WDialog

Web Path: WDialog

1 WDialog - Toolkit for Dialog-Centric Web Applications

WDialog is an advanced system to create dialog-centric web applications. It focuses on user interfaces with many input elements that need strict control of the possible user interactions (e.g. editors for structured data, form centers, workflows). The key features can be summarized as follows:

- The definition of the user interface (UI) is strictly separated from the backend program (model) carrying out the actions triggered by the user. The UI is described in an XML file that can be developed independently of the backend program. This improves the software engineering process, as developers for the UI part and the backend part can concentrate on their own skills, and need not to be experts for the other part. The developers are encouraged to adopt a modular programming style even for the UI part. Last but not least, the UI part can be re-designed and localized without having to understand the rest of the application.
- The XML document type for the UI part provides a rich language that specifies the dialog structure of the user interface, and the persistence properties of the dialogs. It further includes integrated HTML form elements, and a macro/template calculus. The nature of this *UI language* allows it to develop a prototype of the user interface that is mature enough for early presentations although the backend is not yet available (rapid prototyping). Session persistency is defined on the level of the UI language, so dialog variables do not forget their state even in the early development stages. The UI language determines the possible ways the user can take through the application (storyboard). Although it has been tries to make the UI language a powerful input/output processor, some features are intentionally not available. It is not possible to develop real algorithms in the UI language, and external stores (databases) are not directly accessible.
- The interactive parts of the dialogs are modeled like classic GUI widget sets, i.e. the HTML interactors can be used like widgets visualizing the current state of the dialogs, and user clicks are considered as events that can be handled by configurable callback methods.
- The callbacks are programmed in a real programming language (Objective Caml, or Perl). The callbacks usually implement the algorithmic part of the application, and have access to all system resources.
- There are several runtime environments for the final application: It can be run as CGI program, but also as dedicated application server to improve the performance.
- The WDialog toolkit itself does not require any database system as background store, and provides several high performance alternatives.
- As security issues become more and more important in these days, it should be mentioned that WDialog includes features that gives the security of web applications a solid and trustworthy basis. Error-prone tasks such as handling escape encodings, parameter passing, and implementing persistency are taken away from the application developer, and are realized in a sound way by the WDialog toolkit. Furthermore, the framework nature of WDialog enforces a certain structure of the application, and keeps developers away from adventurous designs.

These are only the highlights of the WDialog feature list, making it a somewhat different web toolkit. This manual gives an introduction to WDialog explaining its innovative concepts for newcomers. The reference part describes every little piece of the toolkit in detail.

Introduction

Web Path: WDialog / Introduction

2 Introduction

The following pages have been written to explain what WDialog really is. The problem is that I cannot refer to well-established standards, as the WDialog approach is new, and non-trivial to understand. There is also no striking example one only has to study to get the point. Many of the features can only be explained by experience, as there are often shorter ways to achieve similar effects, but I have learned that these do not scale well, or have other disadvantages.

The chapter about the *architecture* (→ 3) is crucial. First it explains a number of constraints every web platform must deal with, independent of the implemented technology. Second, it explains the target model WDialog wishes to realize. This target model is taken from the world of GUI programming where user interfaces are built by combining widgets, and by programming callbacks (which in turn bases on the model-view-controller approach of Smalltalk). The problem is that the constraints of the web protocols are basically incompatible with the GUI model. Nevertheless, there is a way to emulate many aspects of the GUI model, and this path is taken by WDialog.

The chapter about *dialogs and pages* (→ 9) introduces into fundamental concepts of WDialog by explaining code snippets. It gives a first impression what you can really express in the user interface (UI) language. The following chapter gives a complete *example* (→ 12) of a very simple web application.

I think examples are the way to study WDialog, and because of this a realistic application has been written: WTimer. This is a groupware application allowing the users to edit time sheets, and to generate reports from them. It is complete, and ready to be used in production environments. See the chapter entitled "*More examples* (→ 20)" for links to this example.

Architecture

Web Path: WDialog / Introduction / Architecture

3 Architecture

WDialog provides a programming model taken from object-oriented GUI programming, and adopts this model to HTTP-based Web applications. In the following paragraphs, I try to explain the WDialog model, the similiarities to the GUI model, and the differences to other Web application concepts.

3.1 From the GUI model to WDialog

Picture 1 outlines some architectural aspects of traditional graphical user interfaces. Usually, there is a window object (or several objects) for every displayed window. The window object is the base object for programming, and implementing a certain window behaviour means to override methods of the object. The window system manages it to connect the window object to a real input/output device (i.e. an area of the screen, mouse, keyboard). User input (keystrokes, moving the mouse etc.) is translated into a stream of events sent to the window object. Conversely, the window object can send display requests to the window system.

The window object and the real window exist as long as the session remains open. It is important to mention that there is a session, as this is one of the problems with Web-based applications.

The window object stores internally some state. The state reflects both the current step in the dialogue with the user as well as the data the window displays.

Picture 2 shows the architectural constraints of HTTP-based applications. The browser and the HTTP server communicate via HTTP, and the server finds the right Web application (which is usually a separate program) and invokes it. There is no normative protocol for the invocation of the Web application, but the most popular is certainly CGI.

The requests (HTTP or application) play a similar role as the events in the GUI model; they normally indicate user interaction. In the same manner, the responses can be compared with the display requests. However, there are some fundamental differences between both models which makes it difficult to simulate the GUI model on top of an HTTP architecture.

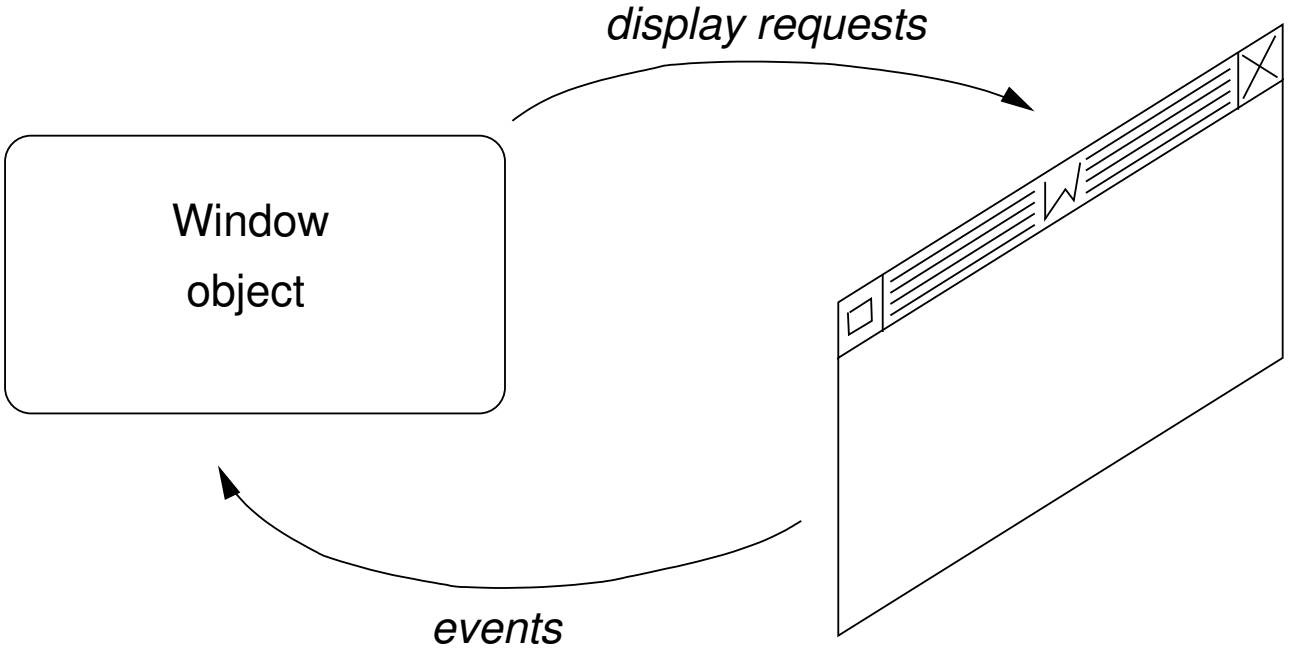
In contrast to the GUI architecture discussed above, there are no sessions. The HTTP protocol isolates consecutive request/response cycles from each other. The protocol does not give any hint whether several such cycles belong to the same logical series of interactions, or whether they are unrelated.

Furthermore, there must always be a response for every request in HTTP. The response normally replaces the previous response, it is not possible to transfer only the parts of the page that have changed (for example, to add some rows to an HTML table). The response is a function of the request, and only of this single request. This is different from the GUI model that tries to only exchange delta information between the window object and the window system.

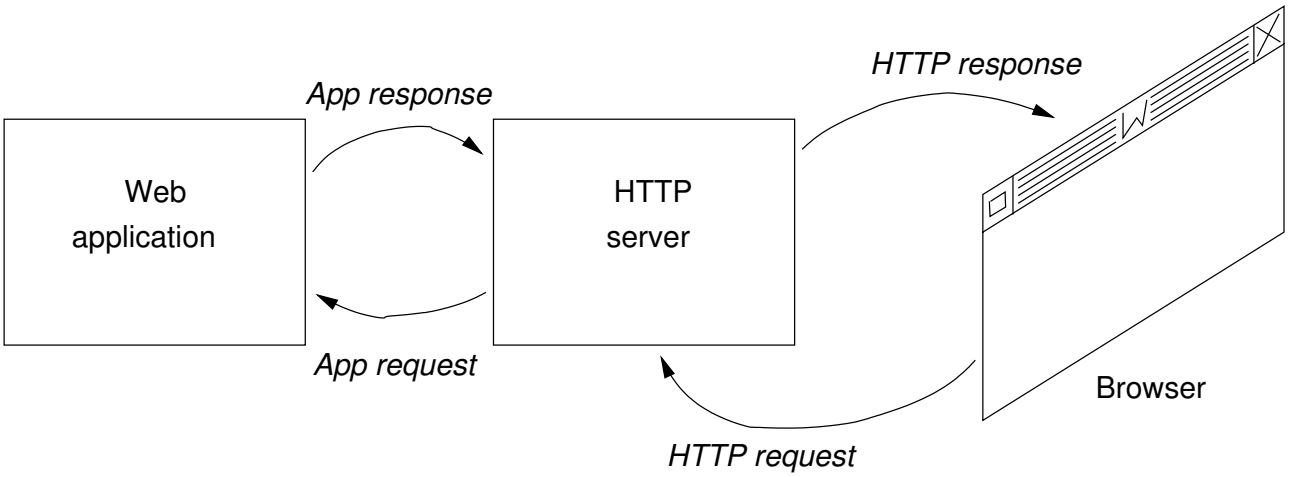
The strict request/response scheme means that the Web application works like a function that processes the request and produces the response. The application protocol connecting the Web application with the HTTP server reflects the functional behaviour. If CGI is used as application protocol, every invocation of the Web application is perfomed in a separate process such that the operating system already guarantees that invocations do not interfer.

The idea of WDialog is to simulate aspects of the GUI model in the HTTP architecture. Picture 3 illustrates this: The WDialog library (linked as part of the application) creates the illusion of a window object communicating with an application window. The window object receives events if the user presses buttons or clicks at hyperlinks, and the window object sends display requests to the application window. Of course, the illusion is not perfect, and there are many restrictions

Picture 1: The GUI programming model



Picture 2: HTTP-based applications



compared with the GUI model.

In reality, the application window is an ordinary browser window displaying HTML pages. The HTML code is mixed from two sources: On the one hand, the application programmer designs the visual layout of the page, and writes an algorithm that creates the necessary HTML elements implementing the desired look. On the other hand, some of the HTML code is generated by WDialog because it is needed to simulate the GUI model. In order to mix the elements from the two sources accordingly and in a predictable manner, WDialog evaluates a special code block called *User Interface (UI) Definition*. This is an XML file containing normal HTML elements for the visual layout, and special "ui" elements that are going to be replaced by simulation code during page processing. For example, the page

```
<html>
  <body>
    <ui:form>
      This page has a <ui:button name="b" label="button"/>.
    </ui:form>
  </body>
</html>
```

contains the special elements `ui:form` and `ui:button`. `ui:form` establishes the simulation environment needed for `ui:button`, as it is transformed into a HTML `FORM` element with additional parameters referring to the current state of the simulation. `ui:button` is transformed into a HTML `INPUT` element that is able to send a "button press" event to the application in a way that can be uniquely decoded by WDialog. The overall effect is that the application has the illusion that the browser understands all the `ui` elements, or in other words, that the browser provides an application window that can be programmed in the GUI style.

The window object is modeled using the object-oriented features of the underlying programming language; however there is one important difference to the corresponding objects in the GUI model. The window object exists as object of the programming language only from time to time, especially in the moment the response is computed from the request. As already pointed out, the response must be a function of the request, and because of this, it is not possible that the object persists between two consecutive request/response cycles, for example in a background store. However, the object *must* be persistent, and the solution is that the object is serialized (as a stream of bytes) and transferred within the request and response messages. Every time the application receives a request, the WDialog input transformer deserializes the object, and awakes it again as object of the programming language. Conversely, the output transformer serializes the object and includes it into the outgoing response message. In HTML the so-called hidden form fields can be used to transfer such an invisible data field in the request and response messages. Actually, these form fields are generated as part of the mentioned `ui:form` transformation.

