CGILua: A Multi-Paradigmatic Tool for Creating Dynamic WWW Pages

Anna M. Hester Renato Borges
Roberto Ierusalimschy
TeCGraf, Departamento de Informática, PUC-Rio
{anna,rborges,roberto}@tecgraf.puc-rio.br

Abstract

The dramatic growth of the Internet and the World Wide Web creates great demand for tools to support the construction and maintenance of WWW sites. CGILua intends to simplify the task of creating dynamic Web pages, supporting three different paradigms for describing dynamic Web pages: Programming, templates, and database. Besides fully supporting these three paradigms, the user can also freely mix them, allowing the use of CGILua in new interesting ways, even in the description of static pages. CGILua is fully implemented, and is already being used in industrial sites.
Keywords: HTML, WWW, CGI, dynamic pages

Resumo

O enorme crescimento da Internet e do WWW cria uma grande demanda para ferramentas de suporte à construção e manutenção de "sites" WWW. CGILua foi desenvolvido visando simplificar a tarefa de criação de páginas Web dinâmicas, oferecendo suporte a três diferentes paradigmas: programação, modelos ("templates") e bancos de dados. Além de suportar integralmente os três paradigmas, o usuário pode combiná-los livremente, permitindo o uso de CGILua de novas e interessantes maneiras, mesmo para descrição de páginas estáticas. CGILua está completamente implementado, e está sendo usado em diversos "sites" comerciais.
Palavras-chave: HTML, WWW, CGI, páginas dinâmicas

1 Introduction

Due to the growth of the Internet and the World Wide Web there is an increasing demand for the use of WWW as a dynamic media. Two aspects should be considered to achieve this goal: Web pages should be created based on real time information, and there must be conditions for the construction and maintenance of the Web sites in a practical and effective way.

Many tools have been developed to fulfill these two requirements. A strategy is to take advantage of the fact that a Web server can execute a program as a response to a Web resource request. This program can then create a *dynamic page*, based on the information acquired on execution time.

In order to receive and send data to the Web server, such programs use a communication protocol. Among these protocols, the most widely used is CGI (Common Gateway Interface) [2]. For this reason, programs activated by the server are also called *CGI applications*.

Most tools for creating CGI applications can be classified in three broad paradigms:

programming The more "traditional" way for building CGI applications is using a full-fledged programming language, typically C or Perl, sometimes with the help of a specific library, such as CGI.pm [16] and libcgi [19]. The main advantage of this approach is its expressiveness. Also, some programmers find it convenient because they can still use a conventional programming style. Nevertheless, this approach is quite difficult for non-programmers, and even for programmers it is not very effective, since it operates on a very low abstraction level.

templates This class includes tools such as Web* [11]. In this paradigm, a dynamic page is always based on a static version of the page, the *template* (or *outline*). Usually, the template is written directly in HTML, with "escapes" to mark fields to be filled in. When the page is accessed, the template feeds a pre-processor that creates the final page.

The main advantage of this approach is that it allows the use of conventional HTML editors, such as Microsoft's Front Page, for building the template, requiring no programming knowledge. There are some systems based on templates that use an embedded programming language (like Tcl [15]) to describe how to fill in each field. However, the paradigm imposes severe limits on what can be done with the language, since it must be used *within* the template. For instance, it is impossible to use the conditional mechanism of the language to select among different templates based on some condition.

data bases This paradigm, best illustrated by Microsoft's IDC [4], sees accesses to a server as queries to a data base. A page is described mainly by the SQL statement to perform the query. It also uses templates, but only to specify how to show a query result.

The main advantage of this approach is its high level of abstraction, which greatly simplifies the creation of pages, as long as they are in fact answers to database queries. An interesting feature of these systems, when compared to simple templates, is their ability to repeat a piece of a template, in order to show multiple results of a query. Therefore, a fixed template is able to show variable size information.

