Journal of Universal Computer Science, vol. 4, no. 9 (1998), 748-762
submitted: 1/9/98, accepted: 14/9/98, appeared: 28/9/98 © Springer Pub. Co.

Building Flexible and Extensible Web Applications with Lua[1]

Anna Hester
(Catholic University of Rio de Janeiro - PUC-Rio, Brazil
anna@tecgraf.puc-rio.br)

Renato Borges
(Catholic University of Rio de Janeiro - PUC-Rio, Brazil
rborges@tecgraf.puc-rio.br)

Roberto Ierusalimschy
(Catholic University of Rio de Janeiro - PUC-Rio, Brazil
roberto@inf.puc-rio.br)

Abstract: The World Wide Web is in constant renovation, with new technologies emerging every day. Most of these technologies are still incipient, and there are few de facto standards for this "new Web". There is a need for tools that can run with current standard support, but which are flexible and extensible enough to be eventually ported to new APIs and to incorporate new technologies. On the other hand, many Web developers cannot keep pace with the fast track of Web technologies. Therefore, it is important for new tools to be simple enough to be mastered quickly by the average programmer. This paper presents CGILua, a Web development tool that matches these requirements. The paper also discusses why this tool is being adopted in many commercial and academic projects, focusing on issues such as flexibility, extensibility, simplicity, and portability.

Key Words: scripting languages, CGI, Web dynamic pages
Category: D.2, H.5

1 Introduction

The World Wide Web has gone through a big change from its initial goal. The electronic publication of static, read-only and file-based documents is being replaced by a more complex environment, where dynamic and interactive pages are produced by components of a distributed system. In this setting, some of its adopted technologies cannot satisfy the new requirements, driving the search for many alternative technologies.

Unfortunately, embracing any new technology at this time may be premature, since there are no de facto standards for this "new Web". It is well recognized that the Web standards are key factors in its success [Hadjiefthymiades and Martakos 97]. Currently, many new proposals are tied to a specific vendor or operating system, compromising the openness that allowed the Web to grow explosively. Another problem is the lack of robustness, since many of the technologies are quite new, and they often have incompatibilities with different versions

[1] This is an extended version of a paper presented at the WebNet'98 conference in Orlando, Florida. The paper has received a "Top Full Paper Award".

of hardware, operating systems, HTTP servers and browsers [Hadjiefthymiades and Martakos 97][Everitt 96][Duan 96].

Finally, as noticed by [Everitt 96] and [Lazar and Holfelder 96], Web applications are frequently written by casual programmers, following a rapid prototyping approach. To tie new technologies to the knowledge of complex languages such as Perl, C++ or even Java may put a heavy load on these programmers.

Considering all these aspects, three points emerge as requirements for a Web site development tool:

1. suitability to work based on common Web standards (like CGI [CGI 96]), being portable to different platforms and servers;
2. openness to incorporate new technologies, in a gradual way;
3. flexibility to accommodate different uses, from the "quick and dirty" approach of casual programmers, to an object-oriented structured approach of a skilled team.

CGILua is a Web development tool based on CGI and the extension language Lua [Ierusalimschy et al. 96][Figueiredo et al. 96]. Although based on an "old" technology (CGI scripts), it differs from other tools by its set of features:

- a flexible and simple scripting language (Lua);
- the ability to mix different paradigms (templates and programming);
- an extensibility mechanism to dynamically load libraries written both in Lua and in C/C++.

These features make CGILua unusually portable, flexible, and extensible, while keeping it simple to use.

2 The Language Lua

Lua is a general purpose extension language that arose from our group's need to use a single extension language to customize industrial applications [Ierusalimschy et al. 96][Figueiredo et al. 96]. Currently, Lua is being used in more than a hundred products and prototypes, in many academic institutions and companies. The whole package is written in ANSI C, and compiles without modifications in all platforms that have an ANSI C compiler (DOS, Windows 3.1-95-NT, Next, Sun-OS, Solaris, Mac, Linux, OS/2, etc).

