The contents of this manual describe the product, BDS WEB-PROLOG for Windows Servers (hereinafter referred to as Weblog), version 1.0, and are believed correct at time of going to press. They do not embody a commitment on the part of Brian D Steel (BDS), who may from time to time make changes to the specification of the product, in line with his policy of continual improvement. No part of this manual may be reproduced or transmitted in any form, electronic or mechanical, for any purpose without the prior written agreement of BDS.

Welcome to Weblog

Weblog is the fresh, new toolkit which provides a direct, streamlined way of combining beautifully designed web pages with logical scripts written in that most powerful of all Artificial Intelligence languages, Prolog.

Designed as a replacement for LPA's ageing ProWeb Server, Weblog separates the processes of website design and content programming, so that web pages can be created in familiar tools such as DreamWeaver, PhotoShop and Illustrator, independently of the application code which supports them.

Copyright (c) 2016-2019 Brian D Steel

Designed and Written by Brian D Steel

The "wallpaper" used in the screen shots in this publication is based on the Willow Boughs design by William Morris (1834-96)

Welcome to Weblog 1.0!
# Contents

Weblog 1.0 - User Guide and Reference ........................................................................................................... 2
  Welcome to Weblog ........................................................................................................................................ 2
Welcome to Weblog - Prolog for Web Applications ......................................................................................... 6
  The Full Power of Prolog - on the Web! ........................................................................................................ 6
  About this Manual ......................................................................................................................................... 6
  Enjoy the Software ....................................................................................................................................... 6
Weblog - The Examples ................................................................................................................................. 7
  Weblog is Easy ............................................................................................................................................. 7
  It Just Works ............................................................................................................................................... 7
Example 1 - One Shot Program: "Hello World" .......................................................................................... 8
  Labels .......................................................................................................................................................... 8
  The weblog(0 and submit/2 Predicates ........................................................................................................ 8
  The Basics of Weblog ............................................................................................................................... 9
Example 2 - Input and Backtracking .............................................................................................................10
  Bonjour le Monde ...................................................................................................................................... 10
  The getarg/2 Predicate ............................................................................................................................. 11
  Backtracking ............................................................................................................................................. 12
  Continued Execution .............................................................................................................................. 12
  The Browser Back Button ...................................................................................................................... 13
  Interacting with Weblog ......................................................................................................................... 13
Example 3 - Learning and Storage .............................................................................................................. 14
  Persistent Private Storage ....................................................................................................................... 14
  Extended Form ......................................................................................................................................... 16
  Learning and Storage with Weblog ........................................................................................................... 17
Example 4 - Flexible Output ....................................................................................................................... 18
  John Likes Mary ...................................................................................................................................... 18
  Shedding Light ........................................................................................................................................ 20
  Preformatted Output ............................................................................................................................. 20
  What Next? .............................................................................................................................................. 22
  Flexible Output with Weblog .................................................................................................................. 22
Example 5 - Logic and Design ..................................................................................................................... 23
  Tables and Forms ................................................................................................................................. 23
  Great Oaks From Little Acorns Grow .................................................................................................... 23
The Initialisation File
  timer_mutex
  timer_start
  timer_query
  timer_idle
  timer_temp
  buffer_temp
  buffer_size
  buffer_heap
  error_head
  error_foot
  error_size
  merge_top
  merge_end

Developing a Weblog Application
  Input and Output
  Labels and Forms
  Submit vs Write, Getarg vs Read
  Developing on the Desktop
  Preparing for Web Deployment
  Fresh Start
  Irreversible System Changes
  Time for Deployment
  Deploying Linked Assets
  The Truth Behind Submit
  Interrupted Execution
  Scalability and Security
Welcome to Weblog - Prolog for Web Applications

Welcome to Weblog, a brand new toolkit which provides everything you need to start writing sophisticated AI-based web applications, which can employ and enjoy the full power of BDS WIN-PROLOG, both in its established 32-bit form or in its exciting, new 64-bit release. Your Weblog application code will be 100% compatible with either version, so you can design on 32-bit, deploy on 64-bit, or vice versa, in any combination you like. After this, the software will "just work", whichever platform you choose.

The Full Power of Prolog - on the Web!

The key design goal of Weblog, was to provide you, the systems coder, with a clean, genuine Prolog environment in which to build and test your applications, using back-tracking, assert/retract and pretty much any and all other Prolog language features, while presenting intermediate pages and obtaining user input across the conventional web interface.

Your server installation is fully scaleable, and can include multiple machines each running multiple copies of Prolog; Weblog manages user interactions using secure server-side state files, and permits any instance of the system to pick up the next HTTP submission from the user.

About this Manual

The next few chapters will introduce Weblog with a series of simple, increasingly feature-full demonstration programs, before a main reference section will take you through everything you need to know to get the best out of this package.

Enjoy the Software

We hope you will find much of interest in Weblog, and will take a few moments to read this document before getting underway with your new software package.

Brian D Steel, 01 Mar 2019
Weblog - The Examples

Welcome to Weblog, an exciting new way to create Web applications using the full power of Prolog and its Toolkits. So what's so special about Weblog? Read on and find out!

Weblog is Easy

If you can write it in Prolog, you have already written it in Weblog. Unlike earlier toolkits, Weblog is Prolog, running on the Web: nothing more, nothing less. So big deal, right? Wrong! This really is something special.

It Just Works

There are just two things you need to create a Weblog application: first, a web page, perhaps like this:

```html
<html>
  <head><title>Animals</title></head>
  <body><p>A {:animal:} is a kind of {:type:}</p></body>
</html>
```

And second, some Prolog code, maybe like this:

```prolog
weblog :-
    getarg( animal, A ),
    type_of( A, T ),
    submit( 'animal.htm', [(animal,A),(type,T)] ).

type_of( dog,   mammal  ).
type_of( snake, reptile ).
type_of( crow,  bird    ).
type_of( _,     unknown ).
```

And that's it! You now have a Weblog application that will generate the page shown right.

How? Please read on ...
Example 1 - One Shot Program: "Hello World"

Let's jump right in with the simplest of all examples, a "Hello World!" program. As with all Weblog applications, you need just two things: an HTML page and a Prolog program.

Labels

The HTML page can be as simple or as complex as you like, whether written by hand or using a web development tool such as DreamWeaver. Within the page, you can add Weblog "labels", which act as placeholders for information that will be returned by the Prolog program. Here is an example, "hello_1.htm":

```html
<html>
  <head>
    <title>Hello World Example</title>
  </head>

  <body>
    <h2>{:hello:}</h2>
  </body>
</html>
```

Labels in Weblog are just text names, surrounded by the special brackets, "{:" and ":}"; in the example above, there is just one label, whose name is "hello". When the Weblog application completes its work, this label will be replaced with whatever text the program supplies.

The `weblog/0` and `submit/2` Predicates

Your application can do absolutely anything you want: think "blue sky", and you get the idea. The only fixed thing, is that whatever it does, your program should be called `weblog/0`. 
Just think, "weblog :- anything_at_all.", and you will be in the right zone.

When it comes to displaying the results of your blue sky application, you just call submit/2 to combine your HTML web page with one or more pieces of data that your program has created, pairing your data with Prolog atoms which match the labels on the page.

In the simplest case of all, all that your program needs to do is call submit/2 directly from within your definition of weblog/0, for example:

```prolog
weblog :-
  submit( 'hello_1.htm', [(hello,`Hello World!`)] ).
```

OK, so this is a minimalist program, but it works! The weblog/0 program does nothing other than call submit/2, to display the HTML file, "hello_1.htm", with the label, "{:hello:}" replaced with the text string, "Hello World!".

The Basics of Weblog

So far, you have seen the very basics of Weblog: combining an HTML page with a matched Prolog program, and displaying the result. Next, you'll see how to process input from a web page, and also demonstrate backtracking.
Example 2 - Input and Backtracking

So far, you've seen that a simple Weblog application comprises two files: an HTML web page, and a Prolog program. The HTML file contains special labels, which are replaced with output from the Prolog program, using the submit/2 predicate.

The "Hello World!" application on the previous page, is an example of a "one-shot" program, which simply runs, generates output, and is done. But Weblog can do much more than this; in particular, it can continue execution after the call to submit/2, process further input from displayed web page, even backtracking to provide alternative answers.

Bonjour le Monde

Let's look at a slightly more interesting program, which returns the message, "Hello World!", in several different languages. Here is the Prolog code:

```
weblog :-
    ( greeting( Tongue, Phrase ),
      submit( 'hello_2.htm', [(language,Tongue),(hello,Phrase)] ),
      getarg( choose, Next ),
      Next = done -> submit( 'hello_21.htm', [(hello,`You pressed "DONE": Goodbye!`)]
    ; submit( 'hello_21.htm', [(hello,`No More Answers: Goodbye!`)])
).

greeting( english, `Hello World!`).
greeting( french, `Bonjour le Monde!`).
greeting( german, `Hallo Welt!`).
greeting( italian, `Ciao Mondo!`).
greeting( spanish, `Hola Mundo!`).
```

As before, your program is called "weblog"; this time, however, rather than simply call submit/2 with a hardwired response, "Hello World!", it picks up a language and a phrase a program you have written, called "greeting/2". On the first call, greeting/2 returns the results, "english" and "Hello World!", and both these values are passed into the call to submit/2.
Notice how, unlike the previous example, things don't finish with the call to `submit/2`, but continue. In effect, your program is waiting for the user to view the web page, and then respond to it, by selecting or editing one or more items on a displayed form. But how does execution continue, and how does your Weblog application know what options the user has chosen? The answer, is in another predicate, `getarg/2`.