In this paper, we present CGILua, another system for developing CGI applications. The main novelty of CGILua is that it fully integrates the three paradigms in a single system. Besides supporting each individual paradigm style, users can combine the paradigms. This allows the use of CGILua in other useful ways. For instance, templates can be used inside other scripts to describe common patterns shared by many different pages.

CGILua uses the extension language Lua [6] both in its implementation, as a configuration language for the whole package, and as the scripting language for writing CGI pages. Therefore, CGILua inherits most of Lua features: Portability, a simple Pascal-like syntax, small size, and flexibility [10]. Most of its flexibility is due to its architecture, based on the use of an extension language [3]. A *kernel*, written in C, provides basic generic services. Upon it, a *configuration part*, written in Lua, gives the program its final shape. With this architecture, many characteristics, like security policies and the inclusion of extra libraries, can be easily tailored in each individual site.

CGILua is fully implemented, and is being used in real applications since 1996. Its documentation and code can be accessed at

`http://www.tecgraf.puc-rio.br/manuais/cgilua`

The next section describes how CGILua supports each individual paradigm presented above, and how they can be merged. The architecture of CGILua is described in Section 3, and Section 4 compares CGILua with other proposals. The last section is reserved for some final remarks; it briefly presents some current commercial uses, portability issues, and other features of CGILua.

2 An Overview of CGILua

This section shows the use of CGILua using the three paradigms described above: Programming, templates, and database.

Following the first paradigm, a dynamic WWW page is simply a program; when the page is accessed, the program is ran and its output is interpreted as the final HTML page sent to the browser. In CGILua, these programs are written in Lua. Figure 1 illustrates this paradigm with a simple script, which writes the Collatz sequence of a given number. Notice that all form data is previously decoded by CGILua and stored in a table called `cgi`. Therefore, the expression `cgi.number` results in the field `number` given to the script.

As already stated, the main advantage of this paradigm is its power. The full flexibility of Lua is available in the creation of a page. That includes all abstraction facilities of a programming language, plus pre-defined functions for pattern-matching and the like.

The second paradigm considers a script as a template: An HTML document where special marks indicate fields to be handled by the preprocessor. CGILua supports three kinds of fields: *Statement* fields, *expression* fields, and *control* fields. Statement fields contain Lua statements to be executed by the preprocessor; they generate no implicit output, although they can explicitly write anything to the final page. Such fields are written between the marks `<!--$$` and `$$-->`. Expression fields contain Lua expressions, which are evaluated by the preprocessor, with the result used as the final text of the field. Such fields are written between the marks `$|` and `|$`. Finally, control fields indicate parts of the document to be repeated or conditionally inserted. All these kinds of fields are

```
-- defines function coll (this line is a comment)
function coll (x)
  if mod(x,2) == 0 then return x/2
  else return 3*x+1 end
end

write("Content-type: text/html\n\n")
write("<html><head><title>Collatz Sequence</title></head><body>")
write("<h1>The number you have chosen is: ", cgi.number, "</h1>")
n = cgi.number;
while n ~= 1 do write(n, "<br>"); n=coll(n) end
write("</body></html>" )
```

Figure 1: A simple CGILua script

shown in Figure 2, which again describes a page to show the Collatz sequence of a given number.

The LOOP construct acts as a C for statement: It repeats all the text between it and the matching ENDLOOP. The fields start, test and action contain the Lua code that controls the loop. Loop constructs can be freely nested in a template, whenever more complex structures are needed.

It is interesting to notice that all marks have been carefully chosen so that a template has a sensible appearance in a browser even when it is not preprocessed. Statement and control marks, which do not generate any implicit output, are handled as comments by HTML syntax, while expression marks appear literally in the browser, acting as a place-holder. In this way, a template can be edited as a regular static HTML page.

In the last paradigm, a dynamic access is mainly a database query. This paradigm, in CGILua, is quite similar to templates, but uses functions to access a database. These functions are simply Lua functions provided by a database library. Figure 3 is an example of a page created from database access. Function DBOpen establishes a connection with a database, function DBExec executes an SQL statement, and function DBRow traverses the resulting table. Each row is returned as a Lua table, which is then stored in variable m. It is interesting to notice in this example the interaction between the control marks and the HTML marks to format a table. The final result of this template, after preprocessing, is shown in Figure 5.

