

# GWL – A Generative Language for Building Web Applications

David J. Cooper Jr.  
Genworks International

May, 2000

## Abstract

This paper describes an approach to building Web applications using a tree of dynamic Common Lisp objects to represent the tree of pages in a Web application. The macro *Defpage* is introduced, which stands for *Definition of a Page* and is used to define the “blueprint” for a web page and its role within the web application hierarchy of pages.

## 1 Introduction

This paper presents a basic overview of an open-source implementation called GWL, or Generative Web Language. GWL is a superset of Common Lisp embedded in Common Lisp and makes use of either the Common Lisp Hypermedia Server or the Allegroserve webserver from Franz, Inc. GWL runs on top of KTI’s ICAD/IDL/KBO System as well as in a “standalone mode” on top of base Common Lisp.

GWL essentially “models” a traditional website’s hierarchy of web pages, using dynamic object instances. The system creates object instances as the user or group of users visits the corresponding “pages,” and the computed information required to generate the HTML for the page can be cached using automatically generated memoized methods.

Each object which is to be presented as a web page uses a presentation method, defined on or associated with that object, which generates the actual HTML. This method will generally make heavy use of HTML synthesis macros, such as those provided with CL-HTTP [Mallery] or the “htmlgen” package provided with Allegroserve [Allegroserve].

The current GWL implementation is available from Genworks as free, open-source software.

## 2 Background

According to Paul Graham [Graham], most web sites can be represented with a tree structure, with the root-level node being the main “homepage,” and internal site hypertext links being pointers into direct children and other descendants of this root node.

Conventional web sites are in fact made up in the form of a tree structure, but this structure is usually more or less a static one. A completely static, non-interactive site will literally consist of a directory tree of static HTML files, hopefully linked together with hypertext links in some manner which makes navigation convenient.

Web sites which support some sort of form submittal and scripted response represent a slightly increased level of dynamicism, but usually still in the framework of a basically static structure.

If we think of a web site hierarchy as an actual knowledge model, any form submittal and response involving a procedural script execution must procedurally modify the state of the model, and procedurally keep track of any dependent data, elsewhere in the hierarchy.

GWL, in the spirit of a Knowledge Base or Knowledge-based Engineering system, presents a true dynamic model to represent the web site, supporting demand-driven evaluation of attributes (cached methods) which use a backward-chaining approach to keep track of their dependencies when the state of the model is modified, e.g. through form submittal.

## 3 Language Elements

### 3.1 Defpage

*Defpage* is the basic macro for defining web pages in GWL. The original meaning of “*page*” refers to actual web pages to be generated, but in practice the usage can be extended to include any kind of object to compute information. A Defpage maps directly into a Class definition.

The basic syntax of the Defpage is

```
(Defpage Class-Name Mixin-List Spec-Plist)
```

*Class-Name* is any non-keyword symbol. A class definition will be generated for this symbol, so any name you use will override a class definition if one is already defined with the same name.

*Mixin-List* is a list of other defpage-names from which this defpage will inherit. It maps directly into the normal mixin list.

*Spec-Plist* is a plist made up of pairs made from special keywords and other keyword lists or plists. The special keywords currently supported are the following:

- `:Inputs` (*section 3.2*)
- `:Attributes` (*section 3.3*)
- `:Children` (*section 3.4*)
- `:Methods` (*section 3.5*)
- `:Optional-Inputs` (*section 3.6*)

All the above *Defpage* sections define *messages* on the page object instance, which fundamentally are methods which take the instance as their first argument. In the case of `:Inputs`, `:Attributes`, and `:Optional-Inputs`, the methods do not take any additional arguments. In the case of `:Children` and `:Methods`, the methods may take additional arguments.

### 3.2 `:Inputs`

`:Inputs` are keyword symbols which represent values which can be supplied either:

1. into the toplevel page of a page hierarchy, at page instantiation, or
2. into a child page, using a `:Children` specification (see *section 3.4*, `:Children`, below)

Inputs are specified as a simple list of keyword symbols, for example:

```
(defpage person ()
  :inputs
  (:first-name
   :last-name
   :age
   :image-url))
```

### 3.3 :Attributes

`:Attributes` are keyword-expression pairs which are strictly computed by a page instance (i.e. they cannot be specified upon instantiation or modified in any way; they are always computed based on their expression).

Attributes actually translate into cached methods on the instance. They will only be computed when demanded, then their values will always be cached. Only if a value on which they depend becomes modified will their values be recomputed from their expressions (but still, only when demanded).