**The getarg/2 Predicate**

Let's take a look at this example's HTML web page, "hello_2.pl":

```html
<html>
  <head>
    <title>Hello World Example with Backtracking</title>
  </head>
  <body>
    <h2>In {:language:}, you say, "{:hello:}"</h2>
    <p>Please choose your option:</p>
    <form name="" method="get" action="/weblog/hello_2.exe">
      <input type="submit" name="choose" value="more"/>
      <input type="submit" name="choose" value="done"/>
    </form>
  </body>
</html>
```

This time, you will notice there is an additional label, "language", alongside the original "hello" label, but more importantly, there is also a "<form>" element: this is entry which enables your Weblog application to continue after it calls `submit/2`. In this case, the form contains just two buttons (input type="submit"), both named "choose", but with different values, of "more" and "done" respectively.

When you click on either button, your Weblog application picks up where it left off, and you can find out which button was pressed, by calling `getarg/2`. If you click "done", the Weblog application succeeds, and uses `submit/2` to display the message, "You pressed "DONE": Goodbye!", and stops. However, if you selected "more", something more interesting happens ...
**Backtracking**

One of Prolog's special features, is its ability to "backtrack" in order to find subsequent solutions, and of course, this ability is fully supported in Weblog. In this example, if you selected "more", the simple test, "Next = done" will fail, causing the program to backtrack to the next choice point, which will be the next case of "greeting/2". This time, "Tongue" and "Phrase" will be bound to "french" and "Bonjour le Monde!" respectively.

Once again submit/2 will display the "hello_2.htm" file, with the latest result, and once again you will be able to select "more" or "done", and the process repeats.

If you press "more" too many times, so that you run out of cases for greeting/2, the program backtracks into the "else" branch, and displays the message, "No More Answers: Goodbye!", at which point the program stops.

Rather than simply submit an HTML file with a fixed response, "Hello World!", this version of the program has a backtrackable table of languages and their corresponding translations of the greeting. As always, the weblog/0 program is called automatically, and it picks up a greeting, and calls submit/2 to combine the HTML file, "hello_2.pl", with values for both the labels, "{:language:}" and "{:hello:}".

One of Prolog's special features, is its ability to "backtrack" in order to find other solutions, and of course, this ability is fully supported in Weblog. Let's add a "<form>" element to our HTML example, and call the file "hello_2.htm":

**Continued Execution**

So far, so good, but that's not all. Notice how our second example carries on executing after calling submit/2 to show a language and greeting. In fact, it is effectively waiting for you to choose one of the two "<form>" options on the "hello_2.htm" page, "more" or "done".

The call to getarg/2 is all you need to pick up the value of a form field, in this case the one called "choose": if you chose "done", the program prints a polite message which is displayed in a copy of the the old, "hello_1.htm" page, renamed "hello_21.htm"; if you chose "more", the program attempts to backtrack to the next solution for "greeting/2", and execution continues. If there are no further solutions, the program backtracks to the "else" branch, and displays an appropriate goodbye message, again with "hello_21.htm".
The Browser Back Button

One very useful way in which Weblog actually exceeds the features of regular Prolog, is that it responds intelligently to the browser's "back" button. If you make a selection on a form, and then change your mind, simply press "back" and select something else. Or step several levels back, right up to the very start of your application, if that's what you want.

In fact, you are not limited to moving back: you can move both back and forwards as much as you like, using your browser buttons, and take your time deciding where to start interacting with your application, from whatever point you want.

Interacting with Weblog

Now you have seen the essence of how to interact with Weblog: creating an application which displays multiple pages, waiting for your input in between, and controlling its function accordingly. Next, you'll see how to store information and even modify your application while it is running.
Example 3 - Learning and Storage

Over the past few pages, you have learnt about the essence of Weblog, with programs that can carry on executing after calling submit/2, using getarg/2 to obtain data from web forms, allowing your application to respond to input from web pages. But as you've guessed, there is plenty more that Weblog lets you do!

Along with its pattern matching and backtracking behaviour, one special feature makes Prolog ideal in the world of Artificial Intelligence: applications can modify their own programming at run time, to learn from experience, and store information for later reference. Naturally, Weblog inherits these features.

Persistent Private Storage

In Prolog, if you want to change a program during runtime, or maybe just store a little data for later use, you do so with the assert/1 predicate or one of its close relatives. And guess what: that is exactly the same in Weblog! Yes sir, that's right: Weblog fully supports assertions, retractions, and pretty much everything else that you would expect in a Prolog program, with no funny tricks to learn.

An important feature of Weblog, is that while your program can do what it likes in terms of assertion, retraction, perhaps even consulting source files and much more, these runtime changes remain completely private, to you and your individual web session. Even if dozens of people are simultaneously running the same Weblog application, whether on a single server, or cluster of servers, your assertions and retractions only affect your personal web session, and theirs, only theirs.

So let's explore this a little further; here is another version of our "Hello World/Bonjour Le Monde" program:

```prolog
:- dynamic( greeting/2 ).

weblog :-
    (  greeting( Tongue, Phrase ),
        submit( 'hello_3.htm', [(language,Tongue),(hello,Phrase)] ),
        getarg( choose, Next ),
        Next \= more
    -> (  Next = done
            -> submit( 'hello_31.htm', [(hello,`You pressed "DONE": Goodbye!`)] )
```
Just as before, the weblog/0 application picks up a language and phrase from the greeting/2 predicate, before displaying a result with submit/2. Notice, though, that this time, greeting/2 has been declared "dynamic", and among the options that can be returned by getarg/2, is a new one, "teach". Here is the HTML web page for this example, "hello_3.htm":

```html
<html>

<head>
    <title>Hello World Example with Backtracking and Assertion</title>
</head>

<body>

<h2>In {:language:}, you say, "{:hello:}"</h2>

<p>Please choose your option:</p>

<form name="" method="get" action="/weblog/hello_3.exe">
    <input type="submit" name="choose" value="more"/>
    <input type="submit" name="choose" value="done"/>
</form>

</body>
</html>
```
Extended Form

Notice that the "<form>" is a little bigger than before, as it includes two "text" fields, into which you can type a new language and phrase, to "teach" Prolog new cases. The program itself runs much as before, but with an additional case, which asserts your new phrase whenever you press the "teach" option:

Just as with the previous example, this version chooses a greeting from the database, and submits it, before waiting for a response. If you choose "done", the program stops, as before, with a polite message shown in a copy of "hello_1.htm", renamed "hello_31.htm"; however, if you press "teach", it calls {getarg/2} to pick up your new language and phrase, which it then asserts, before starting over, displaying an appropriate message in yet another HTML file, "hello_32.htm":

```html
<html>
<head>
    <title>Hello World Example with Backtracking and Assertion</title>
</head>
<body>

<h2>{:hello:}</h2>
<p>Click here to start over:</p>
<form name="" method="get" action="/weblog/hello_3.exe">
```
Learning and Storage with Weblog

You have now seen all the key features of Weblog, which can be summarised like this:

* Your application program is called `weblog/0`, and is written directly in Prolog

* You call `submit/2` to combine an HTML page with one or more results from your program

* You call `getarg/2` to read values from a web form that has been displayed by `submit/2`

* You can use `assert/1`, `retract/1` and more for persistent storage in your application

On the next page, you will see these ideas adapted still further, to demonstrate some of the flexibility of Weblog output.
Example 4 - Flexible Output

You have now seen some simple examples which built up from an initial, "Hello World" program, through one which can backtrack through solutions, into one which let you add your own cases dynamically. However, all these programs limited their output to simple substitutions, where a label, such as "{:hello:}", was replaced with a phrase such as, "Bonjour le Monde".

With this next example, you will see that you can go further, displaying whatever you like on your web pages. This example of a classic Prolog program demonstrates the ease of use and flexibility of Weblog output.

John Likes Mary

Well, we all knew that, didn't we! From the earliest days of teaching Prolog, the "X likes Y" example has appeared in some form or another in every Prolog text book. And here is our simple version, ready for exploring and modifying on Weblog.

As you found out on the previous page, your Weblog application can use assert/1 and retract/1 to modify its coding at runtime. This next example does likewise, but with a twist: the program being modified is not just a look-up table of greetings that you can extend; it consists of a mixture of facts and rules.

This time, as well as adding more cases to the facts, you can also selectively delete them, and the best bit of all, is that you are shown the whole "X likes Y" program as a listing, just as you would see on a classic Prolog console if you typed the command, "?- listing.", so that you can keep track of what you've been doing. And without further ado, here is the "John Likes Mary" program in full, complete (at last!) with some comments:

    :- dynamic likes/2.

