From Glosser-R$\underline{u}$G to Glosser-W$\underline{e}$B

D.A.Dokter
Alfa-Informatica, University of Groningen
Postbus 716, 9700 AS, Groningen
dokter@let.rug.nl

April 18, 1997

Abstract

The *World-Wide Web* can in a sense be regarded as a *global platform* for applications, in addition to the more common view of the Web as being a *global database*. However, exploiting the Web in this respect has a number of limitations, mostly concerning security and the heterogeneity of the individual components that constitute the Web. These limitations effect the development of Web applications in several ways. This report concerns the re-design of Glosser-R$\underline{u}$G, an application that facilitates the reading of French texts for Dutch speaking students, into a Web-borne version. This Web version, called Glosser-W$\underline{e}$B, exploits a number of methods and tools that enable fast re-development and re-use of code for applications.

1 The Web as an application platform

The last decade has shown an enormous growth of electronically available information, not only because of the constant generation of new data, but also because of the increasing efficiency in communication between computers. Electronic communication channels have evolved into the highways of the *Internet*, providing world wide access to corpora, papers, libraries, virtual musea, etc. Apart from the development of the *physical* realization of the Web, a number of tools and protocols have been developed that not only increase the speed and efficiency of data transfer, but also drastically change the nature of the information that can be accessed. Recently, *CGI scripting*[1] [7] and the development of *JAVA* and *JavaScript* (JS) [6] have introduced the possibility of running applications on the Web. The first standard enables a client to execute programming code on the host, the second and third enable clients to retrieve code with a common Web request and execute it locally. The development of these standards, imply that the Web can be regarded as an *abstract platform* for applications, much like UNIX and Windows.

The most powerful instruments in rapid re-development of applications for the Web are CGI scripting and JS. The latter standard is, in principle, intended only to enhance the mark-up of Web documents with a number of high-level options to enable interactivity on Web pages, while CGI scripting can be used to perform advanced computing. Since the code for CGI scripts resides on the server-side of the communication, most code of existing applications can easily be re-used for Web versions of these systems. Exploiting the Web as a *global* platform causes an enormous decrease in the space and time needed for acquisition and storage of application related data, and makes distribution and maintainance of software almost trivial once a Web version has been launched. A multitude of platforms might use the same source, residing on a single machine, such that the distribution of a new version is just a matter of updating the source code. Of course, this seemingly ideal picture is clouded by some quite severe restrictions. Most restrictions involve security, some are concerned with physical constraints on the Web architecture and protocols.

Further, although JS is an ideal means for rapid development, the number of functions this script language can perform is still quite limited. For Java, this limitation does not exist, since this is a full-fledged programming language, but rewriting an application in Java takes much more time and effort. This implies that the range of remote applications that can be easily re-developed for the Web is also restricted (future development of JS will probably enhance the number of possible actions).

[1] CGI stands for Common Gateway Interface, a protocol designed to enable interactivity of Web documents

One of the main questions then is, what the range of applications is, that a combination of JS and CGI scripting can handle, and what methods exist to deal with typical Web limitations.

2 Glosser-R*u*G

Glosser-R*u*G is a *Computer Assisted Language Learning* (CALL) application that facilitates the reading of French texts for native Dutch speakers. The application supports this task by providing an on-line dictionary, morphological analysis, part-of-speech disambiguation and examples of word use in a corpus of texts. For each word in the text the user can select what specific information is needed and start a lookup for this particular word. Hence, the application is both useful in language learning and as a practical tool that provides just-in-time knowledge.

The fact that the focus of the Glosser-R*u*G is on *support*, not drills, this application typically does not require supervision, so that the potential use of Glosser-R*u*G is almost unrestricted, given that the program is sufficiently self-explanatory. This makes Glosser-R*u*G extremely suitable for use on the World-Wide Web. An early spin-off of the application has been developed, to study the feasibility of a Web-compatible version in a technical sense. The result of this study is outlined in the WELLS project [1], and clearly showed that most of the necessary functions are easily implemented with the available features. However, WELLS does not implement the complete functionality of Glosser-R*u*G, and the program uses the Web in a different way: WELLS exploits the Web not only as a platform, but also as a distributed, global database, whereas Glosser-R*u*G is developed to incorporate some state-of-the-art linguistic techniques, where much more control over the database is necessary. Data for Glosser-R*u*G therefore resides local to the server. Also, local sources obviously require far less time for search and retrieval, and are less likely to be or become inconsistent.