The referencing macro “the” is used to refer to the values of other messages in the same page, or, through reference-chaining (see 3.4, `:Children`, below), the values of messages in other page instances.

Example:

```
(defpage person ()
  :inputs
  (:first-name
   :last-name
   :age
   :image-url)
  :attributes
  (:full-name (concatenate 'string
                           (the :first-name)
                           " "
                           (the :last-name))))
```

### 3.4 :Children

`:Children` is used to specify a list of instance specifications, where each instance is considered to be a *child* of the current page. `:Children` may be defined as single instances (non-*Quantified*), or as single-dimensional arrays of instances (*Quantified*).

Inputs to each instance are specified as a plist of inputs and value expressions, spliced in after the page’s name and type specification

Note the *reference-chain*,

```
(the :hotel :water-usage)
```

This will reference the value of the `:water-usage` message in within the instance returned by the `:hotel` message:

#### 3.4.1 non-*Quantified* :Children

Examples:

```
(defpage city ()
  :attributes
  (
   :total-water-usage (+ (the :hotel :water-usage)
                        (the :bank :water-usage)))
  :children
  ((:hotel :type 'hotel
       :size :large)
   (:bank :type 'bank
       :size :medium)))

(defpage hotel ()
```

```

:inputs
(:size)

:attributes
(:water-usage (ecase (the :size)
                     (:small 10)
                     (:medium 20)
                     (:large 30))))

(defpage bank ()
  :inputs
  (:size)

  :attributes
  (:water-usage (ecase (the :size)
                       (:small 2)
                       (:medium 3)
                       (:large 4))))

```

### 3.4.2 *Quantified* :Children

:Children may be quantified as a single-dimensional array. This is known as *:Series Quantification*.

Each member of the quantified set will automatically answer an *:Index* message, which starts at 0 and goes up to one less than the total number of elements in the quantified set.

Note that the referencing macro *the-child* may be used to reference into the current child instance (in *Quantified* :Children as well as in normal non-*Quantified* :Children). This can be particularly useful for *Quantified* :Children, in order to access the *:Index* of the current member.

Example:

```

(defparameter *presidents-data*
  '(:name
    "Carter"
    :term 1976)
    (:name "Reagan"
    :term 1980)
    (:name "Clinton"
    :term 1990)))

(defpage presidents-container ()
  :attributes
  (:data *presidents-data*)

  :children
  ((:presidents :type 'president
    :quantify (:series (length (the :data)))
    :name (getf (nth (the-child :index)
                    (the :data))
               :name)
    :term (getf (nth (the-child :index)
                    (the :data))
               :term))))

```

```
(defpage president ()
  :inputs
  (:name :term))
```

The members of a quantified set are accessed like methods, by wrapping extra parentheses and sending an index number as the argument.

```
(the (:presidents 0) :name)
--> "Carter"
```

### 3.5 :Methods

Methods are actual uncached methods on the object. They are defined with a lambda list.

Methods are called in a normal reference chain but their name is wrapped in parentheses and their argument list is spliced on after the name, within the parentheses.

Example:

```
(defpage hotel ()
  :inputs
  (:room-rate)

  :methods
  ((:total-cost
    (number-of-nights)
    (* (the :room-rate) number-of-nights))))

(the (:total-cost 7))
--> 700
```

### 3.6 :Optional-Inputs

:Optional-Inputs are like a hybrid of :Attributes and :Inputs – they have a default expression but this can be overridden by passing it as an input either from the toplevel or from the parent page instance.

## 4 Basic Usage

Every *defpage* whose instances are to be presented in an actual application must define a presentation method called :Write-HTML-Sheet. This method will normally make use of HTML synthesis macros such as those found in CL-HTTP and in Allegroserve.

Example:

```
(defpage president ()
  :optional-inputs
  (:name "Carter"
   :term 1976)

  :methods
  ((:write-html-sheet
    ()
    (html (:html
           (:head (:title "Info on President: " (:princ (the :name))))
```

```

(:body
  (:table (:tr (:td "Name")
               (:td "Term"))
          (:tr (:td (:princ (the :name)))
               (:td (:princ (the :term))))))))))

```

Once a page has been defined with its `:Write-HTML-Sheet` method, a user may instantiate it by visiting the pre-defined search URL “make,” and specifying the symbol for the *Defpage* as the search argument, for example:

```
http://mie.genworks.com:9000/make?president
```

The above URL would create an instance of the *President* page, and display the resultant HTML from running its `:Write-HTML-Sheet` method.