Lua integrates in its design data-description facilities, reflexive facilities, and familiar imperative constructs. On the "traditional" side, Lua is a procedural language with a Pascal-like syntax, usual control structures (*whiles*, *ifs*, etc.), function definitions with parameters and local variables, and the like. On the less traditional side, Lua provides functions as first order values, and dynamically created associative arrays (called *tables* in Lua) as a single, unifying data-structuring mechanism. As a simple illustration of Lua syntax, the code below shows two implementations for the factorial function in Lua:

```
function factorial (n)              function fatorial (n)
  local i = 1                         if n == 0 then
  local r = 1                           return 1
  while i<=n do                       else
```

```
    r = r*i                                return n*fatorial(n-1)
    i = i+1                              end
  end                                  end
  return r
end
```

There is no notion of a “main” program in Lua; being an embedded language, it only works embedded in a host client. Lua is provided as a library of C functions to be linked to host applications. The host can invoke functions in the library to execute a piece of code in Lua, write and read Lua variables, and register C functions to be called by Lua code. Moreover, *fallbacks* can be specified to be called whenever Lua does not know how to proceed. In this way, Lua can be augmented to cope with rather different domains, thus creating customized programming languages sharing a single syntactical framework [Beckman 91].

Functions in Lua are *first class* values. Like any other value, function values can be stored in variables, passed as arguments to other functions, or returned as results. The code in the `factorial` example shown above is actually syntactic sugar for the more general syntax

```
factorial = function (n) ... end
```

This piece of code creates a value of type `function`, and assigns it to the global variable `factorial`.

Lua is dynamically typed. Variables can handle values of any type. Whenever an operation is performed, it checks the correctness of its argument types. Besides the basic types `number` (floats) and `string`, and the type `function`, Lua provides three other data types: `nil`, with a single value, also called `nil`, whose main property is to be different from any other value; `userdata`, that is provided to allow arbitrary host data (typically C pointers) to be stored in Lua variables; and `table`.

The type `table` implements associative arrays, that is, arrays that can be indexed not only by integers, but by strings, reals, tables, and function values. Associative arrays are a powerful language construct: Many algorithms are simplified to the point of triviality because the required data structures and algorithms for searching them are implicitly provided by the language [Bentley 88]. Most typical data containers, like ordinary arrays, sets, bags, and symbol tables, can be directly implemented by tables. Tables can also implement records by simply using field names as indices. Lua supports this representation by providing `a.name` as syntactic sugar for `a["name"]`.

Unlike other languages that implement associative arrays, such as AWK [Aho et al. 88] and Tcl [Ousterhout 94], tables in Lua are not bound to a variable name; instead, they are dynamically created objects that can be manipulated much like pointers in conventional languages. The disadvantage of this choice is that a table must be explicitly created before used. The advantage is that tables can freely refer to other tables, and therefore have expressive power to model recursive data types, and to create generic graph structures, possibly with cycles.

Tables are created with special expressions, called *constructors*. The simplest constructor is the expression `{}`, which returns a new empty table. An expression like `{n1 = exp1, n2 = exp2, ...}` creates a new table, and stores in each field $\mathtt{n}_i$ the result of $\mathtt{exp}_i$. A typical example is the creation of a table to represent a point:

```
point1 = {x = 10, y = 30}
```

Constructors can also build lists: The expression `{exp1, exp2, ...}` creates a new table, and stores in each field i the result of $\texttt{exp}_i$. Therefore, after the assignment

```
days = {"Sun","Mon","Tue","Wed", "Thu","Fri","Sat"}
```

the expression `days[3]` will result in the string `"Tue"`.

Sometimes, more powerful constructor facilities are needed. Instead of trying to provide everything, Lua provides only a simple syntactic sugar: The syntax `name{...}` stands for `name({...})`; that is, a table is created, initialized, and passed as parameter to a function. This function can do whatever initialization is needed, such as (dynamic) type checking, initialization of absent fields, and auxiliary data structures updating (even in the host program). Often, Lua users are not aware that the constructor is a function; they simply write something like

```
window1 = Window{x = 200, y = 300, foreground = "blue"}
```

and think about "windows" and other high level abstractions.