Although Glosser-R*u*G is specifically designed for support on French texts, the architecture is largely modular, so that the application is easily adaptable to processing any language for which the informational sources used in the application are available. Moreover, a number of tools have been developed to enable the maintainer of the program to modify the informational sources the program uses, see for instance [4]. Glosser-R*u*G is in principle an integrated and structured account of several previously developed linguistic tools and existing resources. The major informational resources that are incorporated are: an on-line dictionary provided by Van Dale Lexicografie [5], morphological analysis software provided by Rank Xerox [2], and examples of the use of words in especially collected text corpora. The program relies heavily on the morphological analysis software, which provides the link between the text that is being processed, and the other informational sources. Each word for which information is being sought is analyzed within its linguistic context (the sentence it occurs in) and disambiguated with respect to its base form, part-of-speech (POS) and the properties of the word (or, inflections) that determine its appearance. The analysis thus provides the link between the possibly inflected form of the word as it appears in the text and the base form, or "citation form", listed in dictionary entries [8]. This so called 'lemmatization' is also exploited for lookup of examples. All various forms of a word occurring in the text can be referred to by a single lemma, and can thus be indexed by this single form. This effectively guarantees a wide variety of inflections and lexical environments in the examples for a specific word, providing a rich context for vocabulary improvement and word comprehension. *Lemmatization* thus plays a dual role: it is informational in that it displays properties of words, and it is functional for processing, by providing the link between text on the one hand, and dictionary and examples on the other. Figure 1 provides a complete overview of the flow of information within Glosser-R*u*G.

This scheme clearly shows the modular organization of Glosser-R*u*G, and the interdependencies of the composing parts. To grant the user as much freedom as possible in specifying the information needed, any source can be switched on or off at any stage in a session. This is implemented by providing a separate window for every source. Also, a separate window is used to provide help on the application.

The goal in re-developing Glosser-R*u*G into a version on the Web was to maintain as much of the functionality as possible. However, Glosser-W*e*B is intended for demonstrational purposes only for the time being, and some of the sources can not be made publicly available, so that the number of texts that can be processed is limited, and these texts reside on the Glosser-W*e*B server.

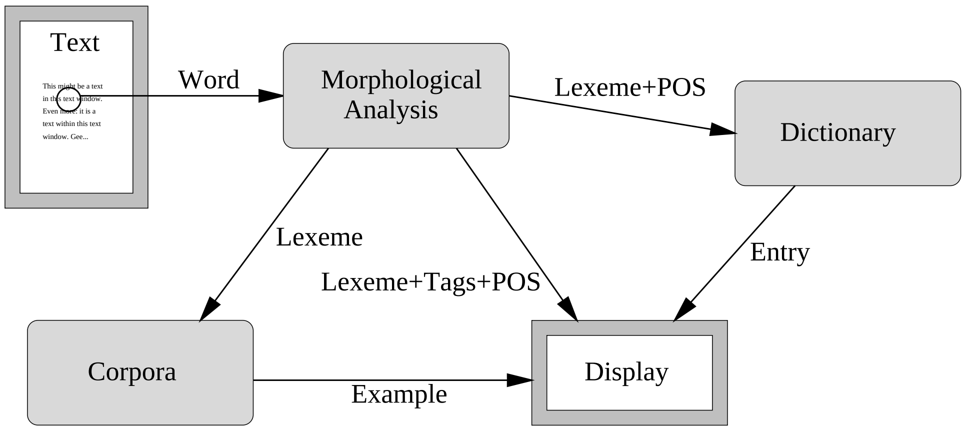

Figure 1: *The flow of information for a single lookup within Glosser-RuG, given that all sources are activated.*

