [2001]. “Beginning JSP Web Development”. Wrox Press.

- J. Glenn Brookshear [1993]. “Teoría de la Computación. Lenguajes formales, autómatas y complejidad”. Addison-Wesley Iberoamericana.

- Dean Kelley [1995]. “Teoría de Autómatas y Lenguajes Formales”. Prentice Hall.

- Ben Laurie & Peter Laurie [1999]. “Apache: The Definitive Guide”. O’Reilly Books.

- Lincoln Stein and Doug MacEachern [1999]. “Writing Apache Modules with Perl and C”. O’Reilly Books.

- http://www.gnu.org/software/bison/bison.html
- http://www.apache.org/
- http://apr.apache.org/
- http://httpd.apache.org/