    % call "likes(X,Y)", and show the results one at a time on a web page
    weblog :-
listing( likes ) \rightarrow \text{List},
\begin{align*}
& \text{( likes( X, Y ),} \\
& \hspace{1em} \text{submit( 'likes_1.htm', [(x,X),(y,Y),(list,List)] ),} \\
& \hspace{1em} \text{getarg( choice, done )} \\
& \rightarrow \text{submit( 'likes_2.htm', [(say,'You Stopped Me'),(list,List)] )} \\
& ; \text{submit( 'likes_2.htm', [(say,'No More Results'),(list,List)] )} \\
& ), \\
& \text{what_next.}
\end{align*}

% see if the user wants to add or delete a case, and add it if so

what_next :-
getarg( choice, More ),
\begin{align*}
& \text{( More = add} \\
& \rightarrow \text{getarg( x, X ),} \\
& \hspace{1em} \text{getarg( y, Y ),} \\
& \hspace{2em} \text{asserta( likes(X,Y) )} \\
& \hspace{1em} \text{; More = delete} \\
& \rightarrow \text{getarg( x, X ),} \\
& \hspace{1em} \text{getarg( y, Y ),} \\
& \hspace{2em} \text{retractall( likes(X,Y) )} \\
& \hspace{1em} \text{; More = continue} \\
& ), \\
& \text{weblog.}
\end{align*}

% the classic prolog example: what john and mary like

likes( john, mary ).
likes( john, food ).
likes( mary, wine ).
likes( mary, john ).

likes( john, Something ) :-
likes( mary, Something ).
Shedding Light

As usual, weblog/0 is the application's entry point, and here, the first thing it does is to perform a Prolog "listing" into the variable, "List". Next, it calls the "likes/2" program to generate a solution, and it shows this solution, together with the program listing, using this HTML page, "likes_1.htm":

```html
<html>
  <head>
    <title>John Likes Mary Page 1</title>
  </head>
  <body>
    <h2>John Likes Mary Page 1</h2>
    <p>This is a listing of the "John Likes Mary" program so far:</p>
    <pre>{:list:}</pre>
    <p>And here is a solution:</p>
    <h3>{:x:} likes {:y:}</h3>
    <form name="likes" method="get" action="/weblog/likes.exe">
      <input type="submit" name="choice" value="more"/>
      <input type="submit" name="choice" value="done"/>
    </form>
  </body>
</html>
```

Preformatted Output

By using a "<pre>" element, the program listing is shown, correctly formatted, just as it would be on a Prolog console. Although a very simple illustration, this example shows how easily you can produce any kind of output in Weblog.
The form on this HTML page gives you just two options, "more", which causes backtracking, or "done", which stops generating solutions. Either when you hit "done", or you hit "more", but there are no more solutions, an appropriate message ("You Stopped Me" or "No More Results" respectively) is shown in a second HTML page, "likes_2.htm":

```html
<html>
  <head>
    <title>John Likes Mary Page 2</title>
  </head>

  <body>
    <h2>John Likes Mary Page 2</h2>
    <p>This is a listing of the "John Likes Mary" program so far:</p>
    <pre>{:list:}</pre>
    <h3>{:say:}</h3>
    <p>Now you can add or delete some cases for "X likes Y", or simply continue</p>
    <form name="likes" method="get" action="/weblog/likes.exe">
      <input type="text" name="x" value="" placeholder="type X here"/>
      <input type="text" name="y" value="" placeholder="type Y here"/>
      <input type="submit" name="choice" value="add"/>
      <input type="submit" name="choice" value="delete"/>
      <input type="submit" name="choice" value="continue"/>
    </form>
  </body>
</html>
```
What Next?

After displaying the message as described above, this program calls a new predicate, which has been called "what_next/0". This, in turn, calls getarg/2 to pick up the "choice" field from the form in likes_2.htm.

If the choice returned was "add", the program then calls getarg/2 twice more, to pick up the "x" and "y" fields, and asserts a new case for likes/2 with a call to asserta/1. Similarly, if the choice was "delete", the program again calls getarg/2 to collect the "x" and "y" fields, and attempts to retract the given case with a call to retractall/1. Finally, if the choice was "continue", it does nothing to the likes/2 predicate.

In all three cases, "what_next" starts the whole program over, with whatever modifications have been made, by calling weblog/0.

Flexible Output with Weblog

There we have it: Weblog runs just like regular Prolog, and can be used to display anything you might see on a normal Prolog console; it supports backtracking, dynamic programming, and most anything else you can do in Prolog, with absolutely no special coding tricks required, apart from knowing about submit/2 and getarg/2.

In the pages that follow, you will see some more detailed examples, combining the simplicity of Prolog with web pages created with a variety of packages, including DreamWeaver and Photoshop, to give you a better flavour of what is possible with Weblog.
Example 5 - Logic and Design

All of the examples so far have been displayed on very simple web pages, so that you could concentrate on the programming principles involved, rather than simply look in wonder (or something!) at a carefully crafted web application.

A main aim of Weblog is to simplify the process of making interactive, intelligent web pages, and this example is the first of several which will hopefully start to show you how to embed Prolog logic in web pages that have been designed independently, complete with style sheets and graphics elements.

Tables and Forms

You have seen how easily Weblog can let you display Prolog results in a pre-written web page, and in the previous example, saw a first glimpse of multi-line output, in the form of a program listing. But that is just a hint at the real possibilities.

An important thing to remember, is that you can replace a Weblog label with absolutely anything: a single word or number, maybe a few lines of program listing, or indeed, chunks of Prolog-generated HTML. In this next example, you will explore this further, with the creation of tables and enumerated elements in dynamic forms.

It is worth noting that this example still leaves the design and appearance of the web page to the source HTML files, together with a CSS style sheet and some background graphics. The same program could run in a plain, black-text-on-white-background page, just like the previous examples. The key point is that the design of your pages is under the control of your web designer; the content is created by your Prolog programmer.

Great Oaks From Little Acorns Grow

This Weblog demonstration loads and runs the "FAMILY.PL" example, in which relationships between people are very simply defined as "parent of" or "spouse of", after which all other relationships are computed through a set of rules.

When the program starts, it is pre-loaded with the following family tree:
Consulting Room

You will be able to query this family, and to add and delete names, and change their relationships, in a powerful demonstration of Weblog's ability to run pure Prolog programs across the Web, within a page that has been designed independently. Here is the start of the program, "family_0.pl":

```prolog
% this is the entry point for every weblog application
weblog :-
    consult( family ),
    family_loop( `When you are ready, let's go!' ).
```

As ever, weblog/0 is the automatic entry point for the application: in this case, its first job is to call consult/1 to load in the actual example program, "family.pl", which is a standard part of the WIN-PROLOG release. This is another example of Weblog's powerful ability to modify its own code base at runtime.

Once the example file is loaded, weblog/0 calls "family_loop/1" with some text to display, "When you are ready, let's go!"

Tabling Points

The main family tree program generates tables and option lists (see later) of the known persons, parents and spouses which are part of the "family.pl" example, then calls submit/2 to display these, together with the input message text, in an HTML web page, "family_1.htm". On its continuation, "get_atom/2" is used to retrieve an action, which is then passed into "family/1":

```
/*
 * Simple Example - (c) Brian C. McLaughlin 2008
 */

This view shows a simple set of persons with name and gender, which should all be stored in three, dynamic relations, and known_person/2.

Each person is saved in a clause consisting of their name and gender; one of the parents is saved as their father or mother.

As ever, we have two types of tables to generate:

1. Option lists (one per relation)
2. Tables of persons

To facilitate this, use the relations:

- mother/2 and father/2
- spouse/2
- parent/2
- known_person/2

Each of these relations has a corresponding tabular form, and in the case of known_person/2, an option selection list is also provided.

Tabular data is generated by `option_lists/3` and `make_table/3`.