3 CGI scripting: the main routines

A common way to provide interactivity for Web documents, is to incorporate CGI scripting. In fact, a CGI script can be virtually any kind of application that runs on the host server, and relays its output to the client, that is, the only feature that makes a CGI script special is its *status* as such. (A further point here is that although CGI programs are often written in script languages and therefore commonly referred to as CGI 'scripts', this is no prerequisite: C or Visual Basic are likewise possible for implementation.) A CGI script behaves, from the point of view of the client, largely like any ordinary request on the Web: an address is called with the HTTP protocol, and a document is returned. However, a CGI script can be called with additional arguments, specified as part of the URL, such that values specified by the client can be passed to the script. Arguments are either passed as the body of the request, when the HTTP method 'POST' or 'PUT' is used, or the server stores the arguments in the environment variable 'QUERY_STRING', when 'GET' is used. In the former case, the CGI script can internally simply regard the argument as a command line argument, for example *argv[1]* in C. In the latter case, the script has to extract the arguments from the environment, for example $ENV(QUERY_STRING) in Perl. In both cases, request *headers* are stored in environment variables. These headers contain information on the client, the type of the request, etc. (for details on the CGI standard and headers, see [7]). This enables the script to exploit features such as the type of browser the request comes from, or the address of the server that sent the request. Being able to account for requests with variable arguments enables the host to create documents on the fly, tailored to the specific values that were passed. With respect to the Web as a platform for applications, the similarity of CGI scripts and normal programs imply that in re-developing applications for the Web, the developer can incorporate large pieces of code from the original application, and perform the same tasks within Web applications. Moreover, if applications make extensive use of large databases, as is the case for Glosser-WeB, resources, in general, have to reside on the host and functions such as searching and retrieval *have* to be conducted on this machine. However, there is one disadvantage: CGI scripts are addressed by the common HTTP protocol, which does not support control over requests apart from displaying the information returned in the browser. For JS for instance, this means that the result of a CGI script can not be stored in a variable, thus no further processing is possible before display. This poses some problems for scripts that exploit results of earlier scripts, and for display of results in different targets (frames or windows): every request can only display its response in one target.

For Glosser-WeB this implies that different sources have to be implemented by different CGI scripts, since information for each source is displayed in a separate window. The first script is processed with an integer and a filename as arguments, used to retrieve the i^{th} word from that file, which resides on the host, and the surrounding lexical context. This script provides the morphological analysis that can be displayed, and lemmatization for further processing, as explained in section 2. The lexeme found by morphological analysis serves as argument for the other two scripts, that implement lookup of examples and the dictionary entry corresponding to that lexeme.

3.1 CGI and JS communication

A problem of using multiple CGI scripts in a single application, is posed by the lack of communication between the scripts on the one hand, and the (JS) code on the other. Once a request is sent, control over the request is no longer possible for JS. This clearly poses a problem for Glosser-W<u>e</u>B, since the result of morphological analysis is both informational and functional in further processing, as explained in section 2. Moreover, the user is able to specify whether morphological information should or shouldn't be displayed, but in both cases the analysis has te be conducted by the CGI script, that has to be called with a target. The first problem then is to provide both an HTML document and a result for further processing by a single CGI script, and to be able not to display the document if the user doesn't want the analysis to be displayed. A solution to the first problem is provided by the availability of so called *cookies*. Cookies are in fact just small strings, containing some specific information, such as a name, a value, a return address, etc[2]. Two features of the cookie that are exploited by Glosser-W<u>e</u>B are the possibility of adding a specific value, and an address specification for URL's [3] that, when requested, will receive the cookie as response header. In this way, one CGI script can 'install' a cookie on the user side, and specify a URL af another CGI script that will receive the cookie when addressed. The use of cookies is the only way for Web applications to store information apart from Java/JS runtime variables on the user side of the connection. The cookies are stored in memory during a single browser session, and written to a special file after the session has been ended. There are of course limitations in size and duration of storage, but this mechanism provides the developer of Web applications with an extremely useful means of exchanging values between different parts of the application. To set a cookie, it can be defined in the response header information for a document, thus returned by a CGI script. The lexeme found by morphological analysis is defined as a cookie, that is to be added to every call to the URL specifying the script for dictionary and examples lookup. The latter two scripts can simply retrieve the cookie from the request header, since the host server stores the cookie in the environment variable 'HTTP_COOKIE'.