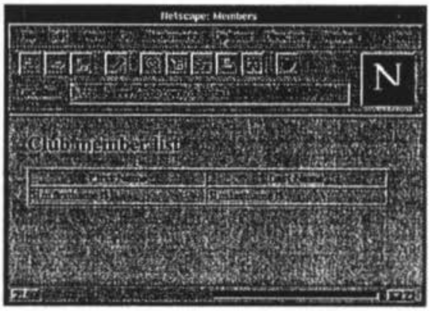
Traditionally, templates are used for more declarative, static uses, while programming is used when there is a need for control structures and dynamic descriptions. CGILua allows a reverse in this conventional use: A template can be used as a kind of subroutine, while Lua is used as the declarative language. Figures 6 and 7 illustrate this style. Function cgilua.preprocess provides a degree of reflexivity: It allows a Lua script to explicitly process a template, as if the user had accessed that template. Notice that the file form.html describes how to show a generic form. The abstract specification of the form, on the other hand, is given in script form.lua, in the format of a table (field). The same table (field), which gives the abstract specification of the form, can be used

```
<html>
<head><title>Collatz Sequence</title></head>
<body>
  <!--$$
      -- defines function coll
      function coll (x)
        if mod(x,2) == 0 then return x/2
        else return 3*x+1 end
      end
  $$-->
  <h1>The number you have chosen is: $| cgi.number |$ </h1>
  <!--$$ LOOP start="n=cgi.number", test="n ~= 1", action="n=coll(n)" $$-->
     $| n |$<br>
  <!--$$ ENDLOOP $$-->
</body></html>
```

Figure 2: A CGILua template

```
<html>
<head><title>Members</title></head>
<body>
  <h1>Club member list</h1>
  <!--$$ DBOpen( "DSN=club;" ) $$-->
  <table border=1 width=100%>
    <tr align=center>
      <td><strong>First Name</strong></td>
      <td><strong>Last Name</strong></td>
    </tr>
  <!--$$ LOOP
      start="DBExec('SELECT firstname,lastname FROM Members'), m = DBRow()",
      test="m ~= nil",
      action="m = DBRow()" $$-->
    <tr>
      <td>$| m.firstname |$</td>
      <td>$| m.lastname |$</td>
    </tr>
  <!--$$ ENDLOOP $$-->
  </table>
</body></html>
```

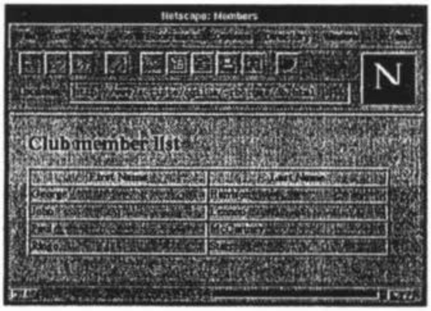
Figure 3: A database query in CGILua



Club member list

First Name	Last Name
[illegible]	[illegible]

Figure 4: Template without preprocessing



Club member list

First Name	Last Name
George	Harrison
John	Lennon
Paul	McCartney
Ringo	Starr

Figure 5: Final result of the template

```
<html>
<head><title>Example Form</title></head>
<body>
  <form method="POST" action="validate.lua">
  <!--$$ LOOP start="i=1", test="field[i]", action="i=i+1" $$-->
    $|field[i].label|$:
    <input type="text"
           name="$|field[i].name|$"
           value="$|cgi[field[i].name]|$"><br>
  <!--$$ ENDLOOP $$-->
    <input type=submit>
  </form>
  <font color=#ff0000>$|error_message|$</font>
</body></html>
```

Figure 6: File form.html

```
field = {
           { name="project",    label="Project" },
           { name="year",       label="Base year" },
           { name="code",       label="Project code" }
         }
cgilua.preprocess("form.html")
```