The effect of this serialization technique is that the object seems to be persistent. However, the illusion is far away from being perfect; the instance variables of the object need to be declared in a special way such that the WDialog library knows which variables make up the object. If the application code uses other instance variables, these are lost between consecutive cycles.

Of course, the simulation cannot break with the strict nature of the request/response cycles, i.e. that every HTTP request must be followed by a complete response. However, the WDialog library implements several techniques making life easier. For example, the application needs not to analyze requests; the WDialog library already does it, and the results are automatically stored in the window object. Requests are handled as if they were events, and the WDialog library extracts button events, hyperlink events and so on from the arriving request messages. Mutable interactors like text input boxes must be tied to instance variables of the window object. This means that the contents of the interactors are initialized from the current value of the corresponding instance variable, and that user changes of the contents are automatically propagated back to the variable.

The composition of response messages is simplified by the user interface (UI) definition language. As already pointed out, the application can (and must) use the abstract `ui` elements hiding implementation details of the simulation. These `ui` elements are part of the UI language among others (all have the `ui` prefix). For example, the HTML code is grouped

into so-called pages. A page can be seen as an output method of the window object, and it generates the whole HTML document that is displayed in the browser window. It is possible to have several `ui:page` sections, and this can be used to provide different visualizations of the same state. The following XML file shows a complete UI document; the application reads such a file in order to get the declarations of the objects and to get the definitions of the pages:

```
<?xml version="1.0"?>
<!DOCTYPE ui:application PUBLIC "-//NPC//DTD WDIALOG 2.1//EN" "">

<ui:application>
  <ui:dialog name="sampledialog">

    <ui:variable name="v1"/>
    <ui:variable name="v2"/>

    <ui:page name="order12">
      <html>
        <body>
          <h1>Variables in order 1, 2:</h1>
          v1: <ui:dynamic variable="v1"/>
          v2: <ui:dynamic variable="v2"/>
        </body>
      </html>
    </ui:page>

    <ui:page name="order21">
      <html>
        <body>
          <h1>Variables in order 2, 1:</h1>
          v2: <ui:dynamic variable="v2"/>
          v1: <ui:dynamic variable="v1"/>
        </body>
      </html>
    </ui:page>
  </ui:dialog>
</ui:application>
```

This file defines one dialog (window) object `sampledialog` containing two instance variables `v1` and `v2` (these are string variables by default). (In the rest of this manual, the term *dialog object* is preferred over *window object*. This stresses the role of dialogs as the link spanning the individual page invocations. It is the series of interactions that counts, not the single web page.) There are two pages: `order12` visualizes the dialog object by first displaying `v1` followed by `v2`, and `order21` shows the variables in the reverse order. When the application wants to send the response message to the browser, it simply selects one of the defined pages of the object, and delegates the details of sending the message to the WDialog library. (This is done by the already mentioned output transformer.)

3.2 The components of a WDialog application and how they interact

The application consists of the following major components:

- The application program written in a programming language
- The WDialog library

- The UI definition

At startup, the application invokes the initialization function of the WDialog library which loads the UI document. By default it is assumed that the application is connected via CGI to the HTTP server, and the library is set up for this environment. However, it is possible to integrate the application into other environments [1].

The next step of the application is to connect classes of the program with the dialogs occurring in the UI document. Besides pages, the UI document declares only the instance variables of the dialog objects, but not methods for other purposes. The WDialog library has a registry associating classes of the program with dialogs of the UI definition. After the registration has been performed, the WDialog library is able to deserialize arriving dialog objects and to represent them as ordinary objects of the program. Furthermore, the library can now invoke methods on the objects; there are two methods with predefined meaning: `prepare_page` and `handle` (see below). Last but not least it becomes possible to serialize the dialog object again.

The application calls now the `process_request` function of WDialog doing all the rest. The result of this function is the response message that is sent to the HTTP server. See the illustration in picture 4.

`process_request` first analyzes the request. The current object is deserialized, and in the following, the other information included in the request are stored into the object: The event is extracted from the request, and all changes to user-modifiable interactors are processed (normally by setting the corresponding instance variables of the object).

The next step is that the method `handle` of the current object is invoked. This method is fully customizable as it is implemented as ordinary object method; its task is to react on the last event. For example, if the application allows users to enter data records, one possible event (button) is "Store this record". An implementor of `handle` would get the entered record (from the instance variables), and put these data into a database.

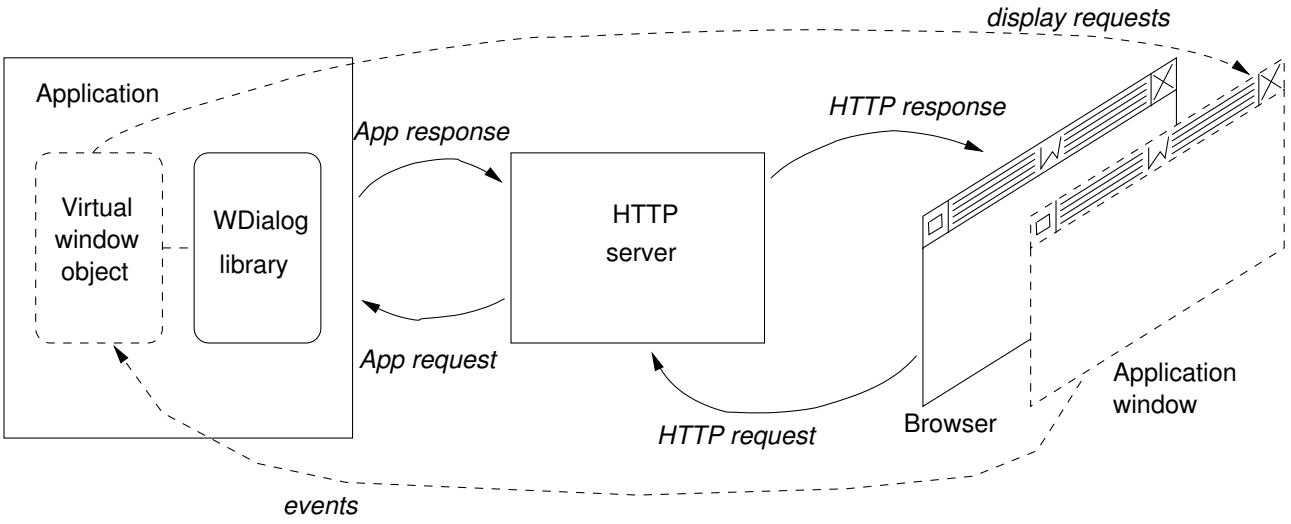
The `handle` method must also determine what has to happen next; we still need a response. One possibility is to select one of the pages (output methods) of the current object. Alternatively, a different object can be created, initialized, and a page from this object can be selected (not displayed in the picture).

Once the page is known, the library could immediately output the contents of the page. However, before output starts, the method `prepare_page` is called; this method is an ordinary method of the object, too. The programmer has the opportunity to set instance variables of the object that are needed for the visualization. For instance, if the application allows users to query data records, the `prepare_page` method will perform the query on the database and write the resulting records into instance variables of the object.

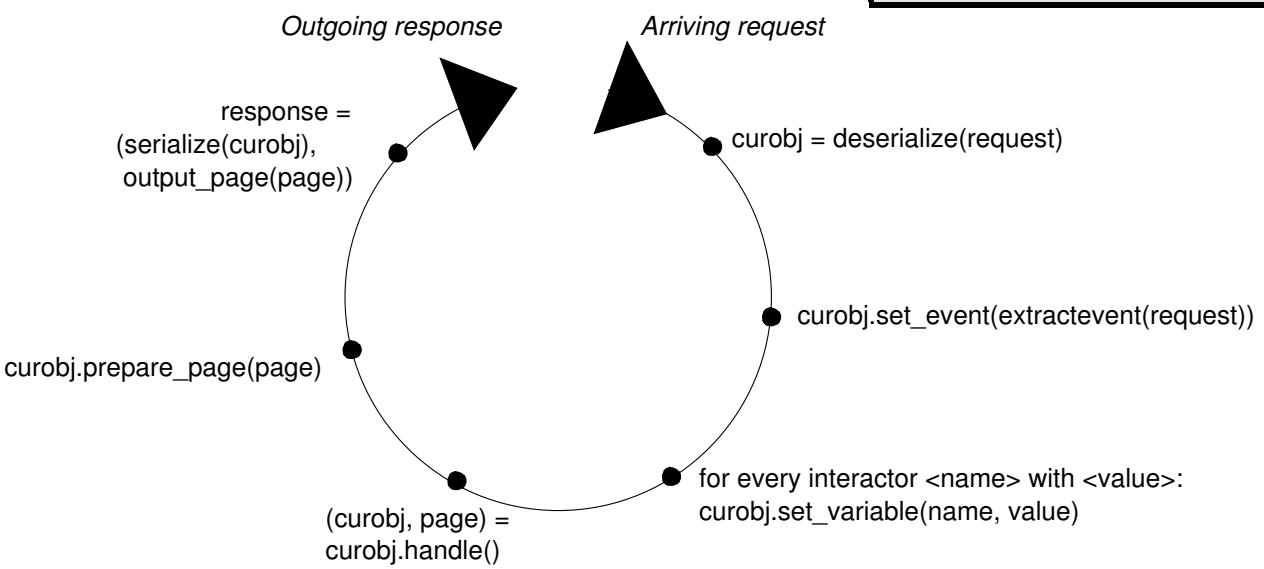
Finally, the selected page is actually transformed to HTML and sent to the HTTP server. This operation includes the serialization of the current object.

[1] This can be done using the netcgi abstraction provided by the ocamlnet library on which WDialog bases.

Picture 3: The WDialog architecture



Picture 4: Steps of the process_request function



Dialogs and pages

Web Path: WDialog / Introduction / Dialogs and pages

4 Dialogs and pages

The user interface definition contained in the index.ui file is divided up into *dialogs* which are themselves divided up into *pages*.

A typical index.ui file looks as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE ui:application PUBLIC "-//NPC//DTD WDIALOG 2.1//EN" "">

<ui:application start-dialog="dlg_A">

  <ui:dialog name="dlg_A" start-page="page_1">
    <ui:page name="page_1">...</ui:page>
    <ui:page name="page_2">...</ui:page>
    ...
  </ui:dialog>

  <ui:dialog name="dlg_B" start-page="page_1">
    <ui:page name="page_1">...</ui:page>
    <ui:page name="page_2">...</ui:page>
    ...
  </ui:dialog>

</ui:application>
```

The file format is XML-1.0 with the addition that the prefix `ui:` is pre-bound to all tags defined by the WDialog system (actually, this prefix is used like a namespace prefix; however a namespace declaration for ui: would not be understood by the system); in contrast to this, tags without prefix are used for the generated HTML code. For example:

```
  <ui:page name="page_2">
    <html>
      <body>
        <h1>This is a HTML page!</h1>
        ...
      </body>
    </html>
  </ui:page>
```

Here, the tag defining page_2 is part of the WDialog language, and its name `ui:page` has the prefix, while the contents of `page_2` are normal HTML tags lacking this prefix.

A *dialog* is a container for a piece of the state of the application. Dialogs begin to exist when the user of the application visits the first page, their state is changed when the user fills out forms and presses buttons, and they vanish when the user leaves the application. The lifecycle of a dialog is the series of interactions of one user in one session. Note that dialogs cannot be used to save the state of the application from one session to the next session; for this purpose an appropriate background store (i.e. database) is required, which is beyond the scope of WDialog.

A dialog has instance variables which can be individually referenced and modified. For example, an application to manage phonebooks has dialogs describing entries of the book:

```
<ui:dialog name="entry" start-page="page_1">
  <ui:variable name="person"/>
  <ui:variable name="number"/>

  <ui:page name="page_1">...</ui:page>
  <ui:page name="page_2">...</ui:page>
  ...
</ui:dialog>
```

Note that variables contain strings by default; however, there are other data types which are important in the context of dynamically generated HTML.

It is very natural that an application consists of several types of dialogs, as there are normally different views on the topic the application manages (e.g. the application may display a list of phonebook entries, or a single entry).

The *pages* of an object define different visualizations of one dialog. This feature can be used to provide different views on the same dialog state, or to split up the state such that every page shows only a subset of all variables.

There is an analogy between dialogs, pages, and variables and the well-known terminology of object-oriented programming:

- Dialog declaration ⇔ class
- Dialog instance ⇔ object
- Dialog variable ⇔ instance variable
- Page ⇔ method

A page is like a method for the object producing HTML output. For example, we can define a page showing the contents of an entry of a phonebook:

```
<ui:dialog name="entry" start-page="show">
  <ui:variable name="person"/>
  <ui:variable name="number"/>

  <ui:page name="show">
    <html>
      <body>
        <h1>Entry</h1>
```

```
        <b>Person:</b> <ui:dynamic variable="person"/><br/>
        <b>Number:</b> <ui:dynamic variable="number"/>
      </body>
    </html>
  </ui:page>