The second problem, where to leave the returned document when no window is opened, is less elegantly solved by providing a 'hidden' frame in the main window, which is not user controlled, and therefore present during the whole session. This frame is still visible, since it is impossible to completely hide frames and windows, but so small that it can not visibly display any information. For the user the document, that is still loaded, is completely hidden. These specifications render the following informational flow within Glosser-W<u>e</u>B:

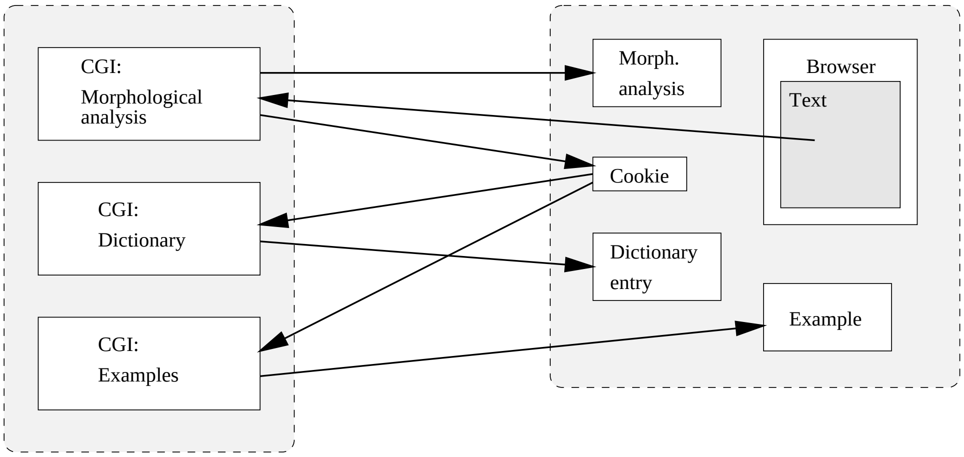

Figure 2: *The flow of information bewteen the major components of Glosser-W<u>e</u>B.*

Another problem in CGI and JS communication is introduced by temporal aspects of the Web. Since there is no way in JS to manipulate responses, the code does not normally wait for a request to be returned and subsequent statements are therefore executed immediately after the request

[2] See *http://www.netscape.com/newsref/std/cookie_spec.html* for the specific syntax of cookies.)

[3] *Uniform Resource Locator*: the address specification of a resource on the Web

has been sent. Due to the unpredictable temporal behavior of the Web, maintaining chronology in a sequence of requests and responses is therefore almost impossible. There is of course a time-out value for a request, which can be set by a client, so that only responses within that period are regarded, but incorporating this feature into your application is obviously an ad-hoc solution This poses a problem for scripts depending on intermediate results, as is the case for Glosser-WeB: morphological analysis provides the lexeme that is used for dictionary lookup and examples. This implies that the CGI scripts performing the latter two functions, can only start when the result of the script performing morphological analysis is set. A solution is provided by the fact that it is possible to determine user-side when a document addressed by a URL has been fully retrieved, by the *onload()* event handler of JS. This feature checks for a specific display (frame or window) when a new document is loaded and performs an action accordingly. This initial solution however, is not completely applicable for Glosser-WeB, due to the multi-window environment specified. Every window is a top level in JS, so that child windows cannot address their parents, nor other windows at the same level of the hierarchy as themselves. Only ancestors can address their children, by the variable that is instantiated when a child window is opened.