```
```
% family time - just pick up the required action, and do it

family_loop( Text ) :-
    table( known_person, Person ),
    table( known_parent, Parent ),
    table( known_spouse, Spouse ),
    option( known_person(X, _), X, Option ),
    option( relation(X), X, Relate ),
    submit( 'family_1.htm',
        [ (text,Text),
          (person,Person),
          (parent,Parent),
          (spouse,Spouse),
          (option,Option),
          (relate,Relate)
        ],
    ),
    get_atom( action, Action ),
    family( Action ).

A Case for Going Loopy

The family/1 predicate comprises a series of clauses each of which handles one of known actions, before returning control to family_loop/1 with suitable text:

% add a person, if not already there

family( add_person ) :-
    get_atom( name_a, Name ),
    get_atom( gender, Gender ),
    (  cmp( 0, Name, ' ' )
        -> message( [`please specify a name`], Text )
    ;  known_person( Name, _ )
        -> message( [Name,`already exists`], Text )
    ;  assert( known_person(Name,Gender) ),
        message( [Name,`successfully added`], Text )
    )
family_loop( Text ).

% delete a person, if already there

family( delete_person ) :-
    get_atom( name_d, Name ),
    ( known_person( Name, _ )
    -> retractall( known_person(Name,_) ),
                 retractall( known_parent(Name,_)) ,
                 retractall( known_parent(_,Name) ),
                 retractall( known_spouse(Name,_)) ,
                 retractall( known_spouse(_,Name) ) ,
                 message( [Name,`successfully deleted`], Text )
    ; message( [Name,`does not exist`], Text )
    ),
    family_loop( Text ).

% add a parent, if not already there

family( add_parent ) :-
    get_atom( name_0, Name_0 ),
    get_atom( name_1, Name_1 ),
    ( ( cmp( 0, Name_0, '' )
        ; cmp( 0, Name_1, '' )
        ; cmp( 0, Name_0, Name_1 ) )
    -> message( [`please specify two names`], Text )
    ; relation( Pred ),
        Pred( Name_0, Name_1 )
    -> message( [Name_0,`is a`,Pred,`of`,Name_1], Text )
    ; assert( known_parent(Name_0,Name_1) ),
        message( [Name_0,`successfully added as parent of`,Name_1], Text )
    ),
    family_loop( Text ).
% add a spouse, if not already there

family( add_spouse ) :-
    get_atom( name_0, Name_0 ),
    get_atom( name_1, Name_1 ),
    ( ( cmp( 0, Name_0, '' )
        ; cmp( 0, Name_1, '' )
        ; cmp( 0, Name_0, Name_1 )
    )
    -> message( [`please specify two names`], Text )
    ; relation( Pred ),
        Pred( Name_0, Name_1 )
    -> message( [Name_0,`is a`,Pred,`of`,Name_1], Text )
    ; assert( known_spouse(Name_0,Name_1) ),
        message( [Name_0,`successfully added as spouse of`,Name_1], Text )
    ),
    family_loop( Text ).

% enquire about the family tree

family( enquire ) :-
    table( known_person, Person ),
    table( known_parent, Parent ),
    table( known_spouse, Spouse ),
    get_atom( relate, Relate ),
    get_atom( name_x, Name_0 ),
    get_atom( name_y, Name_1 ),
    ( relation( Relate )
        -> RELATE = Relate
    ; RELATE = _
    ),
    ( known_person( Name_0, _ )
        -> NAME_0 = Name_0
    ; NAME_0 = _
    ),
    ( known_person( Name_1, _ )
        -> NAME_1 = Name_1
    ; NAME_1 = _
    )
Utility Predicates

A number of utility predicates have been used in this example, to simplify the main body of the code, and to isolate an assortment of nitty-gritty requirements. The first, "get_atom/2", is essentially the same as the built-in Weblog predicate, getarg/2, except that it performs some "normalisation" of the resulting atom. In particular, it forces it to lower case, and retains only alphabetic characters together with the underscore, "_":

% get an argument and convert it into a cleaned-up, lowercase atom

get_atom( Name, Atom ) :-
    getarg( Name, Data ),
    stratm( Text, Data ),
lwrupr( Lows, Text ),
strchr( Lows, List ),
findall( Char,
    (  member( Char, List ),
        member( Char, "abcdefghijklmnopqrstuvwxyz_" )
    ),
    Nice
),
atmchr( Atom, Nice ).

The "message/2" predicate takes a list of Prolog terms, and writes them out into a space-separated list, replacing any variables with "???", ready to display as a nicely formatted message:

% make a simple text message from a list
message( List, Text ) :-
    forall( member( Item, List ),
        (  (  type( Item, 0 )
            -> swrite( `???` )
            ;    swrite( Item )
        ),
            swrite( ` ` )
        )
    ) ~> Text.

The "table/2" predicate picks up all known entries of a given type, then sorts them into order, before calling "tr/3" to write out the result in the form of an HTML "<table>" body, with alternate rows swapping the class names, "head" and "tail":

% tabulate the given dynamic predicate into rows of a table
table( Pred, Text ) :-
    findall( (Name_0,Name_1),
        Pred( Name_0, Name_1 ),
        List
    ),
    sort( List, Sort, [] ),
    ...
tr( Sort, head, tail ) ~> Text.

% display a table with alternate head and tail classes

tr( [], _, _ ) :-
    swrite(`~M~J`).

tr( [(Name_0,Name_1)|More], Head, Tail ) :-
    swrite(`~M~J`),
    swrite(`<tr class="`),
    swrite(Head),
    swrite(`">td>`),
    swrite(Name_0),
    swrite(`</td><td>`),
    swrite(Name_1),
    swrite(`</td></tr>`),
    tr( More, Tail, Head ).

The "option/3" predicate does a similar job to table/2, except this time, it write the elements of the sorted list out in the current HTML syntax for an options list in a "<form>":

% tabulate the given call and variable predicate into options for selection

option( Call, Name, Text ) :-
    findall( Name,
        Call,
        List ),
    sort( List, Sort, [] ),
    ( forall( member( Name, Sort ),
        { swrite(`~M~J`),
          swrite(`<option value="`),
          swrite(Name),
          swrite(`">`),
          swrite(Name),
          swrite(`</option>`)

```
swrite( `~M~J` ) ~> Text.
Example 6 - A Picture is Worth

From a simple "Hello World!" example, through the essentials of basic input and output, and onto a worked example, you have learnt about the philosophy behind Weblog. You can express your ideas in clean, genuine Prolog, and display them with whatever you can imagine on the Web.

The previous example, "Family Tree", went further than the initial few programs, incorporating a CSS style sheet for the first time, and generating entire HTML elements, such as "<table>" bodies and ""<option>" lists for forms. But as ever, there's more that you can do with Weblog!

A Thousand Words

Just as you can generate HTML elements of your own, to "drop in" to your separately designed web pages, you can use similar techniques to embed appropriately chosen graphics diagrams and photographic images. It is no more complicated than figuring out, from within your Prolog program, which file(s) you want to display, and using submit/2 to pass its name to a suitable placed Weblog label in your HTML page.

Loony Moony

One of the Prolog examples that has been around for many years, is a simple program that computes the dates and phases of the moon. When you run it, from the Prolog command line, it checks the current date, and then uses modulo arithmetic to compute the date of the next quarter phase, returning that date and the name of the phase. On backtracking, the program returns the following date and quarter phase, and so on.

The Original Program

Here is the entire source code of the original, Prolog example program, "lunar.pl":

```prolog
% compute the quarter phase of the moon on or after the current date

lunar( Phase, Year, Month, Day ) :-
    time( 1, Time ),
    Time = (Date,_,),
    time( From, 1900, 1, 1 ),
```
lunar( From, Date, Phase, Year, Month, Day ).

% compute the quarter phase of the moon on or after the given date

lunar( From, Date, Phase, Year, Month, Day ) :-
    Lunar is 29.53059,
    Quart is Lunar / 4,
    Days is Date - From,
    Stage is Days mod Lunar,
    Moon is ip(Date + Quart - Stage mod Quart),
    Entry is ip(Stage * 4 / Lunar) + 1,
    ( time( Moon, Year, Month, Day ),
        mem( [ 'First Quarter',
            'Full Moon',
            'Third Quarter',
            'New Moon' ], [Entry], Phase )
    ; Next is Moon + 1,
    lunar( From, Next, Phase, Year, Month, Day )).

Prolog Command Line

The program works from the knowledge that there was a full moon on the very first
day of the 20th Century (01 Jan 1900), and that the lunar month is roughly 29.53059
days in length. Using Prolog's built-in time and date functions, it is easy to figure out
where successive quarters fall. Here is how it looks when running from the Prolog
command line:

    | ?- lunar( P, Y, M, D ).
P = 'Full Moon' ,
Y = 2016 ,
M = 7 ,
D = 19 ;
P = 'Third Quarter' ,
Driving the Moon

So the lunar phase program already exists, and has been part of the Prolog distribution for many years. All you now need, to run this on the Web, is a little bit of Weblog to wrap it up. Here goes, with the whole of the source file, "lunar_0.pl":

```
% call the lunar example and display the result

weblog :-
  ( lunar( Phase, Year, Month, Day ), % generate lunar phase and date
    lunar_image( Phase, Image ), % compute image file phase name
    submit( 'lunar_1.htm', % show the data on an html page
      [ (image,Image),
        (phase,Phase),
        (y,Year),
        (m,Month),
        (d,Day) ]
    ),
    getarg( user, User ), % see what the user wants to do next
    User \= 'next phase' % fail if the user wants next phase
  -> ( User = 'reset moon' % start over to reset the moon
    -> weblog
```
As you know, your program is always called "weblog", and here, all it does is call the lunar/4 predicate from the standard Prolog example, then look up the name of a PNG file that represents the phase graphically, before submitting a HTML web with the data from the example and the name of the matching image. And here is that page, "lunar_1.htm":

```html
<html>
<head>
<title>Loony Moony</title>
<meta http-equiv="content-type" content="text/html;charset=iso-8859-1"/>
<link rel="stylesheet" type="text/css" href="/lunar.css"/>
</head>

<body bgcolor=#ffffff leftmargin=0 topmargin=0 marginwidth=0 marginheight=0>
<table width=960 border=0 cellpadding=0 cellspacing=0>
<tr>
<td colspan=3 background="/images/lunar_1.png" width=960 height=117></td>
</tr>
<tr>
<td rowspan=4 background="/images/lunar_2.png" width=193 height=563></td>
<td background="/images/lunar_3.png" width=250 height=250><img src="/images/{:image:}" width=250
```