Since constructors are expressions, they can be freely nested. This allows the description of quite complex objects in a convenient syntax, where the procedural nature of Lua is disguised with a declarative flavor. The example below illustrates this point, using the IUP library for GUI descriptions [Levy et al. 96] to define a dialog box:

```
Dialog = iupdialog{
           iupvbox{
             iuphbox{iuplabel{title="name: "}, iuptext{}},
             iupbutton{title = "OK"};
             alignment = IUP_CENTER,
             gap = 3,
             margin = 3
           }
         }
```

Another feature of Lua relevant to CGILua is its string manipulation facilities. Like other interpreted languages, such as Perl [Wall et al. 96], strings in Lua do not have a fixed or limited length. In fact, many Lua programs handle text files by first reading the whole file into a single string. Lua's predefined libraries offer two pattern-matching functions, one for finding patterns (`strfind`) and another for pattern substitution (`gsub`). An unusual feature of `gsub` is that a function can be used instead of the replacement string; whenever a match occurs, this function is called with the contents of the matching, and the string returned by it is used to replace the match. For instance, the following function is used in CGILua as part of the decoding of an URL encoding string:

```
function cgilua.unescape (str)
  str = gsub(str, "+", " ")
  return gsub(str, "%%([0-9A-F][0-9A-F])",
              function (x) return strchar(tonumber(x, 16)) end)
end
```

The first statement changes all "+" in the string to spaces. The second `gsub` matches all hexadecimal numerals preceded by "%", and calls a local function (the third argument of `gsub`). That function then converts the hexadecimal numeral (a string) into a number, and returns the corresponding character. (The double "%%" is not a typo; the character "%" has a special meaning in a pattern, so it must be escaped).

As a more complete example, the following code shows all that CGILua uses to decode an URL encoding string, storing the pairs `key->value` in a Lua table called `cgi`:

```
function cgilua.pair (key, value)
  key = cgilua.unescape(key)
  value = cgilua.unescape(value)
  cgi[key] = value
end

function cgilua.decode (string)
  gsub(string, "&?([^&=]*)=([^&=]*)&?", cgilua.pair)
end
```

The last `gsub` matches all pairs in the form `&key=value&` (where the ampersands may be not present), and for each pair it calls the function `cgilua.pair` with the strings `key` and `value` (as marked by the parentheses in the matching string). Function `cgilua.pair` simply "unescapes" both strings and stores the pair in a table.

3 CGILua overview

The simplest form of a CGILua script is as a Lua program; when the page is accessed, the program is ran and its output is interpreted as the final HTML page sent to the browser. The majority of CGI scripts are written this way, as with Perl [Stein 97], Tcl [Libes 96], C [Weber 96], Python [Vanaken 97], etc. In CGILua, these programs are written in Lua.

The main advantage of writing a script as a program is its flexibility. The full power of the language is available in the creation of a page. This includes all abstraction facilities of a programming language, plus predefined functions for pattern-matching and the like. 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 in a very low abstraction level. Moreover, the program logic and its interface get completely mixed, as the HTML tags are scattered around the program text.

An alternative, and more interesting, way to write CGILua scripts is to use an HTML *template* of the document to be generated. A template is a static HTML document, with some marks representing its dynamic parts. When the page is accessed, the template feeds a preprocessor that creates the final page. These templates use special marks to 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; their syntax is shown below.

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. 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.

[Fig. 1] shows a small program to print 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.

```
<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 1: A CGILua template

As a more realistic example, a previous version of this paper, formated in HTML, was written as a CGILua template that uses some Lua functions to automatically deal with cross-references and formatting. [Fig. 2] shows a piece of "code" used to write the second Section of the paper. The function `section` formats the section title according to a given paper format specification, and defines a HTML anchor for link references. The function `cite` is used in the example to generate an in-text citation to the references associated with the keys `lua-spe` and `lua-ddj`, introducing LaTeX-like facilities [Lamport 86]. To produce a static version of the document, we ran CGILua stand-alone over the source scripts; this is possible because CGILua does not need a HTTP server to run.

```
$|section('The Extension Language Lua')|$

<p>
Lua is a general purpose extension language
that arose from our group's need
to use a single extension language to customize industrial applications
$|cite("lua-spe")|$$|cite("lua-ddj")|$.
Currently, Lua is being used in more than a hundred
...
```

Figure 2: A sample of this paper before preprocessing

The use of CGILua to format a paper illustrates a typical use of the tool, where specific Lua functions are defined to shape the environment according to the task to be achieved. Users can explore CGILua in this way to define their abstractions and easily configure the environment for their needs.