Since the result of the different CGI's for Glosser-WeB is displayed in different windows, the *onload()* feature can not be directly used: the window displaying morphological analysis can not address the dictionary or examples window. Moreover, windows might not be opened (a user can specify which sources are to be displayed). Thus, control has to reside on the main (browser) window. From this window control has to check which windows are opened, and when the document created by morphological analysis is completely loaded in the child window. This is implemented by using the *hidden* object of JS. The morphological analysis window contains a hidden form object (that is, the user does not see this object) containing a boolean. As soon as a document is loaded in the morphological analysis window, this boolean is set to *true*. In the main window, a master process checks this value after the CGI script implementing morphological analysis has been started, and as soon as the condition is set to *true*, the scripts implementing dictionary and examples lookup can be requested. This function is elegantly implemented by a "classic event loop", calling itself with a time lag when the hidden boolean is false, and terminating when it is true.

4 Glosser-RuG in the Web-browser environment

The initial goal of re-developing Glosser-RuG into Glosser-WeB, was to maintain as much of the functionality as possible. In terms of the front-end this means: a main window, consisting of a text frame and a frame that specifies most of the control for the application, three windows for displaying dictionary entries, morphological analyses and examples respectively and a window displaying help on the different parts. The main window is displayed in the Web browser itself, the other windows have to be newly created when the user decides to use them. This is implemented quite easily with JS. The front-end of Glosser-WeB is displayed in Figure 3.

The only static window during a session with Glosser-WeB, is the main window. Display of the other windows is controlled by the user, who can at any stage in the session specify what sources of information have to be displayed, thus what windows have to be opened or closed. The same goes for the *help*-function. Help is called from the main control, and provides a number of hypertexts on the visible parts of the application.

Implementing similar *behavior* for Glosser-RuG and Glosser-WeB however, proved to be more tiresome. First, JS and HTML are rather restricted with respect to editor-like text manipulation, so that the selection of a word can not be conducted on the fly, as in Glosser-RuG (see [3] for details on word selection). One alternative, the cut-and-paste method, where the user has to mark a word by dragging the mouse cursor over it with the left button held down and then paste the word in an text-entry object, is particularly error-prone, since the user completely controls selection and could easily submit erroneous strings (numbers, punctuation marks, etc.). Checking all selections and displaying error messages upon erroneous selections is seriously detracts from the ease with which users work with a program. However, since this version of Glosser-RuG is intended for demonstration purposes only, another solution is at hand. A resricted number of texts have been selected server-side, and have already been preprocessed, such that two different files are created. The first file, where every 'proper' word is in fact a hyperlink, is displayed in the main window upon request, and the second file, containing only the proper words in sequence of appearance in the original text, is used as index