Graphic Inserts

As you know, your program is always called "weblog", and here, all it does is call the lunar/4 predicate from the standard Prolog example, then look up the name of a PNG file that represents the phase graphically, before submitting a HTML web with the data from the example and the name of the matching image. And here is that page, "lunar_1.htm":

```html
<html>
<head>
<title>Loony Moony</title>
<meta http-equiv="content-type" content="text/html;charset=iso-8859-1"/>
<link rel="stylesheet" type="text/css" href="/lunar.css"/>
</head>

<body bgcolor=#ffffff leftmargin=0 topmargin=0 marginwidth=0 marginheight=0>
<table width=960 border=0 cellpadding=0 cellspacing=0>
<tr>
<td colspan=3 background="/images/lunar_1.png" width=960 height=117></td>
</tr>
<tr>
<td rowspan=4 background="/images/lunar_2.png" width=193 height=563></td>
<td background="/images/lunar_3.png" width=250 height=250><img src="/images/{:image:}" width=250
```
The "<table>" element that makes up the bulk of the above HTML file, was originally generated in Adobe Photoshop, by chopping up an original image into multiple slices, with the main purpose of isolating the area which displays the moon. Once this was done, four different versions of the "moon" slice were prepared, one for each of the four lunar phases.
The HTML output from Photoshop was hand-edited, both to replace the entry for "lunar_3.png" with the Weblog label, "{:image:}" and to add the all-important "<form>" element, which is what provides interaction with the Weblog application.

After this application calls lunar/4 to compute the next phase of the moon, it this program looks up the appropriate image file name in "lunar_image/2". This name is then passed to submit/2, along with values for date and phase name returned by the original example. As usual, the named HTML web page, "lunar_1.htm", is shown with the appropriate substitutions, this time, including the name of a .PNG file, within an "<img>" element, which displays the image in context.

Photoshop is just one of many well-established graphics design and publishing programs, which works very well alongside Weblog.
Example 7 - Perfect Picture

Here is another example created with the help of a well-known graphics application, this time Adobe Illustrator, together with help from Photoshop and WIN-PROLOG, all three being used to create the visual components of the application interface. Again, the logic, written in Prolog, is independent of the graphic design.

As with the previous "Lunar Phase" program, the bulk of the HTML page that you see was created in Photoshop, and sliced into a number segments, this time to isolate the West Clock Face of the famous Elizabeth Tower at the Palace of Westminster.

Tempus Fugit

As with the Lunar Phase program you saw on the previous page, this application is based on a standard Prolog example, which displays randomly chosen phrases to represent, approximately, the current time. The example itself was inspired by broadcasters at BBC Radio 4, who routinely give time checks that are out by an hour or more, as they read an analogue or digital clock, and try to put the time into words.

Again, as with the Lunar Phase program, the interface is based on an original image, that was sliced in Photoshop to generate the main body of the HTML web page. This time, rather than isolating the moon, it was the West-facing clock face that was placed into its own slice, for runtime substitution.

This time, as well as displaying text phrases that approximate the current time, the idea was to display the time on the clock face, using outlines of the actual clock hands. The approach taken was to be to pre-compute an image for every minute of a 12-hour half day, and then generate the appropriate image file at run time. Unfortunately, this meant creating no fewer than 720 separate .PNG image files!

All Hands on Deck

One of the standard WIN-PROLOG examples, is an analogue lock display, "big_ben.pl", based on the face of the Westminster clock. This example contains a predicate, "show/2", which uses a background image of the blank clock face, which was created
in Adobe Illustrator, onto which it draws the hour and minute hands, drawn by Prolog's gfx/2 graphics predicate.

The outlines of the hands are also based on the Illustrator file, but were converted into Prolog coordinates: they are drawn rotated by the appropriate amounts for any given time.

In order to create the required 720 individual files, a WIN-PROLOG program was written to increment the hours from 0-11 inclusive, and for each such hour, to increment the minutes from 0-59. For each combination of the hands, the hands were displayed, without the clock face background, and a selective screenshot taken and dumped into a .BMP bitmap file, using WIN-PROLOG's bdsbox/2 predicate.

Once the program had run through all the hour and minute combinations, the 720 resulting bitmap files were converted to .PNG files using a batch job in Photoshop, which replaced the white background with transparency, and then reduced the images from their initial 1100*1100 pixels size, to 200*200 pixels. It is one of the 720 "hands" files that is subsequently displayed on the blank, Illustrator-generated clockface on the sliced image of the Elizabeth Tower, with the file name being computed from the 12-hour time.

Tick Tock Timex

The "Broadcast Speak Clock" program, "timex.pl", is a standard, non-graphical Prolog example. All it does is read the current time, and then generate, randomly, different English phrases which represent the time. For example, from the Prolog console, you might type:

```
| ?- clock( Q, W ).
Q = `10:08:42`,
W = `some time after eight past ten`;

Q = `10:08:42`,
W = `a while after ten oh eight`;

Q = `10:08:42`,
W = `heading towards ten oh nine`
```

Rock Around the Clock

Just as with the previous, "Lunar Phase" program, all this application requires to run on the Web, is a little bit of Weblog. Here is the contents of the file, "timex_0.pl":

```
% call the timex example and display the result
weblog :-
```


{ repeat, % repeat so that cases never run out
  clock( Digits, Words ), % generate time and a random phrase
  cat( List, Digits, [2,1,2] ), % compute image file name from time
  List = [Hour24,_,Minute,_],
  fread( r, 2, 10, Int24 ) <~ Hour24,
  Int12 is Int24 mod 12,
  fwrite( r, 2, 10, Int12 ) ~> Hour12,
  submit( 'timex_1.htm', % show the data on an html page
    [ (digits,Digits),
      (hour,Hour12),
      (minute,Minute),
      (words,Words)
    ],
  ),
  getarg( user, User ), % see what the user wants to do next
  User \= 'other words' % fail if the user wants other words
  -> ( User = 'update time' % start over to update the time
    -> weblog
    ; User = 'done', % quit program if all done
      submit( 'timex_2.htm', [] ) % show the goodbye message
  )
).

Graphic by Numbers

The main difference between this application and the Lunar Phase example, is that the file name is computed from the 12-hour time, rather than being looked up from a database relation, which would otherwise require 720 clauses! The computed data is combined with an HTML file with the usual call to submit/2.time itself; here is that file, "timex_1.htm":

```html
<html>
<head>
<title>Big Ben Says</title>
<meta http-equiv="content-type" content="text/html; charset=iso-8859-1"/>
<link rel="stylesheet" type="text/css" href="/timex.css"/>
</head>
```

Weblog 1.0 - User Guide and Reference
<table width=960 border=0 cellpadding=0 cellspacing=0>
  <tr>
    <td colspan=3 background="/images/timex_1.png" width=540 height=100></td>
    <td rowspan=3 background="/images/timex_2.png" width=420 height=720>
      <h2 align="center">at {:digits:} Big Ben says</h2>
      <h3 align="center">{:words:}</h3>
      <form align="center" name="timex" method="get" action="/weblog/timex.exe">
        <input type="submit" name="user" class="azure" value="update time"/>
        <input type="submit" name="user" class="azure" value="other words"/>
        <input type="submit" name="user" class="azure" value="done"/>
      </form>
    </td>
  </tr>
  <tr>
    <td rowspan=2 background="/images/timex_3.png" width=160 height=620></td>
    <td background="/images/timex_4.png" width=200 height=200>
      <img src="/images/tim_{:hour:}{:minute:}.png"/>
    </td>
    <td rowspan=2 background="/images/timex_5.png" width=180 height=620></td>
  </tr>
  <tr>
    <td background="/images/timex_6.png" width=200 height=420></td>
  </tr>
</table>
Example 8 - Visual Rules

Throughout these pages, you will have noticed the stress on the key focal point, that Weblog is full Prolog which is able to run across the Web. The appearance of your Weblog applications is totally separate from the logic that drives them, so that you can design pages any way you like, and then create the content using uncluttered, natural code.

What has not been stressed, so far, is the corollary: a happy side-effect of this true "prologness", is that other toolkits and systems, written in Prolog, will also run naturally on Weblog. One such toolkit is VisiRule 2 …

VisiRule 2

VisiRule 2 (VR2) is a prototype replacement for the current VisiRule business rules program toolkit; unlike the current version, VR2 compiles directly down to Prolog, and consequently, runs very happily in Weblog.

Taking an existing VisiRule example, modified and compiled in VR2, this present application runs a version of the "Lawns" expert system, which aims to advise you on how to improve that patch of grass in your rear garden.