Figure 7: File form.lua

```
field = {
          { name="project",   label="Project" },
          { name="year",      label="Base year" },
          { name="code",      label="Project code" }
        }
i = 1
while field[i] and not fail do
  if cgi[field[i].name] == "" then
    fail = 1
  end
  i = i + 1
end
if fail then
  error_message = "Please fill out all the fields."
  cgilua.preprocess("form.html")
else
  cgilua.preprocess("list-db.html")
end
```

Figure 8: Validating data from a form

to drive the creation of other pages; Figure 8 shows a script that validates the data from such a form. Clearly, in a real case, table `field` would have a single definition shared between both scripts.

Notice how CGILua can improve the management of Web sites even when used for static pages, since the use of parametric pages allows a developer to work in a higher abstraction level. For instance, the same template shown in Figure 6 can be used to create many different forms, when fed with different values for table `field`.

3 The Architecture

Like many programs that use a scripting language, CGILua has two main modules: A *kernel*, written in C, and a *configuration script*, written in Lua. The kernel is the program called by the CGI server when a user accesses a CGILua page. It creates a Lua environment, defines some new functions to Lua and then runs the configuration script. This script decodes the data in the query, redefines some Lua functions to provide a secure environment where the user script will run, locates the user script and then runs it. Notice that, as all these steps are done by a script, they can be easily adapted to local needs by the system administrator. Also, a site may have several configuration scripts, allowing differentiated environments for different projects. For instance, institutional pages may have a weaker security policy than the one enforced on personal user pages.

Figure 9 shows how a client request is processed, since its arrival at the server until its final result. The request URL always has a virtual path that points to the script to be ran. Eventually, the request may also contain codified data, for instance from form

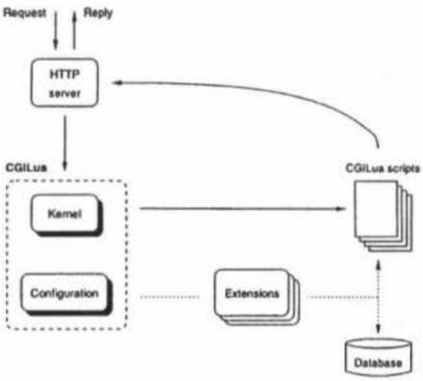


Figure 9: CGILua architecture

contents. CGILua identifies the script type — a Lua program or an HTML template — based on the path extension. If it is a Lua program, CGILua simply runs it. Otherwise, the configuration script calls the preprocessor to run the script as a template. The preprocessor itself is defined in the configuration script, and is completely written in Lua. This fact enhances the reflexivity of the tool, since the preprocessor can be called not only by the kernel, but at any moment by the script, as shown in Section 2.

CGILua also offers a debugger to help the development of scripts. When in debugging mode, the debugger acts as the HTTP server, running the script and interacting with it through CGI. It also enables all usual debugger facilities offered by the Lua debugger [8], such as step by step execution, inspection of the execution stack, etc.

Security Issues

A CGI script has the same security problems of any network server [7], since it is invoked by remote requests; from this point of view, any CGI script can be considered a mini-server. Since CGILua activates user's Lua scripts, all security concerns must be extended to these scripts too.

Most software developers do not have the know-how to implement secure servers in a public network. Security holes may occur when the program accesses non initialized memory, runs other programs, accesses files, etc. Most of these problems happen when the server receives unanticipated wrong data: An apparently inoffensive server can allow the execution of almost any command by an intruder, in such faulty conditions. A typical example is a simple piece of program that receives an e-mail address from an HTML form and uses it to send a message, using the C function popen:

```
sprintf(cmd, "mail %s", user);
file = popen(cmd);
/* writes message to file */
 ...
```

This code works without problems as long as the user string, supplied by an external client, is an e-mail address. However, this string can be used to execute any Unix command. For instance, if the supplied "e-mail address" is the string `"& cd /; rm -fr *"`, the server script will attempt to erase all server files; the string `"& xterm -display intruder.com:0.0"` will open a terminal with all rights of the server script at the intruder console.