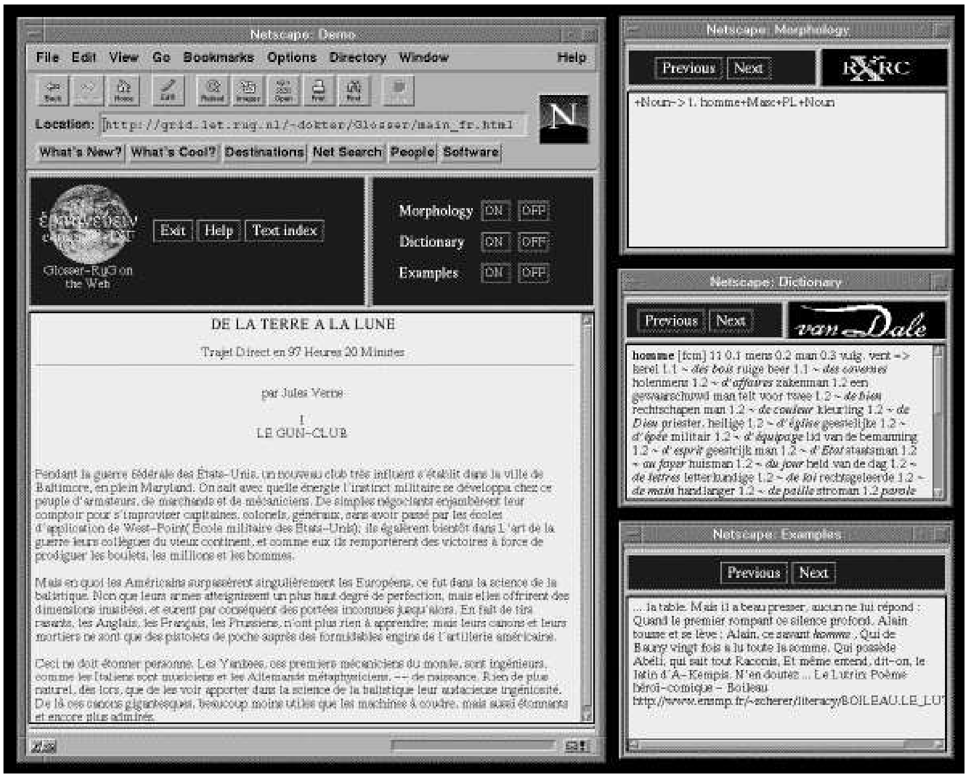

Figure 3: *The front-end of Glosser-WeB, provided that all sources of information are switched on. The help-window is omitted. The specific appearance is of course depending on the type of browser used.*

server-side. The links of the first text do not call a remote HTTP document, but a JS routine, with the position of the word the link refers to as an argument. That is, the i^{th} word of the text calls a JS routine *display(i)*. This routine implements the calls to the different CGI scripts, which provide the requested information, where the name of the file that is being processed is added as argument. For the Web version to pass beyond the demo-stage, a way will have to be found to permit the user total freedom in choosing the text to be processed, but that implementation should be JAVA based, which offers a much wider range of possibilities.

5 Conclusions

Although the Web imposes some constraints on the range of applications that can be usefully made available, even rather complex systems can often be implemented on the Web to some extent within relatively short time. In the case of Glosser-RuG most of the functionality has been preserved, with little cost regarding efficient algorithms. The combination of CGI scripting for the main routines, and JS/HTML as interface and intermediary between interface and CGI scripts offers a rich environment for applications on the Web and a means of rapid re-development of existing applications, although some obscure code might be necessary to overcome some of the shortcomings of JS.

Acknowledgements

I would like to express my gratitude to Prof.Dr.Ir. John Nerbonne for consultancy and for the opportunity of developing Glosser-WeB within the tight schedule of the GLOSSER project. Further, I would like to thank Allessandro Allodi for help on JavaScript, and Edwin Kuipers for searching the Web for 'cookies'. The research on Glosser-RuG is set within the framework of GLOSSER. This project is funded by COPERNICUS grant 343, and aims at applying state-of-the-art linguistic techniques to Computer-Assisted Language Learning.

References

[1] A.Allodi D.A.Dokter and H.W.A.E.Kuipers. WELLS; Web Enhanced Language Learning. In *Language Teaching and Language Technology*, Groningen, The Netherlands, 1997.

[2] F.S.D. Bauer and A. Zaenen. Locolex: Translation rolls off your tongue. In *Proceedings of the conference of the ACH-ALLC'95*, Santa Barbara, USA, 1995.

[3] D.A. Dokter. Glosser-R*u*G, prototype december 1996. forthcoming, Alfa-informatica, University of Groningen, Groningen, The Netherlands, 1997.

[4] D.A. Dokter. Inverted indexing for bilingual examples of word use. forthcoming, Alfa-informatica, University of Groningen, Groningen, The Netherlands, 1997.

[5] Van Dale Lexicografie. *Handwoordenboek Frans-Nederlands,* 2^e *druk.* VanDale Lexicografie b.v., Utrecht, 1993.

[6] J.J. Manger. *JavaScript Essentials.* McGraw-Hill, New York, NY, USA, 1996.

[7] net.Genesis and D. Hall. *Build a Web Site; The Programmers Guide to Creating, Building, and Maintaining a Web Presence.* Prima Publishing, Rocklin, CA, USA, 1995.

[8] R.Sproat. *Morphology and Computation.* MIT Press, Cambridge, MA, USA, 1992.