There is not a lot to say about the application itself, other than that it was compiled directly from the VR2 chart shown below. The Lawns Advisor comprises a series binary questions about the state of your lawn, asking you questions and following through the logic to an appropriate conclusion, which is returned as a Prolog string. It is the text of these questions and the final string which is shown on the web pages.

Weaving the Dream

Apart from an advance glimpse of VisiRule 2, the other feature of this example is that its web page was generated in one of the most popular website development programs, Adobe DreamWeaver.

DreamWeaver is very much like a desktop publishing program, enabling you to play with website designs in a "WYSIWYG" fashion, without every having to get into direct HTML editing, unless you particularly wish so to do. When you want to place a Weblog "{: label:}"{}, all you need to do is type its name in between those curly/colon brackets.
The actual HTML generated by DreamWeaver tends to be a little messy to read in its raw form, so on this occasion, the file will not be listed here. However, the file is essentially in the same format as all the other examples you have seen, complete with a "<form>" element which provides the interaction with Weblog.

A Modicum of Help

As noted above, VisiRule 2 is currently in a prototype state, and as such, does not yet have a complete web interface. However, as this particular example, the Lawns Advisor, is relatively simple, it took only a couple of special predicates to be written to get it running. First up is weblog/0, which here does nothing more than call the compiled, VR2 program, "lawn/3", before displaying the response with submit/2.

% call the grass example and display the result

weblog :-
    lawn( start, [], Answer ),
    submit( 'grass_2.htm', [(answer,Answer)] ).

And here is a part which will not be necessary in due course, providing a "dialog" interface which calls submit/2 to display the query page, followed by getarg/2 to collect the response, and a "button" interface which enumerates the choices offered.

% simplified response dialog

vsr_dialog( [], [], Answer, _, Answer ).

% simplified question dialog

vsr_dialog( [], Choices, (Name,Question), _, Response ) :-
    vsr_buttons( Choices, Buttons ),
    submit( 'grass_1.htm',
        [ (buttons,Buttons),
          (name,Name),
          (question,Question) ]
    )

getarg( response, Response ).

% convert a list of options into html

vsr_buttons( Choices, Buttons ) :-
  forall( member( Item, Choices ),
    ( swrite( `<input type="submit" name="response" value="` ),
      swrite( Item ),
      swrite( `"/>~M~J` )
    )
  ) ~> Buttons.
**Weblog 1.0 - Reference**

**Introduction**

Weblog is a powerful Web interface for **WIN-PROLOG**. It is intended as a replacement for the ageing ProWeb system, and is easier to integrate into existing web designs than is earlier product.

The basic tenet of Weblog, is that web designers can create web pages using whatever technology best suits them, and then drop in components that have been computed at runtime by Prolog. This is different to ProWeb, which attempted to create an old form of HTML from Prolog structures and partial, template files.

Weblog natively supports nearly all Prolog features, including true backtracking, native assert and retract, and much more, none of which was available to ProWeb.

**Loading the Application**

In the present configuration, Weblog is configured to load either a .PL or .PC file of the same root name as the executable file, from the same folder. You can rename ALL the Weblog files to any other, single root name that you want: the .EXE file will loads the .SYS file, which in turn loads the .OVL and .OVX files. Finally, once-off for any given web query, an "ensure_loaded" call will be made to load either the .PL or .PC file of the same root name, whichever is present or newer.

By loading you application at runtime, rather than building into an overlay, debugging and code tweaks is made very simple and fast.

**The Basics**

Minimally, a Weblog application consists of a Prolog program, whose name is "weblog", with no arguments. Though not very useful, even this example would work, returning the words, "Hello World" to the calling browser screen:

```prolog
weblog :-
    write(`Hello World`).
```

In the above, vestigial example, the output is not even encoded as HTML, but simply recognised by the browser as plain text. However, it serves as an initial illustration that Prolog is running as "normal".

Of course, further calls to `write/1` could have been used to add HTML tags, but Weblog has a better way to display web pages: you design the page you
want, and place labels in it where you want Prolog output to be included. Consider this very simple web page:

```html
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <h2>{:hello:}</h2>
  </body>
</html>
```

This page is 100% standard HTML, and could be created with any HTML editor or web design tool; it also includes one special item, a label with the name, "hello". By default, labels are delineated by a pair of tokens, "{:" and ":}" , which are just plain text to HTML. This page can be output by Weblog, using the predicate, submit/2. Suppose the above file is called "file.htm"; the following program would display the file, with the message, "Hello World" displayed as an "h2" header:

```
weblog :-
  submit( 'file.htm', [(hello,`Hello World`)]) .
```

**Continuation**

The submit/2 predicate sends information to a web browser, and also stops execution of the Prolog program, which can then be unloaded from memory on the server. However, it is also possible to get a program to continue after submitting a page, simply by writing further calls.

In order to continue, a submitted page must contain at least one "<form>" element, with one or more input fields to allow the user to click when ready to continue. Consider this extended version of "file.htm", this time called "file_0.htm":

```html
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <h2>{:hello:}</h2>
    <form name="" method="get" action="example.exe">
      <input type="submit" name="foo" value="more"/>
      <input type="submit" name="foo" value="done"/>
    </form>
  </body>
</html>
```
This page includes a form, with two input fields, both called "foo", but with different values, "more" and "done". We can find out which one was selected after calling submit/2, using the predicate, getarg/2. Here is a longer version of our simple example program:

```prolog
weblog :-
    submit( 'file_0.htm', [(hello,`Hello World`)] ),
    getarg( foo, Atom ),
    ( Atom = more -> submit( 'file_0.htm', [(hello,`You chose MORE`)])
    ; Atom = done -> submit( 'file_0.htm', [(hello,`You chose DONE`)])
).
```

**Backtracking**

With a combination of submit/2, getarg/2 and simple tests on the returned value, Prolog can easily be made to backtrack if desired. Consider this version of the same program:

```prolog
weblog :-
    ( salutation( Text ),
      submit( 'file_0.htm', [(hello,Text)] ),
      getarg( foo, Atom ),
      Atom = done,
      submit( 'file_0.htm', [(hello,`Bye Bye`)])
    ; submit( 'file_0.htm', [(hello,`No More Solutions`)])
  ).

salutation( `Hello World` ).
salutation( `Bonjour le Monde` ).
salutation( `G'day mate` ).
```

In this program, a "salutation" is picked up from a Prolog data relation, and this is submitted with our web page. The user can view the result, and then click either "more" or "done". Following a call to getarg/2, a test is made to see what value was chosen, and if it was "done", a "Bye Bye" message is submitted on the page. If the match fails (here, "more" is the only alternative), the program backtracks into salutation/1, and finds the next solution, if any. This is
submitted, and once again, getarg/2 picks up the choice, and so on.

If, on backtracking, there turns out to be no more solutions, the entire "or" branch of the disjunction fails, and "No More Solutions" is submitted instead.

**Assertion and Retraction**

As well as supporting backtracking, Weblog fully supports the use of assert/1, retract/1, abolish/1, dynamic/1, as well as the low-level addcls/2, delcls/2, addprd/2, delprd/2, and so forth. Even though the application can unload from memory between web pages, all changes to the program database work exactly as they do in normal Prolog. Files can be consulted, or loaded as .PC files, and simply become part of the ongoing query image.

**File Handling**

There are very few limitations on what can be done in Weblog; possibly the only significant one concerns file handling: any and all open files must be closed down around calls to submit/2, although they can be reopened immediately after, if required.

This is not really a "limitation" as such, but more, a case of data integrity and sharing. When your program calls submit/2, it is unloaded from memory the instant it has sent your next web page to the browser. Any open files will simply be closed automatically. On resumption, the reason the files cannot simply be opened again automatically, is that they may have disappeared or changed in the meantime, if some other application has been at them. Indeed, the resumption might even be running on a totally different server than the previous page, and the files may not be physically present.

Weblog applications can use files just like any other Prolog application, but in most cases, should do so in an "Open, Process, Close" basis, rather than "Open, and leave open for convenience" model.