## 5 Page Linking

Creating hypertext links to other pages in the page hierarchy is generally accomplished with the built-in method `:Write-Self-Link`. This method, when called on a particular page instance, will write a hypertext link referring to that page instance.

The hypertext link consists of encoded S-expressions including information such as a *root-path*, which is a list of keywords describing the path through the page instance hierarchy, from the root instance down to the specified page, as well as an *instance-id*, which identifies the particular root-level instance which is the “root” of the relevant page hierarchy.

The `:Write-Self-Link` method encodes these S-expressions into an HTML search query argument.

In the following example, note the use of the referencing macro *The-Object*, which is used to send messages to an instance when that instance is held in a variable, rather than in an actual message of the *Defpage* identified by a message keyword. In this case, *The-Object* is used to reference an object which is held in a *Dolist* iterator variable, which is iterating through a list of instances derived from the quantified set `:Presidents` using the function *List-Elements*.

Example:

```

(defpage presidents-container ()
  :optional-inputs
  (:data *presidents-data*)
  :children
  ((:presidents :type 'president
               :quantify (:series (length (the :data)))
               :name (getf (nth (the-child :index)
                               (the :data))
                           :name)
               :term (getf (nth (the-child :index)
                               (the :data))
                           :term)))
  :methods
  ((:write-html-sheet
    ()
    (html (:html
          (:head (:title "Links to Presidents: " (:princ (the :name))))
          (:body (:h1 "Links to Presidents"))

```

```
(:ol
  (dolist (president (list-elements (the :presidents)))
    (html
      (:li (the-object president (:write-self-link))))))))))
```

The `:Write-HTML-Sheet` method in the above page will write an enumerated list, where each item in the list will be a hypertext link leading to the respective President’s actual page instance. The label text to be displayed for the link will default to the `:Strings-for-Display` of the given President object, which defaults to the page name and index number. This default can be overridden as necessary.

## 6 Form Handling

Forms are generated using the HTML “Form” tag and specifying a page instance to *handle* the form submission (the “respondant”), as well as a page instance to contain the new *query-values* from the form (the “bashee”).

These instances, as well as an attribute called the `Instance-ID`, are all included in the form as hidden fields, as follows:

```
(html ((:form :method "post" :action "answer")
  ((:input :type :hidden
    :name "respondant"
    :value (root-path-to-query-arg (the :presidential-display))))
  ((:input :type :hidden
    :name "bashee"
    :value (root-path-to-query-arg self)))
  ((:input :type :hidden
    :name "iid"
    :value (the :iid)))))
```

The above code-snippet would be included in a `:Write-HTML-Sheet` method of a page definition. In this code snippet, the `:response-form` message is a built-in generic URL already defined in all pages defined with *defpage*, which handles form submissions in a generic way.

“Respondant” is specified as

```
(the :presidential-display :root-path-query-arg)
```

which means that when this form’s “Submit” button is pressed, the instance named `:presidential-display` will now present its presentation method to the stream.

Finally, “Bashee” is specified as the `:Armor-Plated-Root-Path` of the current page (“Self”), which means that the current page will have an attribute called `:Query-Values` modified to reflect the new values of fields from the submitted form.

Since `:Query-Values` will have been modified, any other attributes *anywhere* in the page hierarchy which in any way (directly or indirectly) depend on these `:Query-Values` will recompute themselves the next time they are demanded.

`:Query-Values` is a plist containing keywords representing the form field names, and values which will be strings representing the submitted values. In the above example, this plist might be used for passing inputs into a child page, which is the actual `:Presidential-Display` to display a chosen president, as follows:

```
(defpage presidents-form ()
  :children
```

```

((:presidential-display :type 'presidential-display
                        :name (getf (the :query-values) :name)
                        :term (getf (the :query-values) :term)))

:methods
((:write-html-sheet
  ()
  ((:form :method "post" :action "answer")
   ((:input :type :hidden
           :name "respondant"
           :value (root-path-to-query-arg (the :presidential-display))))
   ((:input :type :hidden
           :name "bashee"
           :value (root-path-to-query-arg self)))
   ((:input :type :hidden
           :name "iid"
           :value (the :iid)))

   ((:input :type "string" :name "name" :default ""))
   ((:input :type "string" :name "term" :default ""))
   ((:input :type "submit" :name "OK")))))

```

## 7 Next Steps

### 7.1 Distribution and Licensing

The GWL system is currently open-source software [GWL] released under a license similar to the Apache license agreement. Genworks ([www.genworks.com](http://www.genworks.com)) may in the future make available a commercial version with commercial support, but currently it is unsupported “as-is” software.