Lua is a language with a secure semantics. There are no language constructions with undefined behavior. Lua programs are translated into byte-codes, which are then interpreted in a protected environment. There are no instructions to do real memory access or to call arbitrary C functions; the stack is fully controlled. Besides pure resource consumption, the only way a Lua program interacts with the external environment is through function calls. Therefore, in the realm of a Lua program, security issues can be focused on how to control the use of insecure functions.

In Lua, functions are first class values; Lua programs can freely create, redefine or erase functions at run time. Therefore, a simple solution for a secure environment would be the configuration script to erase all "dangerous" functions before calling the user script. That solution is clearly too much restrictive, since most of these functions can be used in restricted ways without security holes. For instance, a generic open function, which allows a script to write to any file, may be dangerous, but it could be restricted to only open files in a pre-defined directory sub-tree.

The solution adopted by CGILua has two parts. The kernel introduces a single generic facility that allows a Lua script to erase a global function while keeping a private access to it. With these facilities, the configuration script redefines the "dangerous" functions to more secure versions. Notice that the original functions are still accessible by the configuration script, and are used in the implementation of the restricted versions. With this solution, the whole Lua environment is configured in Lua itself, with the usual benefit: Flexibility. System administrators can change the configuration script to adapt the protected environment to their specific needs.

Notice how this solution differs from the one adopted by Perl, which uses a modified interpreter, called *tainted Perl*. In this mode, the interpreter forbids the execution of any command dependent of external data. With this approach, despite its complexity, the security level is fixed and "wired" in the language implementation.

Extensibility

As already explained, the configuration file can define new Lua functions. In this way, Lua libraries can be automatically loaded before the execution of the user scripts, offering new facilities. Sometimes, however, such extension should be written in C, either for efficiency reasons, like a cryptography package, or because it accesses pre-defined C interfaces, like a database.

Again, the solution adopted by CGILua has the same general pattern: The kernel implements a generic mechanism for dynamic library loading, and the configuration script specifies which and how each package will be loaded. After this step, the script erases these loading facilities, thereby restricting the use of any unauthorized extension.

The main example of use of this facility is the database package. Lua itself offers no database facilities. Its standard libraries offer only access to files in conformance to the ANSI C facilities. DBLua is a Lua library that interfaces Lua with a standard database API, called DBGraf [13], which offers access to different database systems, like mini-SQL and ODBC. This library is dynamically loaded by the configuration script, therefore offering all database facilities of CGILua.

4 Related Work

This section compares CGILua with other packages for building dynamic pages. These packages are presented according to the adopted paradigms.

Many available tools for creating dynamic Web pages are based on the programming paradigm. Several publications [14, 5, 9] assume that C and Perl are the most widely used languages for this task. C, like most "general purpose" programming languages, is very low level. Mainly, C lacks automatic memory management, and particularly lacks support for dynamic strings. This not only complicates string manipulation, a most common operation in Web scripts, but also opens the possibility of memory overflow, which can lead to security holes. Moreover, C is a compiled language, and the compilation time can have a negative impact in the development of such small programs. Most scripts go through many small changes until their final look. On the other hand, the efficiency of a compiled language is hardly relevant in a page access, since the whole access time includes network communication, process creation, program load, file access, etc. The time to execute the script itself is usually much smaller than this total time.

For those reasons, several interpreted languages has been used for the development of CGI scripts [12, 18]. Among them, Perl seems to be the most widely used. Perl is an interpreted language, with strong string support and pattern-matching facilities. So is Lua. Perl is certainly more powerful than Lua, in the sense that it has a much bigger built-in library. On the other hand, Lua is much simpler than Perl: It has a Pascal-like syntax, and a simple formal semantics. Its executable file is almost ten times smaller than Perl's [10]. It is not difficult to write libraries for Lua, both in C or in Lua, and CGILua can transparently load these libraries, if required. Moreover, its reflexive facilities allow easy configuration of what is available to the final scripts, improving their security level.