</ui:dialog>
```

Here, the page "show" outputs the name of the person and the associated number. The code contains the special elements `ui:dynamic` which are dynamically replaced by the current contents of the variables they are referring to[2].

We can define another page, `input`, allowing the user to edit the current contents of the two variables:

```
  <ui:page name="input">
    <html>
      <body>
        <h1>Entry</h1>
        <b>Person:</b> <ui:text variable="person"/><br/>
        <b>Number:</b> <ui:text variable="number"/><br/>
        <ui:button label="Change" name="change_entry"/>
      </body>
    </html>
  </ui:page>
```

The special elements `ui:text` generate the HTML code for text input boxes which are initialized with the current values of the two variables. Furthermore, the special element `ui:button` is transformed to HTML code for a submit button. When the user hits this button, the default behaviour is to update the dialog variables which have been bound to input widgets. In this example, first the contents of the variables `person` and `number` are put into the text boxes when the page is displayed by the browser, but after the use has edited them and has pressed the button, the (possibly modified) values of the text boxes are stored back into the variables. This is called *automatic update of interactor variables*.

This shows an important difference to other web platforms for dynamic content: Input and output are seen as a unit during the whole transaction. It is the job of the toolkit to propagate user input to the corresponding programmable containers (i.e. to the variables). Furthermore, the interactors can be output as HTML in a way such that the link to the container is not lost, which makes the automatic update of variables possible.

[2] The default replacement algorithm treats characters such as < and & specially which would be recognized as meta characters by HTML parsers; i.e. they are automatically converted to the sequences < and &, respectively.

An Example

Web Path: WDialog / Introduction / An Example

5 An Example

5.1 The task: Add two numbers

In this chapter I present a very simple web application, and explain it. The task is to let the user enter two numbers, and compute the result once the button is pressed.

You can find the complete solution for this example in the source distribution of WDialog. It consists of three files: `index.cgi`, `index.ml`, and `index.ui`. This example uses Objective Caml as scripting language (compiled ad-hoc); of course, this is not the optimal way to run a web application, but it is the best to learn WDialog as you can change the script, and it has an immediate effect without needing to recompile the program.

The file `index.cgi` is the part that launches the Objective Caml interpreter, and is of no special interest. The file `index.ml` is the application written in Objective Caml, and `index.ui` is the UI definition file in XML syntax.

5.2 The structure of the solution

This example consists of only one dialog object. This dialog object contains the three numbers (first_number, second_number, result), and because this triple is the only data unit we have to deal with there is no other dialog object. This is a rule of thumb: For every part of the user interaction model that is centred around one record of data you need a separate dialog object.

As already mentioned, the dialog object contains three numbers. This can be written as (`index.ui`):

```
<?xml version="1.0"?>

<!DOCTYPE ui:application
          PUBLIC "-//NPC//DTD WDIALOG 2.1//EN" "">

<ui:application start-dialog="add-numbers">

  <ui:dialog name="add-numbers" start-page="enter-numbers">

    <ui:variable name="first_number"/>
    <ui:variable name="second_number"/>
    <ui:variable name="result"/>

    <ui:page name="enter-numbers"> ... </ui:page>
    <ui:page name="display-result"> ... </ui:page>
    <ui:page name="display-error"> ... </ui:page>

  </ui:dialog>
</ui:application>
```

Furthermore, we have three pages, i.e. three ways to present these variables to the user:

- `enter-numbers`: The user can enter the numbers to add, no result is displayed
- `display-result`: The two entered numbers are displayed together with the sum.
- `display-error`: If the user does not enter a syntactically correct number, an error will be displayed.

The start page is `enter-numbers`, i.e. this page will be displayed when the application is visited for the first time.

The ways the three variables are referenced are quite different. In the first page, input boxes prompt the user to enter something. WDialog allows it to bind input boxes to variables. The effect is that the boxes are prefilled with the current value of the variable when the page is displayed (sent to the web browser), and that any modifications of the contents of the boxes are passed back to the variables.

The page `display-result` simply includes the current values of the variables in the displayed HTML text.

The last page `display-error` is only about the contents of the variables without displaying them.

There is another aspect about the pages. The user can change from one page to another page, but of course only on predefined paths. The page transitions can be described in the UI definition file, and they can be programmed by hand. The page `enter-numbers` contains a button "Compute Result" normally directing the user to the second page, `display-result`. However, if the syntax of the numbers is wrong, the next page will exceptionally be `display-error`. As the first page transition is the expected one, it is recommended to describe it in the UI definition file ("goto" attribute, see below), and program the second possible transition manually.

There are further transitions from `display-result` and `display-error` back to the start page.

In the user interface, transitions are usually represented as buttons or links. WDialog allows it to give these elements names, and when they are pressed or clicked, events are triggered that are identified by these names.

So there are two levels: First, buttons trigger events, and second, events are processed and cause page transitions.

5.3 The details of the pages

Here is the first page, `enter-numbers`:

```
<ui:page name="enter-numbers">
  <html>
    <head>
      <title>The Ultimative Adder</title>
    </head>
    <body>
      <ui:form>
        <h1>The Ultimative Adder</h1>

        <p>Please enter the two numbers you want to add:</p>
        <p>
          <ui:text variable="first_number"/> +
          <ui:text variable="second_number"/> =
```

```
          <ui:button name="add" label="Compute Result" goto="display-result"/>
        </p>
      </ui:form>
    </body>
  </html>
</ui:page>
```

The special WDialog text boxes that are linked with the variables are created by `<ui:text variable="name"/>`. They are transformed to ordinary HTML text boxes (INPUT TYPE="TEXT") that are filled with the current values of the variables, and if the user changes the contents of the boxes they will transferred back to the variables.

The special WDialog buttons are created by `<ui:button ...>`. The button `add` triggers the event `Button("add")` when pressed, and the default action is to go to the page `display-result` (by using the `goto` attribute). Note that the action can be overridden by the web application.

The page `display-result` looks as follows:

```
<ui:page name="display-result">
  <html>
    <head>
      <title>The Ultimative Adder</title>
    </head>
    <body>
      <ui:form>
        <h1>The Ultimative Adder</h1>

        <p>The result is:</p>

        <p>
          <ui:dynamic variable="first_number"/> +
          <ui:dynamic variable="second_number"/> =
          <ui:dynamic variable="result"/>
        </p>

        <p>
          <ui:button name="back" label="Go back" goto="enter-numbers"/>
        </p>
      </ui:form>
    </body>
  </html>
</ui:page>
```

On this page, the contents of the variables are dynamically inserted into the generated text. The special element `<ui:dynamic .../>` causes the contents of the variable to be encoded correctly (by substituting meta characters like `<` by their corresponding entities, here `<`), and printed instead of the element.

Finally, the error page is:

```
<ui:page name="display-error">
  <html>
    <head>
      <title>The Ultimative Adder</title>
    </head>
    <body>
      <ui:form>
        <h1>The Ultimative Adder</h1>

        <p>Sorry, one of your numbers is not a number. Please go
        back and correct your input.</p>

        <p>
          <ui:button name="back" label="Go back" goto="enter-numbers"/>
        </p>
      </ui:form>
    </body>
  </html>
</ui:page>
```

An important detail of all three pages is that the special WDialog interactors are placed within the special `ui:form` element. *This is required.* `ui:form` is transformed to an HTML FORM element containing a number of hidden fields that help WDialog managing the interactions.

Note that you can still use the HTML form element directly, but WDialog does not provide any support for this.

5.4 The application

The other part of the web application is the CGI program that actually adds the two numbers and sets the result variable. Because it is relatively short, we include here the whole `index.ml` file:

```
[01] open Wd_dialog
[02] open Wd_run_cgi
[03] open Wd_types
[04]
[05]
[06] class add_numbers universe name env =
[07] object (self)
[08]   inherit dialog universe name env
[09]
[10]  method prepare_page() =
[11]    (* This method is empty in this example *)
[12]    ()
[13]
[14]  method handle() =
[15]    (* Check which event has happened: *)
[16]    match self # event with
[17]      Button("add") ->
```

```
[18]        (* Get the numbers and convert them from string to int. Catch
[19]         * errors.
[20]         *)
[21]        let n_1, n_2 =
[22]          ( try
[23]              let s_1 = self # string_variable "first_number" in
[24]              let s_2 = self # string_variable "second_number" in
[25]              (int_of_string s_1, int_of_string s_2)
[26]            with
[27]              error ->
[28]                (* On error: Jump to the error page *)
[29]                raise(Change_page "display-error")
[30]          )
[31]        in
[32]        (* Add the numbers, and store the result into the variable *)
[33]        let r = n_1 + n_2 in
[34]        self # set_variable "result" (String_value (string_of_int r));
[35]
[36]    | _ ->
[37]        (* Do nothing if the event is not recognized *)
[38]        ()
[39] end
[40] ;;
[41]
[42]
[43] run
[44]   ~charset:`Enc_utf8
[45]   ~reg:(fun universe ->
[46]          (* Bind the XML dialog "add-numbers" to the class "add_numbers": *)
[47]          universe # register "add-numbers" (new add_numbers)
[48]        )
[49]   ()
[50] ;;
```

I have added line numbers to help readers who are not familiar with the language Objective Caml.

Lines 1-3 open the relevant modules of the WDialog library.

Lines 6-40 define the class `add_numbers` that inherits from the WDialog class `dialog`. By inheriting from this base class the class gets all the properties that are required to act as dialog. We will go into detail later.

The lines 43-50 are the main program. It calls the WDialog function `run` doing all the necessary steps. There are two configuration options: `~charset` sets the character set that is internally used by WDialog and that will also be used to represent the generated HTML pages. The option `~reg` configures the registry containing the bindings of dialogs to classes. Here we simply bind the `ui:dialog` called `add-numbers` to the Objective Caml class called `add_numbers`.

The instance of the option `~reg` must be a function that gets the so-called universe as input. The universe consists of all defined dialogs and the corresponding classes, and acts as a factory for new dialog objects. Let `u` be the universe. To create a new object for the dialog "add-numbers", you can call `u # create env "add-numbers"` (the symbol # is the method invocation operator). Here, `env` is the environment, a record of data that are different for every invocation of the CGI; there is normally an environment at hand when you want to create an object. - The function passed to `~reg` initializes the universe by telling WDialog which dialogs correspond to which classes (this needs not to be a one-to-one correspondence).

Now back to the definition of the class. The lines 6, 7, 8, and 39 form the outer "braces" defining the class:

```
class add_numbers universe name env =
object (self)
  inherit dialog universe name env
  ...
end
```

The symbols `universe`, `name`, and `env` are only passed to the super class `dialog`, we can ignore them now. The symbol `self` is the symbolic name of the current object (also called `this` in other languages; in Caml you are free to choose your preferred name).

As you can see, the class has two methods. `prepare_page` is called just before the dialog object outputs the next page (the preprocessing phase). It is normally used to set dialog variables that are only needed for visualization, but do not have any effect on the actions. For instance, if you wanted to display the current date at the top of the page, you could set another dialog variable `date` to the current date in this method. For this example, however, `prepare_page` is not needed, so we leave it empty.

The other method, `handle`, is called after the user has reacted on the page (for instance, pressed a button), and it is always called for the last object seen by the user. This method contains the post processing actions of a page.

Before `handle` is called, the WDialog processor has reconstructed the last dialog object. This is worth to be mentioned, because `handle` is called in the CGI activation following the activation that invoked `prepare_page` and that actually sent the page to the client. So it is usually called in a different Unix process. However, WDialog has managed it that the dialog object of the last activation is reanimated so it is accessible again (by object serialization).

The necessity of object serialization is another reason why we are using the special dialog variables that can be declared with the XML expression `<ui:variable>`. The values of these variables survive from one CGI activation to the next because they are part of the serialized objects; if we just declared ordinary Caml instance variables their values would be lost.

The body of `handle` first checks the last event. This is done using the pattern matching operator

```
match ... with pat1 -> expr1 | pat2 -> expr2 | ...
```

The expression `self#event` returns a symbolic term describing the last event. Here, we are only interested in the case when the last event is `Button("add")`, i.e. the user has pressed the button labeled "Compute Result". In all other cases (the pattern _) we do nothing. These other cases include the events that the user pressed the "Go back" buttons. It is not necessary to do something when these buttons are hit because the `goto` attribute already sets an action, namely to go to the requested page.

From lines 21 to 31 we extract the current values from the dialog variable `first_number` and `second_number`. This is done in two steps:

```
let s_1 = self # string_variable "first_number" in
```

This line looks up the named dialog variable and binds their string contents to the symbol `s_1`. The next line performs the same for the other dialog variable, and defines `s_2`. The method `string_variable` is one of the methods inherited from the super class `dialog`. The strings are converted to integers (by using the function `int_of_string`). Of course, this may fail because the user did not type in a number. In this case, the Caml runtime system throws an exception which is caught by the `try ... with...` block. Normally, when the user entered a correct number, the comma in line 25 forms a pair of the two integers, and this pair is bound to the pair pattern in line 21. The result is that `n_1` contains the first number as integer, and that `n_2` contains the second number as integer.

Line 33 adds both numbers and calls the sum `r`. Finally, line 34 sets the dialog variable `result` to the string value that corresponds to the integer `r`. Here, the method `set_variable` is again one of the inherited methods.

Note that there is no statement requesting a certain page as the next page to display. We do not need that because the `goto` attribute of the button set the name of the next page, this attribute is still in effect.

Line 29 is only executed when the user did not enter valid numbers. In this case, an exception is thrown by the Caml runtime system, and it is caught in lines 26/27. What happens is that another, new exception is raised causing that the rest of the method definition is bypassed. Furthermore, this exception `Change_page "display-error"` overrides the effect of the `goto` attribute by requesting a different page. The effect is that the next displayed page is `display-error`, and not `display-result`.

5.5 See the example live

Under this URL the example is running.[3]

5.6 Some observations

- After you entered two numbers into the boxes, pressed the "Compute Result" button, and then went back, you see the two numbers again. This is because we do not change the contents of `first_number` and `second_number` by our program, so they keep their values. That means that these variables are not just CGI parameters, they have some "magic" causing that they are automatically passed from one page to the next and back to the previous one.
- This works even if you enter invalid numbers. You will see the error message, but when you go back, the invalid values appear again.
- You may ask from where the variables get their initial values. The initial values are implicitly defined by the `<ui:variable>` declarations, because a string variable defaults to the empty string. You can specify another initial value, for example

  ```
  <ui:variable name="first_number">
    <ui:string-value>42</ui:string-value>
  </ui:variable>
  ```

  declares that `42` is the intial value for `first_number`.

5.7 How to get this example running

- Of course, you must have installed WDialog properly (see the INSTALL file coming with the distribution)
- Now go into the directory `examples/adder`. You will find the three mentioned files here. Run

  ```
  ./index.cgi
  ```

  from the command line. A message appears explaining that this is a CGI program, and you will be prompted for parameters. Just enter `.` and press Enter:

  ```
  ice:/home/gerd/npc/uiobjects/examples/adder > ./index.cgi
  This is a CGI program. You can now input arguments, every argument on a new
  line in the format name=value. The request method is fixed to GET, and cannot
  ```

[3](URL: http://wdialog.sourceforge.net/examples/adder/)

```
be changed in this mode. Consider using the command-line for more options.
> .
(Continuing the program)
Content-type: text/html; charset=UTF-8
Cache-control: no-cache
Pragma: no-cache
Expires: Fri, 08 Feb 2002 01:01:37 +0000

<html >
        <head >
          <title >The Ultimative Adder</title>
        </head>
        <body >
        ...
        </body>
      </html>
      ...
```

The script outputs the start page. If you get it, everything is installed right.

- Now you have to configure your web server (do you have one?) to run CGIs. This depends on your product (sorry).
- Finally, you can enter the right URL into your browser, and the example will appear. Again, the "right URL" depends on your web server configuration.
- For maximum speed, it is better to compile the program. Objective Caml has two compilers, one generating byte code, and the other generating assembly language code. Byte code has only medium speed, but it is platform-independent. Of course, the assembly language code is much faster. The two compilers have to be invoked as follows:

```
ocamlfind ocamlc -o byte.cgi -package wdialog -linkpkg index.ml

ocamlfind ocamlopt -o fast.cgi -package wdialog -linkpkg index.ml
```

The UI definition must be named after the CGI, so you better create the symlinks:

```
ln -s index.ui byte.ui

ln -s index.ui fast.ui
```

You do not need to recompile the program if you only change the UI definition.

More Examples

Web Path: WDialog / Introduction / More Examples

6 Examples included in the source tarball

The distributed sources include some examples:

- `adder`: The adder as described in the previous section
- `adder-jserv`: Another version of the adder that uses the JSERV connector instead of CGI
- `list-ml`: Displays directories of the file system. Introduces into the usage of templates and associative variables. This program is also an example of internationalization.
- `list-alt-ml`: An alternate solution of the directory browser.
- `list-dbm-ml`: Another variant (derived from `list-alt-ml`) that stores sessions into databases.
- `list-dbm-popup-ml`: Another variant (derived from `list-dbm-ml`) that demonstrates popup windows.

The examples can be found in the directory `code/examples`.

7 A complete application: WTimer

As a real-life demonstration I have written a complete application for WDialog: WTimer allows the user to edit time sheets that are stored in databases. The application is complete, and proves that a real-life application can actually be developed.

WTimer is not a simple program, but I think it is well-structured, and you can immediately understand which module implements which feature. The code is sometimes tricky, however, as complicated user interactions must be processed. So I think it is not a good idea to read this example as the first one, but it serves as an illustration how to combine the elementary features of WDialog to get non-trivial effects.

You can find more information about WTimer on the *project page*[4].

[4](URL: http://www.ocaml-programming.de/packages/documentation/wtimer/)

Reference

Web Path: WDialog / Reference

8 The reference manual: How it is organized

There are a lot of chapters, but they fall all into one of the following categories.

- A large part of the manual is dedicated to the UI definition language. The section *The UI language* (→ 77) contains descriptions for every ui element such as *ui:application* (→ 86), *ui:variable* (→ 170), or *ui:button* (→ 87). The small chapters about *Processing instructions* (→ 76), and *The standard UI library* (→ 182) should be mentioned here, too.
- Another part describes the API of the WDialog library. The chapter *WDialog API (O'Caml)* (→ 183) goes through the whole interface of the library, and describes every type, class, and function. The manual currently covers only the O'Caml version of the library; the Perl bindings are only documented as `perldoc` in the individual Perl modules.
- There are some fundamental concepts that are important for both the UI definition and the application program. The chapters about *Dialogs* (→ 22), *Data types* (→ 27), *Events* (→ 41), and *Templates* (→ 49) fall into this category.
- Finally, there are chapters about advanced techniques describing expert knowledge: *Internationalization* (→ 67), *Output encodings* (→ 71), *Runtime models* (→ 227).

Dialogs

Web Path: WDialog / Reference / Dialogs

9 Dialogs

Unsurprisingly, dialogs are the central notion of WDialog. The idea is that consecutive user interactions share the same state, one only being a small modification of the other. Imagine you want to program a table editor, and the user interface allows the user to add rows, add columns, delete rows, delete columns, and fill values into the cells of the table. All these actions are very similar, and the most important part of the shared state is the table the operations modify. So in our first approximation the dialog consists of a series of similar actions targeting to modify the same operand.

If we look closer at this model, we can find out that the "shared state" and the "interactions" have a certain structure. The state consists often of some "main state" and some "auxiliary state". In our example, the representation of the edited table would be "main state", but we can easily imagine that there are some variables around it that are only needed for certain actions. The operation "delete row" needs certainly a designator that determines which rows are object of the deletion. This could be some checkboxes, and it is possible to select the rows to delete, or it could be an input box for the index of the row to delete. Such a designator would be an example for "auxiliary state", typically it is only required for certain operations, and only for the next operation the user triggers.

The interactions seem to consist of an "output part" and an "input part". In general, the output part is the HTML page that is generated. The input part are the input boxes, checkbuttons, hyperlinks etc. the user can interact with, and that produce some value that is transferred to the server. Unfortunately, the output part and the input part are mixed up on the HTML level: If I want to create a text box, I have to send a text like

```
<input type=text name="name_of_the_box" value="initial_value">
```

to the browser, i.e. if I want to create the input facility, I have to output text first (and it must be embedded at the right place). Furthermore, I have to remember the name of the text box, so I can later identify the input value the browser sends back when the user has filled out the text box.

It is easy to imagine that it can become quite difficult to manage the various components of the state, and that it is a complicated task to generate the HTML code for input widgets such that the data stream returned by the browser can be interpreted. WDialog even goes one step further: It integrates the concepts for state and the concepts for input widgets.

An input box needs a state variable that is bound to it. Such a variable can be declared by the XML statement:

```
<ui:variable name="textvariable" type="string"/>
```

The state management routines of WDialog automatically ensure that the state variables keep their values in the whole dialog cycle (i.e. in the current series of user clicks). This special property is called the *persistency* of dialog state. This does not mean that the dialog state is stored in some non-volatile memory, it only means that the state survives the cycles of the underlying HTTP protocol.

Furthermore, the creation of the input box is denoted by the special XML element

```
<ui:text variable="textvariable"/>
```

which is automatically translated by WDialog to the previously mentioned `<input type=text>` HTML element, but which should be better considered as a new widget with unique properties. This text box always visualizes the current value of the variable `textvariable`. This means that if the program modifies the variable, the text box will reflect the change at the next opportunity, and it means that if the user edits the text box, the updated value will be stored in the variable.

In WDialog all input widgets are tied to state variables. There is even a symmetry between the possible input widgets and the data types of variables. Obviously, a text input box needs a string variable. A selection box needs an enumerator variable. When I designed WDialog I extended the possible data types until I found a set that can easily represent the states of the widgets.

A dialog consists often only of one major "screen", i.e. one way of arranging the HTML page. In the "table" example, the main layout is certainly the table itself. However, it is also clear that there must be auxiliary pages (e.g. output a warning before continuing an action) such that it is possible to switch between the main page and the auxiliary pages without losing the state. Because of this, the dialogs may define several pages, i.e. fundamental ways of generating the HTML visualization at the current step of the dialog. A page simply consists of a tree of HTML elements and elements of the UI language like `<ui:text>` that are transformed to HTML elements. When WDialog selects the page, it selects between several of such trees of HTML visualization.

There is another aspect of pages. When the user clicks at a hyperlink or presses a button, an action is triggered, and a new page is displayed. The name of the action is called *event*.

9.1 The static properties of dialogs

The static properties are configured in the UI definition. The following elements have impact on this configuration:

- *ui:enumeration and ui:enum* (→ 111) defines types that can be used for enumerator variables. The name of the ui:enumeration can occur in the `type` attribute of *ui:variable* (→ 170) declarations of the same dialog. See the description of *ui:enumeration and ui:enum* (→ 111) for more explanations, and for an example.
- *ui:variable* (→ 170) declares the instance variables of the dialogs. These variables are persistent, i.e. they do not lose their contents between HTTP invocations. The variables are typed, see the chapter about *Data types* (→ 27) for a description of the (very simple) type system. ui:variable allows it to specify the initial value that is assigned to the variable when the dialog object is created. The variables can be accessed from the UI definition by the element *ui:dynamic* (→ 102), and by all elements that have a `variable` attribute (i.e. *ui:checkbox* (→ 91), *ui:radio* (→ 144), *ui:text and ui:password* (→ 160), *ui:select* (→ 150), *ui:textarea* (→ 163), *ui:ifvar* (→ 124), *ui:iterate* (→ 132), and *ui:enumerate* (→ 108)). The *bracket expressions* (→ 176) can refer to variables, too. The UI definition does not allow any algorithmic modifications of variables; the contents of variables can only be changed by user interaction. However, the application program can both read from and write to variables, see the language- specific sections *Data types (O'Caml)* (→ 32), and *Data types (Perl)* (→ 35) for an overview.
- *ui:page* (→ 135) defines the HTML trees that can be output as visualization of the dialog. The pages can also be regarded as top-level templates, and most of the rules explained in the chapter about *Templates* (→ 49) can actually be applied to pages, too. Templates and pages only differ in the way they are called: While templates are explicitly invoked by UI definition code (ui:use), pages are called by the output routine of the dialog.
- *ui:context* (→ 96) allows one to set context parameters for the scope of the dialog. See the chapter about *Templates* (→ 49) for details.
- Another property of dialogs is the ability to generate events on certain user interactions. There must be a *ui:form* (→ 117) element in the page definition in order to enable this feature, and all interactors that can trigger events must only occur inside this ui:form. These interactors are *ui:a* (→ 80), *ui:button* (→ 87), *ui:imagebutton* (→ 127), and *ui:richbutton* (→ 147). There is a chapter about *Events* (→ 41) explaining the details.

Note that all static properties are determined by the UI definition, and none by the application program.

9.2 Programming dialogs

Once defined in the UI language, the dialogs exist for WDialog. However, there are not yet any actions that can be triggered by the events, and there are not yet any ways to set up the state variables such that the page can be displayed properly. This must be done by programming a Caml class (or Perl class, but the following text assumes Caml) that extends the base class `dialog`. The base class implements the fundamental behaviour of dialogs, but it is incomplete and must be complemented by two very special methods.

The first method has the fixed name `prepare_page`, and it is called just before a page is displayed. The task of this method is, in general, to set the state variables to the right values for the page generation. Remember that the generated HTML code usually refers to variables because of bound interactors, and that the code contains the values of these variable in one or the other way. For example, the variables could be loaded from a database.

The other method has the fixed name `handle`, and it is called just after the user has triggered an event. This method should find out the right action for the event, and perform this action. For example, it could save the user-modified contents of the state variables into the database. Another task of the `handle` method is to determine what happens next. The question is which page is to be displayed next (such that `prepare_page` can be called in turn, and that the page can be generated). It is also possible that `handle` decides to leave the whole dialog, and to continue with another dialog.

The dialog cycle consists of a possibly infinite sequence of `prepare_page` and `handle` calls.

The programmed class must inherit from the base class *Wd_dialog.dialog* (→ 187) with the type `dialog_type` as defined in the module *Wd_types* (→ 205):

```
class my_dialog universe name env =
  object(self)
    inherit Wd_dialog.dialog universe name env

    method prepare_page() = ...