GWL currently will compile, load, and run on top of either the ICAD/IDL System from Knowledge Technologies International ([www.ktiworld.com](http://www.ktiworld.com)), or in a “standalone mode” on top of ANSI Common Lisp.

The system is open-sourced for a number of reasons:

- Several enhancements, related to performance and function, are still necessary to make the system viable for large-scale, high-volume website development. Genworks presently does not have the resources to launch such an effort ourselves, so we see open source as a potential means to bring these benefits to the Lisp community at large, including Genworks.
- In the unlikely event that any software techniques being used are patentable, we would like these techniques to be recognized as “prior art in the public domain” to avoid any future proprietary claims on the techniques.
- To the extent that making these systems freely available can expand the use of Common Lisp in general, we feel that this will help all implementations and vendors of Common Lisp to achieve higher volumes and thereby better support, maintenance, and ultimately lower commercial license prices for the commercial implementations.

### 7.2 Application Development

We have several applications in mind for GWL which we feel would leverage its strengths. We would be interested in collaborating with other interested parties in development of such applications for “fun, fame, and fortune.”

- Yet Another Web Calendar
- Cookbook and Recipe Organizer
- Personal Finance Organizer (“Quicken” clone)
- Family Tree builder
- Presentation Builder (“PowerPoint” clone)
- Various Online Catalogs
- Various Web-page Builders and Customized Pages (“My Yahoo” clone)
- and so on and so on...

### 7.3 Performance

GWL performance has been adequate for our small-scale trial deployments thus far. However we suspect that some major optimizations would be necessary to achieve acceptable speed and memory profiles for deployment to large-volume websites.

Here are some examples of potential enhancements/optimizations:

- Root-part instances should time out after a specified period of inactivity (in case the user or group of users who was using that instance simply exits their Web browser, without properly logging out).
- GWL should provide explicit and easy support for Logging Out from an application, thereby setting the corresponding entry in the instance table to *NIL* and freeing up that space for garbage collection.
- Garbage Collection should be enabled to occur more aggressively at low-use periods of time.
- The URL-encoded root-paths should be compressed somehow (currently they can become quite long, increasing the size of the transmitted HTML).

However, before serious effort is put into any of the above optimizations we would like to study some broader usage profiles in order to isolate where we would receive the most “bang” for our time spent on optimization. This is another justification for making the system freely available, i.e. feedback should result in improvement.

## 8 Conclusion

This paper has presented a basic overview of an open-source implementation of GWL, a language embedded in Common Lisp and making use of either the Common Lisp Hypermedia Server or Allegroserve, for modeling web pages using declarative generative objects. By using GWL, an experienced developer can put together a modular web application with dynamic customized pages, forms, etc., in a short period of time and utilizing a minimal amount of source code.

The GWL implementation currently runs on top of KTI’s ICAD/IDL System as well as on ANSI Common Lisp (currently validated on Allegro Common Lisp).

The *Defpage* macro is used in GWL as the basic building block for defining Web page “blueprints,” and these blueprints are instantiated into actual Web pages when they are demanded through the web browser.

GWL supports convenient mechanisms for:

1. Creating hypertext links to navigate through a hierarchy of pages (represented by *Defpage* instances)
2. Handling form submission and managing the resulting query values;
3. Automatically caching computed data on the server to avoid future computation when the page (instance) is revisited
4. Creating and re-using generic computational templates for pages
5. Using the full power of multiple inheritance to define and specialize Web page definitions
6. Interactively and dynamically developing a Web application, as well as being able to patch the server **on the fly** in production without bringing it down and up again

GWL exists as free, open-source software and will continue to do so.

Enhancements to GWL addressing areas like Persistence, Performance, and Memory Usage Patterns will happen with contributions from the user community.

We at Genworks hope that this paper has been informative, and will play a role in encouraging the reader to consider Common Lisp and GWL as a first-choice language for filling a wide variety of needs in the area of Web Application development and beyond.

## References

- [Mallery]                    *A Common Lisp Hypermedia Server*,  
<http://www.ai.mit.edu/projects/iiip/doc/cl-http/home-page.html>
- [Allegroserve]            *Allegroserve Common Lisp Webserver*,  
<http://allegroserve.sourceforge.net/>
- [Graham]                    *ANSI Common Lisp* Prentice Hall, 1996
- [GWL]                        *Generative Web Language* <http://gwl.sourceforge.net>