```
<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>
  <!--$$ DBClose() $$-->
</body></html>
```

Figure 3: A database query in CGILua

[Fig. 3] shows how to create a CGILua script to query a database using templates. This example takes advantage of the CGILua extensibility, using a dynamically loaded package to access the database. The package simply defines four new Lua functions: `DBOpen` establishes a connection with a database, `DBExec` executes an SQL statement, `DBRow` traverses the resulting table, and `DBClose` closes the connection. Each row is returned as a Lua table, which is then stored in variable `m`. Notice in this example the interaction between the control marks and the HTML marks to format a table. The result of this template before processing is shown in [Fig. 4], and the final result —after preprocessing— is shown in [Fig. 5].

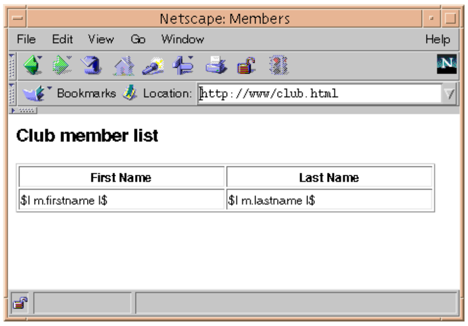

Figure 4: Template without preprocessing

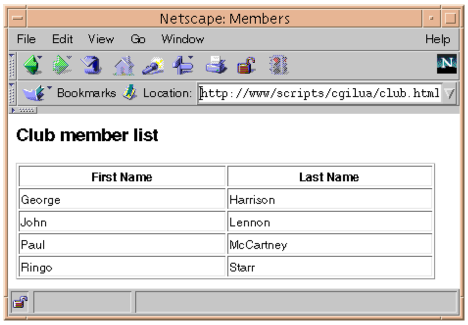

Figure 5: Final result of the template

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. [Fig. 6] and [Fig. 7] illustrate this style. Function `cgilua.preprocess` provides a degree of reflexivity: It allows a Lua script to process a template explicitly, as if the user had accessed that template. 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 to drive the creation of other pages. [Fig. 8] shows a script that validates the data from such a form.

```
<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`

CGILua can improve the management of Web sites even when used as a stand-alone processor 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 [Fig. 6] can be used to create many different forms, when fed with different values for table `field`.

4 CGILua 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 [Fig. 9].

```
if not checkfields() 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

The kernel is the program called by the HTTP 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. The configuration itself is almost 70% of all CGILua code. It 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. Since all these steps are done by a script, they can easily be adapted to local needs by the system administrator. In addition, a site may have several configuration scripts, allowing differentiated environments for different projects. For instance, personal user pages may have a stronger security policy than the one enforced on institutional pages.

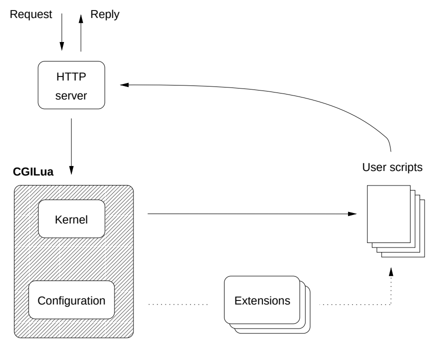

Figure 9: CGILua architecture

4.1 Portability

The tool's portability is ensured by the standards upon which it is based: Lua is fully implemented in ANSI C, which make it portable for every platform that has a C compiler. The CGILua *kernel* is implemented following the POSIX standard [Lewine 91] and uses CGI as the interface with the HTTP server, since this is the only current standard for interfacing servers with external programs. These features allow the use of CGILua in different platforms without modifications, with the same source code. Currently, CGILua is being used with different servers in MS-Windows 95 and NT, and most popular Unixes, such as SunOS, Solaris, Linux, IRIX and AIX.

Despite its name, CGILua does not depend on the CGI standard. It has been developed in a way such that the interface with the server is totally done by the kernel and the configuration script. This architecture allows the porting of CGILua to other APIs, for example Microsoft's ISAPI or Apache API, improving the script's performance.

Since CGILua scripts communicate with the server only through the kernel, they are independent of the kernel's interface with the server. This allows the use of the same scripts, even with different kernels. At the time of writing, only a CGI kernel is available, with an ISAPI kernel under development.

4.2 Extensibility

Both the configuration script and the user script file can run other Lua files. In this way, Lua libraries can be loaded before or during the execution of the user scripts, offering new facilities. Sometimes, however, such extensions must be written in C, either for efficiency reasons (like a cryptography package), or because a predefined C interface (like a database) is accessed.

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.

An example of this facility is the database package used in [Fig. 3]. 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 [Mediano 96], which offers access to different database systems, like mini-SQL and ODBC. This library is dynamically loaded by the configuration script, thereby offering all database facilities of CGILua.

In another example of its extensibility, CGILua is also being used in a network management system based upon SMNP [Rodriguez et al. 98]. Again, a C package has been built to offer SMNP facilities to Lua, allowing a CGI application to get and set SMNP variables. In this way, different management applications can be built over the Web by writing simple CGILua scripts.

Another package that can be used with CGILua is LuaOrb [Ierusalimschy et al. 98], a binding between Lua and CORBA that allows a Lua script to manipulate CORBA objects in the same way it manipulates local objects. LuaOrb is based on the CORBA Dynamic Invocation Interface, mapping its dynamic character to the dynamic type system of Lua.

These feature brings another level of utilization to CGILua, allowing different instances of the tool for different domains. Upon this perspective, CGILua is not only a tool for the creation of Web sites, but also a supporting tool for fully distributed applications.

4.3 Security Issues

A CGI script has the same security problems of any network server, since it is invoked by remote requests; from this point of view, any CGI script can be considered a mini-server. Because CGILua activates the user's Lua scripts, all security concerns must be extended to these scripts [Garfinkel and Spafford 96].

Lua is a language with 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. For instance, to disallow the writing of files, a configuration script has only to include the following line:

```
writeto = nil    -- "writeto" opens a file in writing mode
```

This solution is clearly too simplistic; most system functions can (and, many times, must) be used in restricted ways without security risk. For instance, a generic `writeto` function, which allows a script to write to any file, may be dangerous, but it can be restricted to open files only in a predefined directory. That could be done by redefining the function:

```
unsafe_writeto = writeto
writeto = function (filename)
            if checkfilename(filename) then
              unsafe_writeto(filename)   -- do the "real" open
            else error("cannot open " .. filename)
            end
          end
```

Typically, the new version needs access to the original function to perform the actual task, after the security checks. But, with this previous code, function `unsafe_writeto` is still available not only to the new `writeto`, but to the whole user script. The problem here is how to allow the new `writeto` to access the old, unsecure version, without giving global access to it.

The solution adopted by CGILua is based on a mechanism of Lua called *upvalue*. Originally, upvalues were envisioned to support closures. An upvalue, syntatically written as `%name`, is like a variable access, but whose value is computed when the function is created, instead of when the function is called. With this mechanism, the previous redefinition of `writeto` can be written as

```
unsafe_writeto = writeto
writeto = function (filename)
            if checkfilename(filename) then
              %unsafe_writeto(filename)   -- do the "real" open
            else error("cannot open " .. filename)
            end
          end
unsafe_writeto = nil   -- no more accesses after this point
```

When this new function is created, the upvalue `%unsafe_writeto` is evaluated, resulting in the original function. Therefore, the unsecure version is kept in the closure of the newly created version, but after the last assignment it is no longer accessible in any other point of the program.

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.

5 Final Remarks

Despite its inherent academic nature, CGILua has achieved industrial relevance, being employed in many commercial web systems. A major example of industrial use is SIGMA (*Sistema de Gestão do Meio Ambiente*). SIGMA is a WWW system being developed for PETROBRAS (The state-owned Brazilian Oil company). Its function is to manage the procedure of obtaining environmental licenses and to inform users about rules and technical procedures. The system is part of a strategy to obtain the ISO 14000 certificate. Around one hundred people will use SIGMA as a work tool on a daily basis. Moreover, parts of the system will be available to the general public. The system generates and collects information by an active communication with a database. DBLua is used to provide the connection through ODBC to an SQL Server database.

Unlike many other Web tools, both Lua and CGILua follow the same "minimalistic" principle: Instead of providing a myriad of mechanisms for specific purposes, they provide a few generic *meta-mechanisms* to address general issues. In this way, they can handle rather diverse application domains.

At a glance, the main features of CGILua are

flexibility The use of an extension language both in the architecture of CGILua and for writing user scripts makes the tool highly flexible. A scripting language greatly facilitates rapid prototyping, an important methodology for Web applications [Everitt 96]. There are no fixed roles for what is written in Lua and what is written in HTML templates. Moreover, many aspects of the tool, from error handling to security policies, can be easily tailored by the system administrator.

extensibility Applications can use libraries written both in Lua and in C. Lua libraries allow the extension of internal CGILua facilities, as illustrated by the definition of format functions for papers. C libraries are used to allow access to external facilities, as illustrated by the SMNP example, and to implement performance-critical functions.