    method handle() = ...
  end
;;
```

The base class provides methods to access the variables. For example, it is possible read the value of a variable by

```
let v = self # variable "varname"
```

and you can set the variable to a new value by

```
self # set_variable "varname" v
```

See the chapter about *Data types (O'Caml)* (→ 32) for more such methods. The definition of `dialog_type`, found in *Wd_types* (→ 205), can be used as reference.

After the dialog class has been defined, it must be registered in the universe. The universe contains which Caml class implements which dialog. In the simplest case, the registration can be done as callback from *Wd_run_cgi.run* (→ 196):

```
Wd_run_cgi.run
  ~reg:(fun universe ->
         universe # register "my_dialog" (new my_dialog)
       )
  ()
```

The idea of the universe is explained below.

9.3 The dynamic properties of dialogs

After dialog objects have been created, the following dynamic properties can be expected:

- The *current page* of the dialog is the name of one of the declared pages (initially the `start-page`). During the `handle` callback there is also the *next page* that is normally the same as the current page, but can be set to a different page by the callback implementation. Of course, the system will switch to the next page as the new current page when the next dialog cycle will be performed.
- The values of the variables can be changed by interactors like *ui:text and ui:password* (→ 160) that are bound to variables, and by explicitly calling the `set_variable` method from the application program.
- The name of the most recent event is automatically stored in the dialog object, and can be accessed by calling the `event` method.
- The dynamic properties of dialogs can be extended by defining the two callback methods `prepare_page` and `handle`. The `prepare_page` callback should set variables that are needed to display the next page, and `handle` should look at the last event and should do the actions implementing the intended meaning of the events. The callback routines can access all static and all dynamic properties of the dialog.

9.4 The universe of dialogs

The remaining question is: how are dialog objects created, and what can the application programmer do to modifiy their behaviour? WDialog has the concept of a *universe* that defines what can be dynamically created, and the universe especially contains the registry of dialog object constructors. Such a constructor is simply a function that returns the new dialog object as result. In particular, constructors have the functional type

```
universe_type -> string -> environment -> dialog_type
```

i.e. the three arguments universe, dialog name (the string), and the environment are passed to the constructor, and the object of type `dialog_type` is returned. Because of this signature, implementations of dialog classes usually take exactly these arguments as class arguments (i.e. `class my_dialog universe name env = ...`), such that the `new` operator can be directly used to create the object (i.e. `new my_dialog` is a working constructor).

At program startup, the universe is initialized, and the application program (usually) puts the constructors for all dialogs into the universe. This is done by the `register` method of the universe (e.g. `universe # register "my_dialog" (new my_dialog)`). Later, the universe is used to create new dialog objects, e.g.

```
let dlg = universe # create env "my_dialog"
```

The new object `dlg` knows all static properties that have been declared in the UI definition, but it still in the initial state regarding the dynamic properties. When the application program creates the new object, it has to ensure that the variables are set up as required by the program logic.

There is another, internal usage of `create`. This method is also used by WDialog to reactivate the current dialog after process startup. The variables, and the other dynamic properties are set to the state they had when the last cycle ended, in order to keep the illusion that the dialog state persists across consecutive cycles.

Data types

Web Path: WDialog / Reference / Data types

10 Data types

The WDialog system has its own type system for the values of instance variables of dialogs. On the one hand, such values must be simple to manipulate from programs (which implies a simple representation); on the other hand, such values must be powerful enough to express the states of interactors.

10.1 The string type

There are at least three applications for string types:

- String variables can be included as dynamic text into generated HTML pages, i.e. the program computes some text to be put at a certain place of the HTML code. This can be achieved by the element *ui:dynamic* (→ 102), or by using *bracket expressions* (→ 176).
- String variables can be tied to interactors which use strings as base type. The most common example are text input boxes which can be created by the *ui:text* (→ 160) element.
- Sections of code can be included or excluded depending on whether a string variable has a certain value; see *ui:if* (→ 119), and *ui:ifvar* (→ 124).

In computer science, there is a common understanding of what strings are; so it is not necessary to explain here their fundamentals. However, we state here that every character can be a member of a string (even null characters) and that strings can be reasonably long (for a typical web application, 1MB might suffice as maximum length). The supported character sets are ISO-8859-1 (Latin 1) and UTF-8 (Unicode).

In variable declarations (*ui:variable* (→ 170)), the string type may be referred to by the literal

```
string
```

In order to denote initial string values of variables, one can use the *ui:string-value* (→ 157) element.

10.2 Declared enumerator types

Several interactors allow the user to choose a set of values from a given base set. For example, the HTML select tag lists several options, and one can individually select which options to choose. For such cases, it is possible to declare an *enumerator*, which is a new type enumerating all possible options. An example (see also *ui:enumeration* (→ 111)):

```
<ui:dialog name="example">
  <ui:enumeration name="sex">
    <ui:enum internal="male"/>
    <ui:enum internal="female"/>
  </ui:enumeration>
```

```
  ...
  <ui:variable name="person's_sex" type="sex"/>
  ...
  <ui:page name="select_your_sex">
     ...
     <ui:select variable="person's_sex" multiple="no"/>
     ..
  </ui:page>
  ...
</ui:dialog>
```

An enumerator has the following properties:

- The type defines a list of literals (here "male" and "female"); also called the internal values. An instance of the type is a subset of these values. (In our example, these subsets will have a cardinality of <= 1 because `multiple="no"` restricts the `ui:select` interactor such that only one item can be selected. Actually only the subsets {}, {"male"} and {"female"} are possible.)
- It is not possible to have the same internal item twice in a subset.
- The items are ordered. This is important for the visualization, because although sets are often meant without order, eventually there must be an order when the items are displayed. (In our example, this means that the selection box will first display "male", and then "female".)

Furthermore, the internal items may have corresponding external values. The internal value is used to identify the item in an algorithm or in a database, and the external value is the corresponding representation when the item is displayed. If not defined, the same literals are used for the internal and the external values. Example:

```
<ui:enumeration name="sex">
  <ui:enum internal="male"   external="You are a man"/>
  <ui:enum internal="female" external="You are a woman"/>
</ui:enumeration>
```

Enumerators have the following applications:

- Enumerator variables can be tied to interactors (*ui:select* (→ 150), *ui:checkbox* (→ 91), *ui:radio* (→ 144)) working on sets of items.
- It is possible to use enumerators as index to associative values (see below), and to iterate over the index domain (see *ui:enumerate* (→ 108)).

In order to specify initial values of enumerator variables, one can apply the *ui:enum-value* (→ 107) element.

10.3 The dynamic enumerator type

Declared enumerators are restricted as the possible items must all be known at the time when they are declared. Sometimes the items are only known at run time, and in this case dynamic enumerators may be the solution.

Like declared enumerators, values of dynamic enumerators are ordered sets of internal items with corresponding external values; the difference is that you may add any pair (int,ext) of internal/external pairs at run time to a dynamic enumerator. In the following example, the user can select a subset of things of a computed base set of things:

```
<ui:dialog name="example">
  ...
  <ui:variable name="selected_things"   type="dynamic-enumerator"/>
  <ui:variable name="possible_things" type="dynamic-enumerator"/>
  ...
  <ui:page name="select_your_things">
     ...
     <ui:select variable="selected_things" base="possible_things"
                multiple="yes"/>
     ..
  </ui:page>
  ...
</ui:dialog>
```

It is assumed that the `prepare_page` method of the dialog class fills the variable `possible_things` in a reasonable way. The *ui:select* (→ 150) interactor needs now two attributes, one (`base`) determining the list of items to display, and the other one (`variable`) containing the subset of items that are actually selected by the user. In contrast to declared enumerators, the latter attribute does not imply the base set, so it is necessary to specify it additionally.

In order to specify initial values of dynamic enumerator variables, one can apply the *ui:dyn-enum-value* (→ 101) element.

10.4 The dialog type

It is possible to declare instance variables whose values are complete dialog objects. An important motivation for this possibility are sub dialogs, i.e. dialogs which can be called from arbitrary other dialogs, and which are able to return to their caller object whatever it was.

Consider the following (incomplete) example:

```
<ui:dialog name="calling_dialog" start-page="...">
  ...
  <ui:variable name="v" .../>
  ...
  <ui:page name="calling_page">
    ...
    As in many other dialogs, you can now go to our
    <ui:a name="call_event">special dialog</ui:a>.
    ...
  </ui:page>
</ui:dialog>

<ui:dialog name="called_dialog" start-page="called_page">
  ...
  <ui:variable name="caller" type="dialog"/>
  ...
  <ui:page name="called_page">
    ...
    You can now do ... this ... and ... that.
    <ui:a name="return_event">Return to previous dialog.</ui:a>
    ...
```

```
  </ui:page>
</ui:dialog>
```

The `handle` methods of both objects must perform special actions, which can be described as follows:

- *calling_dialog:* The `handle` method is invoked when the user clicks on the hyperlink pointing to the "special dialog", and in this case the method receives the `call_event`. The special action is to (1) create the new dialog instance for `called_dialog`, (2) initialize the variable `caller` with the current dialog (which is `calling_dialog`), and (3) change the current dialog seen by the user to `called_dialog`. This causes that the system shows the `called_page` as next page.
- *called_dialog:* The `handle` method is invoked when the user clicks on the hyperlink "Return to previous dialog". The event sent to the method is `return_event`, and the following actions are needed to actually return: (1) Fetch the previous dialog instance from the variable `caller`, and (2) change the current dialog to this instance. This causes that the system shows the last page of this dialog again.

Note that there are no dialog literals in the UI language; variables of dialog type can only be set from the program. It is, however, possible to access the inner variables of a dialog variable by using the dot notation, see the section about *Dot notation (v1.v2)* (→ 180) for details.

10.5 Associative variables

Often, it is necessary to manage lists of interactors. Of course, it is possible to generate the HTML code by instantiating a page fragment several times (you can describe page fragments by *ui:template elements* (→ 158)); however, the question arises how to refer to the first, the second, ... the nth generated interactor. One possibility would be to generate the names of the interactors; but this can be complicated. Because of this, the WDialog system provides a general solution to refer to generated interactors.

All interactor elements know the attribute `index` which can be used as a second identifier besides the name or the variable to which the interactor is bound. For example, the following two buttons will trigger different events when the user clicks at them:

```
<ui:button name="click" index="1" label="Select this"/>
<ui:button name="click" index="2" label="Select that"/>
```

The event structure sent to the `handle` method of the object contains both the name of the button and the index value, if present. Note that index values may be arbitrary strings (the system encodes "unsafe characters" such that they can be safely transported over the HTTP/CGI connection).

Many interactors are tied to variables. For example, the *ui:text* (→ 160) element displays an input box which initially shows the current value of the variable, and when the user modifies the box, the changed value will automatically be transferred back to the variable. This shows that if we want indexed interactors, we need indexed variables to which these interactors can be bound.

In the terminology of WDialog these variables are called *associative*. The value of an associative variable is a mapping from keys (indexes) to values where the (index,value) pairs are ordered (for the same reason why enumerators are ordered: when they are displayed, an order is required).

For example, it is possible to declare an associative variable of strings

```
<ui:variable name="v" type="string" associative="yes"/>
```

and to refer to this variable from input boxes:

```
<ui:text variable="v" index="1"/>
<ui:text variable="v" index="2"/>
...
```

Here, the first input box accesses and modifies the value of `v` which is stored at the key "1", and the second box the value which is stored at the key "2". Both input boxes refer to the same variable, but they actually access different components of the variable.

It is not only possible to declare associative string variables, but also associative enumerators, dynamic enumerators and dialogs.

In order to specify default values of associative variables, one can use the *ui:alist-value* (→ 84) element.

Data types (O'Caml)

Web Path: WDialog / Reference / Data types / Data types (O'Caml)

11 The representation of the data types in O'Caml

In the O'Caml environment, the dialog data types are represented as follows:

```
type var_value =
    String_value   of string
  | Enum_value     of string list
  | Dialog_value   of dialog_type option
  | Dyn_enum_value of (string * string) list
  | Alist_value    of (string * var_value) list
```

(You can find this in the module *Wd_types* (→ 205).) This leads to the following value literals:

- *Strings:* A string with value s is written as

  ```
  String_value s
  ```

- *Declared enumerators:* An enumerator value that contains the internal items x1, x2, ... is written as

  ```
  Enum_value [x1; x2; ...]
  ```

  The items may be in any order; when an enumerator value is displayed by an interactor, the items are rearranged according to the declared order (as found in the ui:enumeration element). The enumerator value is checked for compatibility with the declaration when an instance variable is set to the value by the `set_variable` method (i.e. it is checked whether only declared items occur in the passed list of items, and whether they occur only once).

- *Dynamic enumerators:* A dynamic enumerator value that contains the internal items x1, x2, ... and the corresponding external items y1, y2, ... is written as

  ```
  Dyn_enum_value [ (x1,y1); (x2,y2); ... ]
  ```

  The order of the items specify the order of the enumerator. When an instance variable is set to such a value it is checked whether every internal item occurs only once.

- *Dialogs:* A dialog `dlg` is written as variable value in the following way:

  ```
  Dialog_value (Some dlg)
  ```

  Note that there is also the non-existing dialog which is written as

  ```
  Dialog_value None
  ```

  (The system uses this value as default values for variables of type `dialog`; there is simply no other reasonable default.)

- *Associative values:* An associative list with the keys k1, k2, ... and the corresponding values v1, v2, ... is written as

  ```
  Alist_value [ (k1,v1); (k2,v2); ... ]
  ```

  The order of the pairs specify the order of the list. When an instance variable is set to such a value it is checked whether every key occurs only once.

11.1 Getting values of variables

Let `dlg` be the current dialog object in the following examples. In order to get a variable of arbitrary type one can invoke the `variable` method:

```
let v = dlg # variable "name" in ...
```

After that, `v` is a `var_value` (i.e. something like `String_value`, `Enum_value` etc.). - If you already know that the variable is a string, you can also call:

```
let s = dlg # string_variable "name" in ...
```

Now `s` is directly an O'Caml string (and not a `String_value`). - For the other types, there are direct accessor methods, too:

```
let enum     = dlg # enum_variable "name1" in ...
let dyn_enum = dlg # dyn_enum_variable "name2" in ...
let dlg'     = dlg # dialog_variable "name3" in ...
```

For associative lists, there are several possibilities to get values in a more convenient way. First, one can directly get the defining list:

```
let alist = dlg # alist_variable "name" in ...
```

Furthermore, it is possible to look up a component "key" of the list by one of the following invocations (depending on the base type):

```
let s        = dlg # lookup_string_variable "name" "key1" in ...
let enum     = dlg # lookup_enum_variable "name" "key2" in ...
let dyn_enum = dlg # lookup_dyn_enum_variable "name" "key3" in ...
let dlg'     = dlg # lookup_dialog_variable "name" "key4" in ...
```

11.2 Setting values of variables

There is only one method to set a variable to a value `v`:

```
dlg # set_variable "name" v
```

Constructing a `var_value` from a base value is already very simple, such that there is no need for methods setting the base value directly. For example, if a variable must be set to a string `s`, the following elegant notation is possible:

```
dlg # set_variable "name" (String_value s)
```

11.3 Resetting values of variables

There is another method that resets a variable to its initial value:

```
dlg # unset_variable "name"
```

11.4 The dot notation

As a more convenient method to access variables of inner dialogs, it is possible to refer to variables by the dot notation. For example, if there is a dialog-type variable `d`, and the string variable `s` is defined within the dialog that is currently the value of `d`, you can get the string contents of `s` by this expression:

```
let s = dlg # string_variable "d.s"
```

This is a shorthand notation for:

```
let s =
  match dlg # dialog_variable "d" with
    None -> failwith "Dialog variable is empty!"
  | Some d -> d # string_variable "s"
```

See the section about *Dot notation (v1.v2)* (→ 180) for a broader discussion of this topic.

Data types (Perl)

Web Path: WDialog / Reference / Data types / Data types (Perl)

12 The representation of the data types in Perl

In the Perl environment, the dialog data types are represented as Perl classes:

- *UI::Variable::String* is the class representing the string type
- *UI::Variable::Enum* is the class representing the type of declared enumerators
- *UI::Variable::DynEnum* is the class representing the type dynamic-enumerator
- *UI::Variable::Dialog* is the class representing the dialog type
- *UI::Variable::Alist* is the class representing associative lists

Before you can access these classes, you must load them:

```
use UI::Variable;
```

- loads all classes at once.

In the following explanations, the objects of these classes are called "value containers", as they serve as containers for a low-level representation of values[5].

12.1 The interfaces of the value containers

12.1.1 UI::Variable::String

A string container can be created by

```
my $s_container = new UI::Variable::String("The string");
```

The string of such a container can be read by calling

```
my $s = $s_container->value;    # returns "The String"
```

Note that it is normally not necessary to use string containers as there are special access methods for string variables (`string_variable` and `set_string_variable`).

[5]The containers contain methods to map the contained values to the corresponding O'Caml value, and to map O'Caml values back. Actually, the O'Caml values are the low-level representation from the view of the Perl bindings.

12.1.2 UI::Variable::Enum

A container for a declared enumerator can be formed by

```
my $e_container = new UI::Variable::Enum("x1","x2",...);
```

where x1,x2,... are the internal items of the enumerator. If necessary, it is possible to add a new internal item to an already existing container:

```
$e_container->add("x3");
```

The list of internal items of an enumerator can be read by

```
my @items = $e_container->value;  # returns ("x1","x2","x3")
```

The number of items:

```
my $n = $e_container->length;
```

To iterate over the items of a container, apply this method:

```
$e_container->iter( sub { my $item = shift; .... } )
```

12.1.3 UI::Variable::DynEnum

A dynamic enumerator consisting of the internal items x1, x2, .. and the corresponding external values y1, y2, ... can be formed by:

```
my $d_container = new UI::Variable::DynEnum(["x1","y1"],["x2","y2"],...);
```

There is also a method to add another pair:

```
$d_container->add("x3","y3");
```

The list of pairs can be requested by calling:

```
my @pairs = $d_container->value;
```

In this example, `@pairs` is equal to the list

```
(["x1","y1"],["x2","y2"],["x3","y3"])
```

The number of pairs:

```
my $n = $d_container->length;
```

To iterate over the items of a container, apply this method:

```
$d_container->iter( sub { my ($x,$y) = @_; .... } )
```

Here, `$x` is the interal and `$y` the external value of the current pair that is being visited during the iteration.

12.1.4 UI::Variable::Dialog

An UI::Dialog instance `$dlgobj` can be put into a value container by

```
my $o_container = new UI::Variable::Dialog($dlgobj);
```

To get the dialog back, do

```
my $dlgobj = $o_container->value;
```

12.1.5 UI::Variable::Alist

An associative list of values is handled like a dynamic enumerator. An alist container consisting of the keys i1,i2,... and the corresponding values v1,v2,... can be created by

```
my $a_container = new UI::Variable::Alist( ["i1",$v1], ["i2",$v2], ... );
```

Here, i1, i2, etc must be strings, and `$v1`, `$v2`, etc must be value containers (`UI::Variable::String`, or `-::Enum`, or `-::DynEnum`, or `-::Object`). It is possible to add another key/value mapping:

```
$a_container->add("i3",$v3);
```

The list of pairs can be requested by calling:

```
my @pairs = $a_container->value;
```

In this example, `@pairs` is equal to the list

```
(["i1",$v1],["i2",$v2],["i3",$v3])
```

The number of pairs:

```
my $n = $a_container->length;
```

To iterate over the items of a container, apply this method:

```
$a_container->iter( sub { my ($i,$v) = @_; .... } )
```

Here, `$i` is the index (key) and `$v` the corresponding value of the current pair that is being visited during the iteration.

12.2 Getting values of variables

Let `$ui` be the `UI::Dialog` in the following examples. In order to get a variable of arbitrary type one can invoke the `variable` method:

```
my $v = $ui->variable("name");
```

This method returns a value container, i.e. one object of the classes `UI::Variable::String`, `-::Enum`, `-::DynEnum`, `-::Dialog`, or `-::Alist`. For instance, if the object is a string, the scalar value of the string can be obtained by:

```
my $s = $ui->variable("name")->value;
```

For convenience, the types string and enumerator are supported specially. To read a string scalar, one can alternatively invoke:

```
my $s = $ui->string_variable("name");
```

Enumerator values can be accessed by calling:

```
my @e = $ui->enum_variable("name");
```

`@e` will be the list of internal items.

12.3 Setting values of variables

The method set_variable changes the value of a variable. You must pass the name of the variable and the value container `$v` to the method:

```
$ui->set_variable("name", $v);
```

For example, to set the value of a string variable to `"happy"`, the following statement can be executed:

```
$ui->set_variable("name", new UI::Variable::String("happy"));
```

For convenience, there is a "shortcut method" for strings:

```
$ui->set_string_variable("name", "happy");
```

12.4 Resetting values of variables

The following method sets the value of a variable back to its initial value - either the default specified in the *ui:variable* (→ 170) element, or the null value of the type:

```
$ui->unset_variable("name");
```

12.5 The dot notation

As a more convenient method to access variables of inner dialogs, it is possible to refer to variables by the dot notation. For example, if there is a dialog variable `d`, and the string variable `s` is defined in the dialog that is currently the value of `d`, you can get the string contents of `s` by this expression:

```
my $s = $ui->string_variable("d.s");
```

This is a shorthand notation for:

```
my $d = $ui->variable("d")->value();
die "Dialog is empty!" if (!defined($d));
my $s = $d->string_variable("s");
```

See the section about *Dot notation (v1.v2)* (→ 180) for details.

Events

Web Path: WDialog / Reference / Events

13 Events

From an abstract point of view, when an event is triggered an event description is sent to the current dialog object which can react on it. In reality, the process is much more complicated; the WWW browser triggers the event which causes that a HTML form is sent to the server. The server decodes the form, reconstructs the current dialog, analyzes which event has happened, and finally invokes the event handling method of the dialog object.

The following elements can trigger events when the user clicks the interactors they create:

- The *button element* (→ 87) with "name", without "index":

  ```
  <ui:button name="n" ...>
  ```

  This type of event is called *button event*; the only parameter of such an event is the name of the button.

- The *button element* (→ 87) with "name" and "index":

  ```
  <ui:button name="n" index="x" ...>
  ```

  This type of event is called *indexed button event*; the parameters are the name of the button and the index value.

- The *imagebutton element* (→ 127) with "name", without "index":

  ```
  <ui:imagebutton name="n" ...>
  ```

  This type of event is called *imagebutton event*; the parameters are the name of the button and the coordinates of the click.

- The *imagebutton element* (→ 127) with "name" and "index":

  ```
  <ui:imagebutton name="n" index="x" ...>
  ```

  This type of event is called *indexed imagebutton event*; the parameters are the name of the button, the index value, and the coordinates of the click.

- The *anchor element* (→ 80) with "name", without "index":

  ```
  <ui:a name="n" ...>
  ```

  This type of event is called *hyperlink event*; the only parameter of such an event is the name of the anchor.

- The *anchor element* (→ 80) with "name" and "index":

```
<ui:a name="n" index="x" ...>
```

  This type of event is called *indexed hyperlink event*; the parameters are the name of the anchor and the index value.

Besides the events resulting from interactors, there are two additional events:

- When a server popup window has been opened, a *popup request* is triggered; it normally causes that the popup page is generated and displayed in the new window.
- Of course, it is possible that the browser submits the HTML form because of a Javascript statement[6]. In this case, the *null event* is generated (which may actually be a real event that does not fall into one of the event categories).

As already explained, the system reconstructs the dialog object that was active when the current page was generated. Furthermore, the variables are updated which are tied to interactors, i.e. the variables reflect the values as they have been edited by the user.

13.1 Handling events

In order to catch an event, one must define/override the `handle` method of the dialog class. Please see the language-specific interface descriptions for further details.

13.2 Actions

In principle, there are three different actions that may happen as a consequence of an event:

- The dialog sets its instance variables to different values, and generates the same page for the new values of variables.
- The dialog sets the instance variables, but changes the page to display.
- The dialog switches to a second dialog, and asks it to generate the next page.

[6] E.g. by executing `document.uiform.submit()`

Events (O'Caml)

Web Path: WDialog / Reference / Events / Events (O'Caml)

14 The representation of events in O'Caml

In the module *Wd_types* (→ 205), the event type is defined as follows:

```
type event =
    Button of string
  | Image_button of (string * int * int)
  | Indexed_button of (string * string)
  | Indexed_image_button of (string * string * int * int)
  | No_event
  | Popup_request of string
```

The variants of the type correspond to the event types:

- *Button event:* A button event for the button n is represented as

  ```
  Button n
  ```

- *Indexed button event:* A button event for the button n with index i is represented as

  ```
  Indexed_button(n,i)
  ```

- *Imagebutton event:* An imagebutton event for the button n is represented as

  ```
  Image_button(n,x,y)
  ```

  The coordinate (x,y) is the position of the click relative to the image.

- *Indexed imagebutton event:* An imagebutton event for the button n with index i is represented as

  ```
  Indexed_image_button(n,i,x,y)
  ```

  The coordinate (x,y) is the position of the click relative to the image.

- *Hyperlink event:* A hyperlink event for the anchor n is represented as

  ```
  Button n
  ```

  The representations for hyperlinks and for buttons are intentionally the same.

- *Indexed hyperlink event:* A hyperlink event for the anchor n with index i is represented as

  ```
  Indexed_button(n,i)
  ```

  The representations for hyperlinks and for buttons are intentionally the same.

- *Popup request:* A popup request is represented as

  ```
  Popup_request s
  ```

Note that the current page is changed to the page popping up while the popup request is processed.

The parameter s of the popup request is the second argument of the generated `open` Javascript function (to simplify parameterized popup windows). For example, a popup window declared with

```
<ui:server-popup page="x"/>
```

can be opened by the Javascript statement

```
open_x(window_specification, s);
```

The parameter `s` is passed back to the application once the window pops up and the contents of the window are requested. You can use this parameter arbitrarily.

Popups are explained in conjunction with *ui:server-popup* (→ 154).

- *Null event:* The null event is represented as

```
No_event
```

14.1 Handling events

The `handle` method of the current dialog is called when the event has been triggered. The method `event` can be used to find out the last event, e.g.