**Error Handling**

Weblog programs run without an encompassing "catch/n" wrapper, so when an error occurs, the program stops abruptly, and reports the problem to the browser interface. Of course, your code can use catch/n, as well as traditional Prolog error handling, but in many cases, it is more instructive not to do so, since the displayed error messages are more detailed than those available in ProWeb.
The Predicates

We have already discussed the three primary predicates of Weblog, but there are a handful of others that can be useful too. Here is the full list of Weblog predicates:

weblog

<No arguments>

User-defined application; simply, the entry point to your program. You can implement anything, so long as the entry point has this name.

submit( File, List )

File - Atom naming the HTML file
List - List of (atom,term) pairs

Standard way to submit a web page. Each atom in the List is the name of one or more labels in the File (labels can occur more than once in a file); any term type can be supplied, although strings and atoms will be the most usual types.

argenv( Args, Envs )

Args - Variable
Envs - Variable

The Args variable returns a list of all known argument names, and Envs returns a similar list of all known environment string names. All items in each list are atoms; their individual values can be obtained by getarg/2 and getenv/2 respectively.

getarg( Name, Atom )

Name - an atom naming an argument
Atom - a variable to return, or atom to match an argument value

This gets or matches the value of the argument with the given Name. If Atom is specified as a variable, it returns the argument as an atom. If it is specified as an atom, the predicate succeeds or fails depending upon whether it matches.
getenv( Name, Atom )

Name      - an atom naming an argument
Atom      - a variable to return, or atom to match an environment string

This gets or matches the value of the environment string with the given Name. If Atom is specified as a variable, it returns the environment string as an atom. If it is specified as an atom, the predicate succeeds or fails depending upon whether it matches.
The Initialisation File

The initialisation file lives in the same folder as the Weblog executables, and is used to tailor the behaviour of application at runtime. Special WIN-PROLOG memory settings can be stored in the "[pro386w]" section, as described in the WIN-PROLOG user manual; further Weblog-specific settings can be stored in the [weblog] section of the same file. Here is a listing of the standard WEBLOG.INI file:

```
[pro386w]

[weblog]
timer_mutex=5
timer_start=5
timer_query=5
timer_idle=60
timer_temp=3600
buffer_temp=c:\temp
buffer_size=65536
buffer_heap=4
error_head=Weblog Error %s %s %u
error_foot=Please Try Again
error_size=80
merge_top={
merge_end=:
```

Here is a brief description of each entry:

**timer_mutex**

Default: 5

Sets the time-out, in seconds, for mutex creation.

**timer_start**

Default: 5
Sets the time-out, in seconds, for Prolog start-up.

**timer_query**

   Default: 5

Sets the time-out, in seconds, for query execution.

**timer_idle**

   Default: 60

Sets the time-out, in seconds, for idle time.

**timer_temp**

   Default: 3600

Sets the time-out, in seconds, for temporary file lifetimes.

**buffer_temp**

   Default: c:\temp

This names a read/write folder, into which all automatically created temporary files are placed. Any files older than the value indicated in timer_temp, are automatically deleted whenever Weblog runs.

**buffer_size**

   Default: 65536

This is the size, in bytes, of the shared buffer that will be used to pass information from the CGI front end and Prolog back end.

**buffer_heap**

   Default: 4
This is the number of instances of the application which should be allowed to be running concurrently.

**error_head**

Default: Weblog Error %s %s %u

The text which forms the header of any error report, using the "<h1>" tag; the three format fields show, respectively, a three-letter acronym, a three-digit instance suffix, and the return code from the Prolog system.

**error_foot**

Default: Please Try Again

The text which forms the prompt/apology at the end of any error report, using the "<h2>" tag.

**error_size**

Default: 80

The number of characters to display from any error or console output, using the "<code>" tag.

**merge_top**

Default: {:}

This entry shows the token that starts a label; it can be any sequence of characters, but it is recommended to use something easily visible in source HTML files, and which has no syntactic significance.

**merge_end**

Default: {;}

This entry shows the token that ends a label; it can be any sequence of characters, but it is recommended to use something easily visible in source HTML files, and which has no syntactic significance.
Developing a Weblog Application

Weblog applications comprise two main parts: some Prolog code to define the logic of the application, and some HTML and optionally other design elements, such as graphics and style sheets, to specify the application's appearance. Initially, let's look further at the Prolog considerations.

Input and Output

As has been noted already, Weblog programs are effectively just Prolog programs, and yet there is one key difference: how user input and output is performed. Simple, traditional examples of interactive Prolog, make use of `write/1` to display information and prompts, and `read/1` to retrieve responses from the user, for example:

```prolog
my_program :-
  write( `What is your hair colour? ` ),
  read( Hair ),
  write( `What is your eye colour? ` ),
  read( Eyes ),
  write( `So you have ` ),
  write( Hair ),
  write( ` hair and ` ),
  write( Eyes ),
  write( ` eyes!` ),
  nl.
```

If you type this into a window and compile it, you can then run it, something like this (user responses shown in bold):

```
| ?- my_program.
Where is your hair colour? |> blonde.
Where is your eye colour? |> blue.
So you have blonde hair and blue eyes!
yes
```

Labels and Forms

With weblog, on the other hand, you perform all input and output by submitting HTML pages which contain forms, that in turn, return information to your application. You could, for example, write the following HTML page, called (say) "colour_0.htm":

```html
my_program :
  write( `What is your hair colour? ` ),
  read( Hair ),
  write( `What is your eye colour? ` ),
  read( Eyes ),
  write( `So you have ` ),
  write( Hair ),
  write( ` hair and ` ),
  write( Eyes ),
  write( ` eyes!` ),
  nl.
```
<html>
  <head>
    <title>Colour Sheet</title>
  </head>
  <body>
    <h2>Colour Sheet</h2>
    <p>What is your {:type:} colour?</p>
    <form name="colour" method="get" action="/weblog/colour.exe">
      <input type="text" name="colour" value=""/>
      <input type="submit" name="choice" value="enter"/>
    </form>
  </body>
</html>

Two key things to notice here, are the presence of a Weblog "label", denoted by the symbols "{" and ":"}, and so in this case, named "type", and a form with two fields, the first a text box named "colour", and the second, a button named "choice". Viewed directly in a browser, this file will display the label, "{:type:}", literally, as shown in the image to the right.

Now consider the previous program, but modified to replace write/1 and read/1, the Weblog predicates, submit/2 and getarg/2 respectively:

```prolog
weblog :-
  submit( 'colour_0.htm', [(type,hair)] ),
  getarg( colour, Hair ),
  submit( 'colour_0.htm', [(type,eye)] ),
  getarg( colour, Eyes ),
  submit( 'colour_1.htm', [(hair,Hair),(eyes,Eyes)] )
```
The first call to `submit/2`, displays our HTML file, "colour_0.htm", replacing the label, "{:type:}", with the word, "hair", and displaying a text box into which you can type a response, say, "blonde".

When the form has been typed into, and the "enter" button clicked, the first call to `getarg/2` is used to retrieve the value typed into the form entry called "colour", which is the name of the text box.

Now we repeat the call to `submit/2`, this time passing the word "eye" as the "{:type:}", and once the user as entered something into the text box, say "blue", we repeat the call to `getarg/2`, once again retrieving the colour.

The remaining stage assumes we have another HTML page, called (say) "colour_1.htm", and perhaps looking like this:

```html
<html>
  <head>
    <title>Result Sheet</title>
  </head>

  <body>
    <h2>Result Sheet</h2>
    <p>So you have {:hair:} hair and {:eyes:} eyes!</p>
  </body>
</html>
```

The third and final call to `submit/2`, displays this second html file, with the label, "{:hair:}" replaced by the first value typed in by the user ("blonde"), and the label, "{:eyes:}" replaced by the second value, ("blue"), resulting in the browser view shown below.
Submit vs Write, Getarg vs Read

The submit/2 predicate can be thought of as little more than a simple "merge" routine, that takes a boilerplate HTML file, and merges a list of zero or more (label,value) pairs into it, displaying the result. For now, think of it as some kind of "write these values using a given template" function.

Similarly, getarg/2 can be thought of as little more than a routine to scan a filled-in HTML form, and return the value named by its first argument. Think of it as some kind of "read this value from the given form" function.

Indeed, logically, that is exactly how submit/2 and getarg/2 work, and so it becomes easy to see how Weblog programs are effectively just Prolog programs, albeit with different standard input and output predicates.

Developing on the Desktop

A small source file, "WEBLOG_X.PL", is included with Weblog, which provides simplified simulations of submit/2 and getarg/2, as well as the other runtime Weblog predicates. It makes no attempt to render HTML pages to the screen (this could be added as a feature in the future), nor to display form option names; however, it does let us run a Weblog program directly in WIN-PROLOG while developing the logic.

Let's see how it behaves with our program so far; first, we'll compile the example, saved (say) in a file called "colour.pl", and then load the simulator, "WEBLOG_X.PL", from the "WEBLOG" subdirectory:

```
| ?- [colour].

# 0.000 seconds to consult c:\...\colour.pl
yes

| ?- [prolog('weblog\weblog_x')].

# 0.000 seconds to consult c:\...\weblog\weblog_x.pl
yes
```

The simulator file includes a start point, called run/1, which allows you to pass in a list of (name,value) pairs as if from an initial form in an HTML file, that
might be used to initialise your application. Here, there are no initial values, so an empty list is used:

| ?- run( [] ).

The `run/1` predicate performs some simple initialisations, and then calls `weblog/0`, the user-defined entry point to every Weblog application. The first thing our example does, is call:

```
submit( 'colour_0.htm', [(type,hair)] ),
```

This results in the following output:

```
Showing file "colour_0.htm"

<table>
<thead>
<tr>
<th>Label</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>hair</td>
</tr>
</tbody>
</table>
```

Rather than attempt to render the file, "colour_0.htm", the simulator version of `submit/2`, simply displays the name of the file that would be shown, followed by a table of label names and their values, as would be substituted in the deployed system (here, "{:type:}" becomes "hair").