simplicity The whole system has less than ten thousand lines of code ($\sim$ 1500 lines for GCILua plus $\sim$ 8000 for Lua). All its sources and binaries can be put on a single floppy disk. Its use is also simple. Most users are able to start using CGILua in less than half an hour. Lua is a small language, with a simple Pascal-like syntax and a simple semantics.

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 runs without changes in any other platform.

The implementation of CGILua is freely available at `http://www.tecgraf.puc-rio.br/manuais/cgilua`

References

[Aho et al. 88] A. V. Aho, B. W. Kerninghan, and P. J. Weinberger; *The AWK programming language*; Addison-Wesley, 1988.

[Beckman 91] B. Beckman; A scheme for little languages in interactive graphics; *Software: Practice and Experience*, 21:187–207, 1991.

[Bentley 88] J. Bentley; *More programming pearls*; Addison-Wesley, 1988.

[CGI 96] CGI - Common Gateway Interface; W3C - World Wide Web Consortium, 1996.

[Duan 96] Nick N. Duan; Distributed database access in a corporate environment using Java; In *Fifth International World Wide Web Conference*, 1996.

[Everitt 96] P. Everitt; The ILU requester: Object services in HTTP servers; In *W3C Informational Draft*, 1996.

[Figueiredo et al. 96] L. H. Figueiredo, R. Ierusalimschy, and W. Celes; Lua - an extensible embedded language; *Dr. Dobb's Journal*, 21(12):26–33, 1996.

[Garfinkel and Spafford 96] S. Garfinkel and G. Spafford; *Practical UNIX & Internet Security*; O'Reilly & Associates, Inc., second edition, 1996.

[Hadjiefthymiades and Martakos 97] S. P. Hadjiefthymiades and Drakoulis I. Martakos; Improving the performance of CGI compliant database gateways; In *Sixth International World Wide Web Conference*, 1997.

[Ierusalimschy et al. 96] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes; Lua - an extensible extension language; *Software: Practice and Experience*, 26(6):635–652, 1996.

[Ierusalimschy et al. 98] Roberto Ierusalimschy, Renato Cerqueira, and Noemi Rodriguez; Using reflexivity to interface with CORBA; In *IEEE International Conference on Computer Languages (ICCL'98)*, pages 39–46, Chicago, IL, May 1998. IEEE Computer Society.

[Lamport 86] L. Lamport; *LaTeX: A Document Preparation System*; Addison-Wesley, 1986.

[Lazar and Holfelder 96] Z. Peter Lazar and Peter Holfelder; Web database connectivity with scripting languages; 1996.

[Levy et al. 96] Carlos H. Levy, Luiz H. de Figueiredo, Marcelo Gattass, Carlos J. Lucena, and Don D. Cowan; IUP/LED: a portable user interface development tool; *Software: Practice and Experience*, 26(7):737–762, 1996.

[Lewine 91] Donald Lewine; *POSIX Programmer's Guide*; O'Reilly & Associates, Inc., 1991.

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

[Mediano 96] M. Mediano; *DBGraf - Manual de Referência*; TeCGraf, May 1996.

[Ousterhout 94] J. K. Ousterhout; *Tcl and the Tk Toolkit*; Addison-Wesley, 1994.

[Rodriguez et al. 98] N. Rodriguez, M. Lima, A Moura, and M. Stanton; A platform for the development of extensible management applications; In *INET'98*, Geneva, Switzerland, july 1998.

[Stein 97] L. Stein; A Perl library for writing CGI scripts; *Web Techniques*, 2(2), February 1997.

[Vanaken 97] M. Vanaken; Writing CGI scripts in Python; *Linux Journal*, 34, February 1997.

[Wall et al. 96] L. Wall, T. Christiansen, and R. L. Schwartz; *Programming Perl*; O'Reilly & Associates, Inc., second edition, September 1996.

[Weber 96] J. Weber; libcgi; URL: http://wsk.eit.com/wsk/, 1996.

Acknowledgements

We 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 was developed at TeCGraf/PUC-Rio (Group of Technology on Computer Graphics at the Catholic University of Rio de Janeiro), and it has been partially supported by CNPq (the Brazilian Research Council).