Although it is not uncommon to use the bare language when programming CGI scripts, many developers use specific libraries for the task. CGI.pm [16, 17] is a package for Perl 5. The package offers facilities for parsing form parameters, generation of headers and creation of forms.

CGI.pm main features are its functions for form design and management, which encapsulate the creation of form elements in HTML. It can also automatically reinitialize a form with the data from the previous interaction, a useful feature in error situations or data checking. CGILua also parse form parameters, in a way completely transparent to the script. As we have seen in Section 2, field values are directly accessible as fields of a dynamic "record" called `cgi`. CGILua has a weaker set of functions for creating HTML elements. In fact, these functions are seldom used, since most CGILua users prefer to use templates when more complex elements are involved in a page.

Libcgi [19] is a C library for the description of CGI pages. It also offers facilities for parsing form data, and for building HTML elements. Despite its features, it cannot counterbalance the weakness of C for the task. A script still has to handle memory allocation, data conversion, and other details. Moreover, there is the overhead of compilation-link phase while developing a page.

Web* [11] is a tool that adopts the template paradigm. Their templates, called "layout pages", are essentially an HTML document with Tcl code embedded in it. This Tcl code is always handled as expressions; there is no support for control fields, like in CGILua. As in CGILua, form data are automatically available to a script through Tcl variables, and there is support to save part of the script state from one page to the next. Web* offers support for neither database access, nor for writing CGI scripts outside templates. An interesting feature of Web* is its support for CORBA access. A similar feature is available in CGILua, using the LuaOrb package [1].

Microsoft's IDC [4] (IDC stands for *Internet Database Connector*) is a typical tool using the database paradigm. Its main advantage is its high abstraction level. It uses two files to describe a script: One describes, in SQL, the query to be executed when the script is accessed. The other file is an HTML template, to be filled by the SQL results. The lack of a full-fledged programming language imposes limits to its flexibility. For instance, loop fields in a template can only be used to show query results; the template fields must be filled with the bare results of the query, without further processing. IDC is non portable, being available only on Microsoft Windows platform.

5 Final Remarks

CGILua has been described, as a tool for building dynamic Web pages. CGILua allows a user to develop its pages using three different paradigms: Programming, templates, and databases. Moreover, it fully integrates them, allowing the creation of new development styles with a mix of mechanisms from these paradigms.

Besides the support of multiple paradigms, CGILua also presents the following features:

flexibility The use of an extension language in the architecture of CGILua makes the tool highly flexible. Many characteristics, from error handling to security policies, can be easily tailored by the system administrator. Both Lua and C libraries can be dynamically loaded, in order to offer new functionality, like cryptography.

simplicity The whole system has 1500 lines of code. All its sources and binaries can be put in a single floppy disk. Its use is also simple. Most users are able to start using CGILua in less than half an hour. Templates are written mostly in HTML, database accesses are written in SQL, and programs are written in Lua. Lua is a small language, with a simple Pascal-like syntax, simple semantics, and automatic memory management.

scalability With the loop constructor, fixed templates can be used for variable size information. Moreover, a judicious choice of syntax allows the use of conventional HTML editors to manipulate these templates.

portability CGILua runs on Windows NT, Windows 95, Linux, IRIX, Sun-OS, Solaris, AIX, HP-UX, FreeBSD, Unixware, SCO, OSF, and other platforms with essentially the same source code. Applications are fully portable: Any script written in one platform needs no changes to run in other system.

CGILua is fully implemented, and currently is being used for several industrial applications. Below is a list of its main applications.

SIGMA is a WWW system being developed to PETROBRAS (The Brazilian Petroleum Company). Its purpose is to manage the procedure of obtaining environmental licenses and to inform about rules and technical procedures. The system generates and collects information through an active communication with a database. A group of six people has been developing the system, having produced, to this date, around ten thousand lines of HTML templates and Lua code. The system is part of a strategy to obtain the ISO 14000 certificate. Around one hundred people will use SIGMA as a work tool daily. Moreover, parts of the system will be available to the general public.