```
method handle =
  let e = self # event in
  match e with
    Button("this_button") ->
      ...
  | Button("that_button") ->
      ...
  | Indexed_image_button("clicked_icon", x, y) ->
      ...
  | _ ->
      (* It is recommended to do nothing as default *)
      ()
```

When the `handle` method returns normally, the current page of the current dialog is generated again. Alternatively, another page can be set by raising the exception `Change_page`; or it can be changed to another dialog instance by raising the exception `Change_dialog`. Example:

```
method handle =
 let e = self # event in
 match e with
   Button("this_button") ->
     self # set_variable "xy" (String_value "wow");
     (* ... and continue with the same page *)
```

```
| Button("that_button") ->
    self # set_variable "xy" (String_value "boo");
    raise(Change_page "another_page")
| Indexed_image_button("clicked_icon", x, y) ->
    let next_dlg =
       self # universe # create (self#environment) "other_dialog" in
    raise(Change_dialog next_dlg)
```

Events (Perl)

Web Path: WDialog / Reference / Events / Events (Perl)

15 The representation of events in Perl

Events are represented as arrays whose components contain the properties of the events.

- *Button event:* A button event for the button `$n` is represented as

  ```
  ( "BUTTON", $n )
  ```

- *Indexed button event:* A button event for the button `$n` with index `$i` is represented as

  ```
  ( "INDEXED_BUTTON", $n, $i )
  ```

- *Imagebutton event:* An imagebutton event for the button `$n` is represented as

  ```
  ( "IMAGE_BUTTON", $n, $x, $y )
  ```

  The coordinate (`$x`,`$y`) is the position of the click relative to the image.

- *Indexed imagebutton event:* An imagebutton event for the button `$n` with index `$i` is represented as

  ```
  ( "INDEXED_IMAGE_BUTTON", $n, $i, $x, $y )
  ```

  The coordinate (`$x`,`$y`) is the position of the click relative to the image.

- *Hyperlink event:* A hyperlink event for the anchor `$n` is represented as

  ```
  ( "BUTTON", $n )
  ```

  The representations for hyperlinks and for buttons are intentionally the same.

- *Indexed hyperlink event:* A hyperlink event for the anchor `$n` with index `$i` is represented as

  ```
  ( "INDEXED_BUTTON", $n, $i )
  ```

  The representations for hyperlinks and for buttons are intentionally the same.

- *Popup request:* A popup request is represented as

  ```
  ( "POPUP_REQUEST", $s )
  ```

  Note that the current page is changed to the page popping up while the popup request is processed. - For the meaning of the parameter `$s` see the explanations in the O'Caml API.

- *Null event:* The null event is represented as

  ```
  ( "NO_EVENT" )
  ```

Note that the first component is always the type of the event and that the second component is always the name of the event (if any). This constraint will even hold if the list of possible events will be extended in the future.

15.1 Handling events

The `handle` method of the current dialog object is called when the event has been triggered. The method `event` can be used to find out the last event, e.g.

```
sub handle {
    my ($self) = shift;
    my @e = $self->event;

    my $e_name = $e[1];    # the second component is the name

    if ($e_name eq 'this_button') {
        ...
    } elsif ($e_name eq 'that_button') {
        ...
    } elsif ($e_name eq 'clicked_icon') {
        my $x = $e[2];
        my $y = $e[3];
        ...
    } else {
        # It is recommended to do nothing as default case
    };

    return undef;       # Important!
}
```

The value returned by the `handle` method determines the action performed after the event has been processed. An undefined value means to display the same page again. Note that you *must* include a `return undef` statement as last statement of the method; otherwise the value that happens to be at the top of the value stack is returned. Alternatively, another page can be set by returning the name of the page as string (`return 'other_page'`). Last but not least it is also possible to change completely to a different dialog instance by returning the reference to the dialog. Example:

```
sub handle {
    my ($self) = shift;
    my @e = $self->event;

    my $e_name = $e[1];    # the second component is the name

    if ($e_name eq 'this_button') {
        $self->set_string_variable("xy", "wow");
        return undef;
    } elsif ($e_name eq 'that_button') {
        $self->set_string_variable("xy", "boo");
        return "another_page";
    } elsif ($e_name eq 'clicked_icon') {
        my $next_dlg = UI::Universe::create("other_dialog");
        return $next_dlg;
    } else {
        # It is recommended to do nothing as default case
    };
```

```
    return undef;       # Important!
}
```

Templates

Web Path: WDialog / Reference / Templates

16 Templates

Templates are well-formed HTML fragments containing placeholders (parameters). They are defined outside the dialogs, but can be instantiated within the page definitions of the dialogs. When the template is instantiated, the (formal) parameters are replaced by the passed values, resulting in a complete HTML subterm. Of course, a template can be instantiated several times with different instance values.

16.1 Scope of templates

Templates are always globally known. For example, in the following application every page can refer to every defined template:

```
<ui:application start-object="...">

  <ui:template name="t1">
  ...
  </ui:template>


  <ui:dialog name="o1" ...>
    ...
    <ui:page name="p1">
       ...
       <!-- Instantiations of t1 and t2 are allowed here -->
       ...
    </ui:page>
    ...
  </ui:dialog>


  <ui:template name="t2">
  ...
  </ui:template>
</ui:application>
```

Templates can be instantiated from within *ui:page* (→ 135) definitions, and applying a template has the same effect as if the expanded HTML fragment had been written in place of the application. It is also possible to instantiate templates from within templates; however, recursive instantiation is not allowed.

16.2 Definition and instantiation of templates without parameters

A template is defined by the *ui:template* (→ 158) element, and the simplest way to apply it is the *ui:use* (→ 168) element. For example, the template `t` is defined as:

```
<ui:template name="t">
This is text from template t.
</ui:template>
```

The following page applies this template twice, resulting in a page displaying "This is text from template t. This is text from template t.":

```
<ui:page name="p">
  <html>
    <body>
      <ui:use template="t"/>
      <ui:use template="t"/>
    </body>
  </html>
</ui:page>
```

There are some special whitespace rules for template definitions. The template `t` could be read as "\nThis is text from template t.\n" because there are newline characters before "This" and after the period. However, WDialog ignores whitespace at the beginning and at the end of template definitions, so you can nicely format your definition. (See below for techniques that force the inclusion of spaces at these special locations.)

16.3 Templates with parameters

The placeholders of templates are called *parameters*, and they may occur either within character data, or within attributes. They are denoted by a dollar followed by the identifier, or by a brace containing the identifier (as in shell expressions). Example:

```
<ui:template name="three_columns" from-caller="col1 col2 col3">
  <table>
    <tr>
      <td>$col1</td>
      <td>$col2</td>
      <td>$col3</td>
    </tr>
  </table>
</ui:template>
```

This template expands to a table with one row and three columns, and the contents of all cells are passed by parameters to the template. The parameters are called `col1`, `col2`, and `col3`, and they must be declared by the `from-caller` attribute. The replacement texts of the parameters are inserted where the parameters are referred to by the dollar notation; here

as `$col1`, `$col2`, `$col3` within the `td` element. Note that it is also possible to put the names into curly braces such as `${col1}` - this is especially necessary if the parameter names consist of characters other than a-z, A-Z, 0-9, and _.

Note that the dollar character must be either written as `$$` in template definitions, in order to denote the dollar character as such.

The `from-caller` declaration is needed for every parameter which is referred to using the dollar notation. One reason for this rule is to make it more likely that typos in parameter names are recognized as errors and that such templates are rejected by the system. Furthermore, `from-caller` also indicates that the parameters have so-called lexical scope. Alternatively, parameter may also declared by an `from-context` attribute with a different scoping rule (see below).

The actual values for the parameters are passed by *ui:param* (→ 140) elements which may be included into `ui:use` applications:

```
<ui:use template="three_columns">
  <ui:param name="col1">This is the left column!</ui:param>
  <ui:param name="col2"><b>This is the middle column!</b></ui:param>
  <ui:param name="col3">The right edge.</ui:param>
</ui:use>
```

Note that it is possible to pass whole XML subterms (as in `col2`), and not only plain texts.

Rule to process inner elements when expanding parameters in character data context:

- The passed element tree is inserted where the dollar notation occurs. Further expansions are performed within the inserted tree after the insertion has happened (lazy evaluation).

The consequences of this rule are discussed later.

The dollar notation can be used where normal text is allowed as in the example above, or within attributes. For instance, the example can be extended by passing the alignment attributes of the `td` cells:

```
<ui:template name="three_columns"
             from-caller="col1 col2 col3 align1 align2 align3
                          valign1 valign2 valign3">
  <table>
    <tr>
      <td align="$align1" valign="$valign1">$col1</td>
      <td align="$align2" valign="$valign2">$col2</td>
      <td align="$align3" valign="$valign3">$col3</td>
    </tr>
  </table>
</ui:template>
```

For simple, unstructured texts the dollar notation behaves in the same way when used within attributes as when applied in the body of elements. However, there are differences regarding real subtrees. Attributes cannot represent inner elements; for example, it is not reasonable to pass the element `<b>top</b>` as parameter `valign1`. Because of this, the following rules are applied to remove/process inner elements:

Rules to process inner elements when expanding parameters in attribute context:

- If there is a special processing rule for the element, the rule will be applied. There are only few elements defining such a rule, the element *ui:dynamic* (→ 102) is among them. When used in the replacement text within attributes, `ui:dynamic` expands to the value of the specified variable, and the result of the expansion is included into the attribute value (i.e. it works in the usual way). There is an important application; you can pass the current value of a string variable as parameter that is used within attributes. For example, if there is a string variable `a1` containing the alignment for the left column, this `ui:use` statement passes the contents of `a1` to the template:

```
<ui:use template="three_columns">
  <ui:param name="align1"><ui:dynamic variable="a1"/></ui:param>
  ... <!-- other ui:param statements -->
</ui:use>
```

  (However, there is a lighter notation with the same effect; see below.)

- All other elements are included as plain text. For example, if the *element* "top" is passed as parameter value, the expansion is the *string* "top".

16.4 Parameters with default values

The above definition of `three_columns` is a bit impractical because whenever the template is instantiated all nine parameters must be passed. This can be avoided by providing defaults for parameters. The default values are simply specified as inner text of the `ui:default` declaration. Continuing our example:

```
<ui:template name="three_columns"
             from-caller="col1 col2 col3 align1 align2 align3
                          valign1 valign2 valign3">
  <ui:default name="align1">left</ui:default>
  <ui:default name="align2">left</ui:default>
  <ui:default name="align3">left</ui:default>
  <ui:default name="valign1">middle</ui:default>
  <ui:default name="valign2">middle</ui:default>
  <ui:default name="valign3">middle</ui:default>
  <table>
    <tr>
      <td align="$align1" valign="$valign1">$col1</td>
      <td align="$align2" valign="$valign2">$col2</td>
      <td align="$align3" valign="$valign3">$col3</td>
    </tr>
  </table>
</ui:template>
```

Now, only `col1`, `col2`, and `col3` must be passed, and the other parameters may be passed or omitted.

The `ui:default` declarations must be written at the beginning of the template; between two such declarations only white space and comments are allowed. The following rule helps avoiding extra white space in the expansion of templates.

White space at the beginning of templates:

- White space between the `ui:template` start tag and the first `ui:default` start tag is ignored. Furthermore, all white space between consecutive `ui:default` elements is ignored, and white space after the last `ui:default` end tag is ignored.

For our `three_columns example`, this rule means that the first relevant member of the template is the `table` start tag, all white space before this tag is ignored.

There is a corresponding rule for white space at the end of templates: White space before the end tag of `ui:template` is ignored.

However, what to do if I want whitespace? For example, can I define a template only containing a single white space character? Yes, it is possible, but only with a trick. Note that the following trials *do not work:*

```
<ui:template name="space"> </ui:template>

<ui:template name="space">&#32;</ui:template>

<ui:template name="space"><!-- --> <!-- --></ui:template>

<ui:template name="space"><![CDATA[ ]]></ui:template>
```

They do not work because the used XML parser normalizes white space before the WDialog transformation engine gets the XML tree. So these definitions look all the same for WDialog. The solution is to include a reference to an empty template in the definition. The standard library for templates defines the empty template as `wd-null`:

```
<ui:template name="space">
  <ui:use name="wd-null"></ui:use> <ui:use name="wd-null"></ui:use>
</ui:template>
```

16.5 Templates calling templates

Of course, it is possible that a template instantiates another template. Example:

```
<ui:template name="mk_hyperlink" from-caller="href">
  <a href="$href">$href</a>
</ui:template>

<ui:template name="caml_homepage">
  <ui:use template="mk_hyperlink">
    <ui:param name="href">http://caml.inria.fr/</ui:param>
  </ui:use>
</ui:template>
```

The rules for passing parameters may become inconvenient when parameters must be passed from one template to the next template. For instance, if we also want to be able to specify the target frame of the hyperlink, we must write:

```
<ui:template name="mk_hyperlink" from-caller="href target">
  <a href="$href" target="$target">$href</a>
</ui:template>
```