The next call in our program, is:

```
getarg( colour, Hair ),
```

In the deployed application, this call would directly obtain the value of the "colour" field in the form displayed by the previous call to `submit/2`; here, there has been no form, so instead, it displays a prompt and waits for the user to respond:

```
Please enter the value of "colour"
-> blonde
```

Note that when using the simulator, you do not need to finish your inputs with a period ("."): indeed, you can include spaces and punctuation, without worrying about Prolog syntax. Think of this as what you would type into an actual web form.

Now there is a second call to `submit/2`, and indeed a second call to `getarg/2`:

```
submit( 'colour_0.htm', [(type,eye)] ),
```
getarg( colour, Eyes ),

The simulator will handle this pair of calls as following, pausing to allow the user to respond to the new question:

Showing file "colour_0.htm"

<table>
<thead>
<tr>
<th>Label</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>eye</td>
</tr>
</tbody>
</table>

Please enter the value of "colour"
-> **blue**

All that is left to do is make one final to `submit/2`, this time to display our second HTML file, "colour_1.htm", showing two values:

```prolog
submit( 'colour_1.htm', [(hair, Hair), (eyes, Eyes)] ).
```

The simulator will show the following output:

Showing file "colour_1.htm"

<table>
<thead>
<tr>
<th>Label</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>hair</td>
<td>blonde</td>
</tr>
<tr>
<td>eyes</td>
<td>blue</td>
</tr>
</tbody>
</table>

yes

And that's it: our desktop simulation of the program is complete.

**Preparing for Web Deployment**

The example we're working with is intentionally simple, to the point of being simplistic. Not least, any half-decent web application would display a larger form, asking for all the information at once, rather than successive pages asking for one item at a time. But that's for later. For now, let's see how to save this example as a Web application.
Fresh Start

For starters, we need to replace the "WEBLOG_X.PL" simulator with the real Weblog system. The best approach is to save your source Prolog files (in this case, "colour.pl"), quit from WIN-PROLOG, and then start a fresh session - guaranteeing you avoid saving all manner of test cases, random compile/trace modules, etc., in your web application image.

So fire up WIN-PROLOG, load in your application file, and the Weblog system file:

```
| ?- [colour].
# 0.000 seconds to consult c:\...\colour.pl
yes
| ?- load_files( prolog('weblog\weblog') ).
yes
```

Now all we have to do is perform a little magic with a predicate, so far not mentioned, \texttt{weblog/1}:

```
| ?- weblog( colour ).
```

Within half a second or less, WIN-PROLOG will have quit - but will have left four brand new files, called (in this case):

- \texttt{COLOUR.EXE} - copy of "WEBLOG.EXE", which is the CGI interface for Weblog
- \texttt{COLOUR.OVL} - copy of "WEBLOG.OVL", which is the CGI overlay for Weblog
- \texttt{COLOUR.SYS} - copy of your WIN-PROLOG executable file (PRO386W.EXE or PROX64W.EXE)
- \texttt{COLOUR.OVX} - a special "state file", containing your Weblog application

Irreversible System Changes

The reason why WIN-PROLOG quit, is that in order to prepare for Weblog, it changes and/or deletes a number of internal predicates, before saving your application in a web-ready form.

However, panic not! If you try to call \texttt{weblog/1} while you have some unsaved program windows open, it will not quit, and will not change anything, but
instead will politely prompt you:

```
| ?- weblog( colour ).
Please save program windows to avoid losing work!
yes
```

At this point, either save or close any unsaved program windows (which, if you followed the above instructions, should not actually be there, but we'll let that pass!), and then try again.

**Time for Deployment**

This current example is very simple to deploy. "All" that you need to do, is set up a folder on your web server, with execute-only access, and copy all the four files mentioned above, plus the two template HTML files ("colour_0.htm" and "colour_1.htm"), into that folder.

When we put "All" in scare-quotes, this is because setting up web servers can be a messy job, and certainly the details for any one version of Microsoft IIS or Apache, are beyond this document: please consult your relevant manuals and online resources.

That said, once set up, and with (in this case) just those six files in place, our web application is ready to run! Let's load it into a browser, and see what happens: the sequence of screens, and responses (using the same inputs as before) are shown on the right.

**Deploying Linked Assets**

The example we've used so far in this chapter, has no linked "assets", and everything that arrives client-side from HTTP interactions with the server, are explicitly generated by Weblog. For this reason, all the files can live together, in an execute-only web folder.

However, most applications will rely heavily on external assets, which will appear in the response pages from the Weblog application, as links, say to .PNG graphics files or .CSS stylesheets.

Although you can and should think about these as being part of your Weblog application, it is important to realise which items are being used for directly generating the output from Weblog (for instance, the two template HTML files in the above example), and which are being requested subsequently by the client's browser, when it encounters links.

In short, anything that will be requested by the browser, perhaps as result of being served an "<img ...>" link, needs to be in a web folder with read-only (for safety) or read/write (if you like to live on the edge) rights.

The supplied Weblog examples all work like this:
The four executable application files (.EXE, .OVL, .SYS, .OVX) live in an execute-only folder, for example, "scripts", together with any HTML template files that they use with submit/2, and the ?.INI file if the application uses one.

All *.PNG, *.JPG, *.MP3, and other graphical or multimedia assets, live in read-only folders, such as "wwwroot\images", "wwwroot\music", or suchlike.

Any linked *.HTM files (complete ones, not containing "{:labels:}", and not used by submit/2, together with *.CSS stylesheets, live in the main read-only web folder, "wwwroot" itself.

Temporary "state" files, created by submit/2 (see below), are created by default in "c:\temp", but you can change this by editing the .INI file which is an optional part of your application.

The Truth Behind Submit

Earlier, we said that the submit/2 predicate can be thought of as little more than a simple "merge" routine: this is, indeed, how it behaves, logically. However, it is considerably more complex than that. Why? Well it's all about how the HTTP web interface works.

Typically, you fire off a Weblog application from an HTML page, which contains a link to "http://my_server.com/scripts/my_app.exe" or whatever, possibly with arguments too. If "my_app.exe" is a copy of "WEBLOG.EXE", then it in turn communicates with "my_app.sys" (a copy of your WIN-PROLOG executable), which may already be running in the background, or if not, is fired up. This initially loads the file, "my_app.ovl" (a copy of the tiny "WEBLOG.OVL"), which in turn looks for a "state" file, initially, "my_app.ovx", your saved application image.

On the first instance, your program begins execution at weblog/0, performing whatever Prolog code you have written, perhaps generating some initial values, or simply getting ready to ask the user for some information. At this point, your application calls submit/2, and a web page is generated.

Interrupted Execution

What Weblog does not do, is then sit, idling away on the server and waiting for you to reply. It actually completes its processing, and quits, this being the signal to the IIS or Apache web server that it's time to send the results to the remote browser. But how does Weblog continue once the user reviews the page, fills in various form entries (or simply presses a form button)?

The answer lies in "state" files. The very last thing submit/2 does, after merging your template HTML page, is to save all the Prolog stacks, heaps, atom dictionaries, predicate definitions and other data areas to a special file, using a predicate called xsave/1. The resulting file contains a total compressed image of the state of execution that Prolog was in, at the point the file was saved. When this same file, if loaded by any instance of Prolog, using the sister predicate, xload/1, the whole of Prolog's memory is wiped, and restored from
the compressed image in the state file: all Prolog stacks, heaps, atom dictionaries, predicate definitions and other data areas become exactly as they were when the previous instance of Prolog called \texttt{xsave/1}, so execution can continue from exactly where it previously left off. All assertions, flags, variable bindings, backtrack points, etc., are preserved in the file, and become active again immediately on loading the previously saved state file.

So, in pseudo-code, \texttt{submit/2} is runs something like this:

\begin{verbatim}
submit( File, Info ) :-
    create_unique_file_name( Rand ),
    merge_for_output( File, [(weblog,Rand)|Info], Html ),
    delete_older_state_files,
    save_state_file,
    show_page( Html ),
    quit.
\end{verbatim}

The "(weblog,Rand)" pair is added to the list of arguments you are storing in the page generated by \texttt{submit/2}, by tagging on a hidden field to the form. When the user clicks on a button on this page, to proceed with execution, Weblog runs again, picks up the name of the saved state file from this hidden field, loads the state file, which carries on from exactly where you left off.

To the user, it feels like one, solid, bespoke connection; to the system, it feels like a series of quick, one-shot, HTTP requests.

\textbf{Scalability and Security}

The beauty of this system, is that it is highly scaleable. Provided the state files are stored in a folder visible to multiple servers, any instance of Weblog on any of the servers, can load and continue your application, with totally no risk of cross-contamination of data. Using \texttt{xload/1} to load a file generated by \texttt{xsave/1}, completely restores the previous set of heaps and stacks, and all other data, even if they were saved by a different instance of Prolog.

Furthermore, without knowing the state file name (randomly created names in a folder that is not available to web browsing or other snooping), there is no way to pick-up on and take over an existing application session.

All in all, Weblog is as secure as the web server upon which it runs.