CGILua has also being used to build RPA, an Intranet system used by PUC-Rio (The Pontifical Catholic University of Rio de Janeiro) to keep track of the academic production of its faculty. Currently the system is being used by more than one hundred faculty members and staff, from eight different departments.

Finally, Medialab, a Brazilian company that develops WWW sites for industries like Shell and Microsoft, has been using CGILua for the creation of dynamic pages in eleven sites, among them the site of Volkswagen do Brasil and Dutyfree.

Acknowledgements

The authors would like to thank André Clinio, who helped the development of the very first version of the kernel; André Carregal, who first suggested the idea of control fields in templates; and Mônica Leitão, for pioneering the use of CGILua in real applications. This work has been partially supported by CNPq and MCT.

References

[1] R. Cerqueira, N. Rodriguez, and R. Ierusalimschy. Using Lua to access CORBA objects. Monografias em ciência da computação, PUC-Rio, Rio de Janeiro, Brazil, 1997.

[2] CGI - Common Gateway Interface. W3C - World Wide Web Consortium, URL: http://www.w3.org/pub/WWW/CGI/, 1996.

[3] D. Cowan, R. Ierusalimschy, and T. Stepien. Programming environments for end-users. In *12th World Computer Congress*, volume 3, pages 54–60, Madrid, Sep 1992. IFIP.

[4] C. Doyle, editor. *Microsoft Windows NT Server Internet Guide.* Microsoft Press, 1996.

[5] M. Erwin, J. Dwight, et al. *Special Edition Using CGI.* QUE, April 1996.

[6] L. H. Figueiredo, R. Ierusalimschy, and W. Celes. Lua—an extensible embedded language. *Dr. Dobb's Journal*, 21(12):26–33, 1996.

[7] S. Garfinkel and G. Spafford. *Practical UNIX & Internet Security*. O'Reilly & Associates, Inc., second edition, 1996.

[8] T. G. Gorham and R. Ierusalimschy. Um sistema de depuração reflexivo para uma linguagem de extensão. In Roberto Bigonha, editor, *I Simpósio Brasileiro de Linguagens de Programação*, pages 103–114, Belo Horizonte, September 1996.

[9] S. Gundavaram. *CGI Programming on the World Wide Web*. O'Reilly & Associates, Inc., 1996.

[10] R. Ierusalimschy, L. H. Figueiredo, and W. Celes. Lua—an extensible extension language. *Software: Practice & Experience*, 26(6):635–652, 1996.

[11] V. Jagannathan, G. Almasi, and A. Suvaiala. Collaborative infrastructures using the WWW and CORBA-based environments. In *Proceedings of the IEEE Fifth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises June 19-21, 1996, Stanford, CA*. IEEE Computer Society Press, 1996.

[12] D. Libes. Writing CGI scripts in Tcl. In *Tcl 96 Conference*, May 1996.

[13] M. Mediano. *DBGraf - Manual de Referência*. TeCGraf, May 1996.

[14] R. J. Mudry. *Serving the Web*. Coriolis Groups Books, 1995.

[15] J. Ousterhout. Tcl: an embeddable command language. In *Proceedings of the 1990 Winter USENIX Conference*. USENIX Association, 1990.

[16] L. Stein. CGI.pm - a Perl 5 CGI library. URL: http://www.genome.wi.mit.edu/ftp/distribution/software/WWW/, April 1997.

[17] L. Stein. A Perl library for writing CGI scripts. *Web Techniques*, 2(2), February 1997.

[18] M. Vanaken. Writing CGI scripts in Python. *Linux Journal*, 34, February 1997.

[19] J. Weber. libcgi. URL: http://wsk.eit.com/wsk/, 1996.

[20] Yahoo! URL: http://www.yahoo.com/Computers_and_Internet/Internet/World_Wide_Web/CGI__Common_Gateway_Interface/, April 1997.