```
<ui:template name="caml_homepage" from-caller="target">
  <ui:use template="mk_hyperlink">
    <ui:param name="href">http://caml.inria.fr/</ui:param>
    <ui:param name="target">$target</ui:param>
  </ui:use>
</ui:template>
```

The parameter `target` is simply passed through from `caml_homepage` to `mk_hyperlink`. It is possible to avoid such stupid forwarding of values by using parameters with dynamic scope; see below.

Note that it is not allowed that a template instantiates itself recursively.

Another way of interaction between templates is that the passed parameter value contains another `ui:use` statement. For example, we can put the hyperlink to the Caml homepage into the middle cell of the three column scheme:

```
<ui:use template="three_columns">
  <ui:param name="col1"> </ui:param>
  <ui:param name="col2">
    <ui:use template="caml_homepage">
      <ui:param name="target">_blank</ui:param>
    </ui:use>
  </ui:param>
  <ui:param name="col3"> </ui:param>
</ui:use>
```

This seems to be straight-forward; however it is important to mention that `ui:use` is resolved lazily. This means that first the template `three_columns` is expanded, leading to this intermediate result:

- Step 1:

```
<table>
  <tr>
    <td align="left" valign="middle"> </td>
    <td align="left" valign="middle">
      <ui:use template="caml_homepage">
        <ui:param name="target">_blank</ui:param>
      </ui:use>
    </td>
    <td align="left" valign="middle"> </td>
  </tr>
</table>
```

In contrast to this, a direct evaluation scheme would first expand `caml_homepage` and pass the result of this first step to `three_columns`. However, this scheme has not been implemented. (I do not want to argue that the lazy scheme is better, it was only simpler to implement. Evaluation depends on whether the parameter occurs in character data or attribute context; for a direct scheme additional analysis would be necessary to find out which context applies (or the replacement text is always computed for both cases which is time-consuming); it might be worth-while to switch to a direct scheme in order to reduce the total number of expansions, however.)

The further expansion steps are:

- Step 2:

```
<table>
  <tr>
    <td align="left" valign="middle"> </td>
    <td align="left" valign="middle">
      <ui:use template="mk_hyperlink">
        <ui:param name="href">http://caml.inria.fr/</ui:param>
        <ui:param name="target">_blank</ui:param>
      </ui:use>
    </td>
    <td align="left" valign="middle"> </td>
  </tr>
</table>
```

- Step 3:

```
<table>
  <tr>
    <td align="left" valign="middle"> </td>
    <td align="left" valign="middle">
      <a href="http://caml.inria.fr/" target="_blank">http://caml.inria.fr/</a>
    </td>
    <td align="left" valign="middle"> </td>
  </tr>
</table>
```

16.6 Calling templates indirectly

The following technique demonstrates how to call a template by passing the name of the template:

```
<ui:template name="for_caml" from-caller="templname">
  <ui:use template="$templname">
    <ui:param name="href">http://caml.inria.fr/</ui:param>
    <ui:param name="target">_blank</ui:param>
  </ui:use>
</ui:template>
```

This template calls the template `$templname`, and passes the fixed set of parameters `href` and `target` with a fixed set of values to the invoked template. A possible way to use it:

```
<ui:use name="for_caml">
  <ui:param name="templname">mk_hyperlink</ui:param>
</ui:use>
```

This code creates again the Caml hyperlink. One possible application for indirect calls is to dynamically select the template to use:

```
<ui:use name="for_caml">
  <ui:param name="templname">><ui:dynamic variable="templname"/></ui:param>
</ui:use>
```

We could have another template, `no_hyperlink`, which simply displays the `$href` parameter without making the hyperlink; the variable `templname` selects the template. We could set this variable in the `prepare_page` method of the dialog object, and make the selection of the particular template dependent on an arbitrary condition.

16.7 A better notation to reference dialog variables

In the last example, we replaced the parameter `templname` by the contents of the dialog variable `templname`. There is a better notation than using `ui:dynamic`:

```
<ui:use name="for_caml">
  <ui:param name="templname">$[templname]</ui:param>
</ui:use>
```

This means exactly the same. Moreover, the square brackets notation can be used inside of attributes (where `ui:dynamic` cannot be applied).

Example: The following template is a variant of `mk_hyperlink` that extracts the values for the URL and the target from dialog variables:

```
<ui:template name="mk_hyperlink">
  <a href="$[href]" target="$[target]">$[href]</a>
</ui:template>
```

In recent versions of WDialog, the bracket notation has been generalized, and it is now allowed to write more complex expressions inside the brackets. See the chapter about *$[expr]* (→ 176).

16.8 Lexical and dynamic scope

When templates call templates, it is often necessary to pass parameters through from one template to the next one. Until now, we only have the solution to do this parameter forwarding explicitly:

```
<ui:template name="t1" from-caller="p">
  ...
    <ui:use template="t2">
      ...
      <ui:param name="p">$p</ui:param>
      ...
```

```
    </ui:use>
  ...
</ui:template>
```

The parameter `p` is introduced at the beginning of `t1`, and `p` gets a value at this moment (either because a value has been passed, or because there is a default value). `p` is visible everywhere within the definition text of `t1`, but it is not automatically visible in called templates such as `t2`, even if there is a parameter with the same name. We must explicitly pass p to subsequent templates to extend its scope. This way of handling the visibility of parameters is called the lexical scoping rule.

This rule has the advantage that it can be exactly controlled which parameter is passed to which template, and it works in most cases fine.

However, if many parameters must be simply forwarded to inner templates, a dynamic scope will better fit to the situation. To explain it, we need the concept of a *dynamic parameter context*. Such a context is a binding of some parameter names to values, and it is automatically passed to inner templates. We can add a particular binding of a name to a value to the context for a certain period of time, and the new binding will hide any previous binding of the same name while it is in effect.

The improved definitions of `mk_hyperlink` and `caml_homepage` are:

```
<ui:template name="mk_hyperlink" from-caller="href" from-context="target">
  <a href="$href" target="$target">$href</a>
</ui:template>

<ui:template name="caml_homepage">
  <ui:use template="mk_hyperlink">
    <ui:param name="href">http://caml.inria.fr/</ui:param>
  </ui:use>
</ui:template>
```

`mk_hyperlink` now gets the parameter `target` from the current context. `caml_homepage` no longer passes this parameter. This template should now be called as follows:

```
<ui:context>
  <ui:param name="target">_blank</ui:param>
  <ui:use template="caml_homepage"></ui:use>
</ui:context>
```

The `ui:context` element extends the context by the parameters denoted by `ui:param` and expands its body, here the `ui:use` application.

The context parameters are a like a set of background definitions that are in effect for the time of the `ui:context` expansion. Every template called from within `ui:context` can access these parameters by importing them with `from-context`.

16.9 The rules of parameter passing

When a template is expanded, the definition text of the template must only consist of declared parameters, i.e. there must be only a dollar notation for a parameter that has been declared by `from-caller` or `from-context` at the beginning of the template.

A parameter is either lexical or dynamic. It is not allowed that the same name appears in both `from-caller` and `from-context`.

For lexical parameters (`from-caller`), only the `ui:param` elements of the calling `ui:use` are searched for instance values.

For dynamic parameters (`from-context`), only the context is searched for instance values. If several `ui:context` elements are in effect for same parameter, the most recent definition wins and is used.

After the values have been collected, all dollar notations are replaced by their corresponding values.

16.10 Encodings

It is sometimes useful to encode the current value of a parameter. For example, imagine you have a template `print-as-html` that prints the HTML code of the parameter `body`:

```
<ui:template name="print-as-html" from-caller="body">
The HTML code is as follows:
<pre>
${body/html}
</pre>
</ui:template>
```

This has the effect that the encoding "html" is applied to the value of `body`. "html" replaces < by < etc. There are a number of other encodings (see the chapter on *Output encodings* (→ 71)).

16.11 The t and p namespaces

Because `ui:use` is a quite long notation, there are two ways to abbreviate it. Instead of

```
<ui:use template="x">
  <ui:param name="p1">t1</ui:param>
  ...
  <ui:param name="pN">tN</ui:param>
</ui:use>
```

you can also write

```
<t:x>
  <p:p1>t1</p:p1>
  ...
  <p:pN>tN</p:pN>
</t:x>
```

Furthermore, the parameters can also be passed as attributes if they only consist of unstructured text:

```
<t:x p1="t1" ... pK="tK">
  <p:pJ>tJ</p:pJ>
  ...
  <p:pN>tN</p:pN>
</t:x>
```

16.12 The q namespace

The other way to abbreviate `ui:use` is the `q` namespace. Instead of writing

```
<ui:use template="x">
  <ui:param name="p1">t1</ui:param>
  ...
  <ui:param name="pN">tN</ui:param>
  <ui:param name="body">tBODY</ui:param>
</ui:use>
```

(note the fixed name `body`) it is also possible to call the template by:

```
<q:x p1="t1" ... pK="tK">
  tBODY
</q:x>
```

16.13 Elements related to templates

The following elements have a relationship to templates:

- *ui:template* (→ 158) defines templates
- *ui:default* (→ 97) declares defaults for template parameters
- *ui:context* (→ 96) extends the current context by adding dynamic parameters
- *ui:use* (→ 168) instantiates templates once
- *ui:param* (→ 140) passes parameters explicitly to called templates
- *ui:iterate* (→ 132) and *ui:enumerate* (→ 108) instantiate templates more than once
- *ui:page* (→ 135) behaves like a template, i.e. you can also use the dollar notation inside pages

Template API (O'Caml)

Web Path: WDialog / Reference / Templates / Template API (O'Caml)

17 The Template API for O'Caml

Templates can be accessed by the O'Caml code of the application. The module Template has the following signature:

```
exception Template_not_found of string

type template
type tree

val get          : application_type -> string -> template
val apply        : dialog_type -> template -> (string * tree) list -> tree
val apply_byname : ?localized:bool -> dialog_type -> string -> (string * tree) list
-> tree
val apply_lazily : dialog_type -> string -> (string * tree) list -> tree
val concat       : application_type -> tree -> tree list -> tree
val empty        : application_type -> tree
val text         : application_type -> string -> tree
val html         : application_type -> string -> tree
val to_string    : dialog_type -> tree -> string
```

The type `template` is the abstract handle of the representation of the template definition. It is possible to get such template handles from the current UI definition, but they are read-only, and they cannot be constructed manually (i.e. without reference to the UI definition). A template should be considered as an XML tree with placeholders (dollar notation).

The type `tree` stands for an XML tree without placeholders, or better, for an XML tree where the placeholders have already been resolved. There are several ways to get such a tree: It is possible to form trivial trees (an empty tree, or a tree consisting only of one data node), or to apply a template by replacing the placeholders with subtrees, or to concatenate several trees. Note that the term "tree" has been chosen because the underlying data structure are actually XML trees; however, there are no real tree operations, and I hope that the name is not too confusing.

There are two other types playing a role here: `application_type` and `dialog_type`. The first abstracts the UI definition file, and the latter is the type of the dialog objects. Templates can only be used in the scope of a UI definition (because they may refer to other templates, and these are defined for the UI definition), and template application even needs a dialog object (because the template may refer to dialog variables).

Description:

- `Template_not_found`: This exception is raised by one of the functions accessing templates by name if there is no template with the searched name. The argument is the name of the missing template.
- `get app n`: Returns the abstract handle to the template with the name `n` defined in the application `app` (or raises Template_not_found).
- `apply dlg t p`: Instantiates the parameters of the template `t` with the passed trees `p`, and returns the resulting instantiated template tree. If a parameter is missing, but the template defines a default value, this default value will

be taken as the parameter value. If a parameter is missing, and there is no default value, an error will be reported. (This means that this function does not support dynamic contexts as alternate source for parameter values.)

This function does not perform further expansion of templates; only the passed template is instantiated.

- `apply_byname dlg n p`: This is the same as a `get` followed by an `apply`:

```
let apply_byname dlg n p = apply dlg (get dlg#application n) p
```

  The optional boolean argument `~localized` (default: `true`) selects whether the template for the current language is preferred over the generic template. (See also the chapter about *Internationalization* (→ 67).)

- `apply_lazily dlg n p`: This function forms a new tree which is guaranteed to expand the template `n` in the same manner as `apply_byname` by passing the parameters `p`; however the instantiation is not performed immediately but deferred until it is really necessary.

- `concat app s tl`: Concatenates all the trees of the passed tree list tl and separates the trees by s. This means, if tl = [t1; t2;...;tN] the resulting structure is the same as if t1 s t2 s ... s tN had been written in order. (Don't be confused that the result cannot be a tree (but perhaps a forest) - we have never said that *tree* behaves exactly as a tree structure; *tree* is only a metaphor.)

- `empty app`: Creates an empty tree.

- `text app s`: Creates a tree with one data node containing the string s. When the tree is expanded to HTML, the problematic characters are converted to the corresponding HTML entities such that the text s appears in the user interface.

- `html app s`: Creates a tree with one data node containing the string s. When the tree is expanded to HTML, the string s is left as it is without any conversion.

- `to_string dlg t`: Converts the tree t to HTML in the context of the dialog object dlg. Templates are now expanded until no more templates remain. The ui:xxx elements are expanded, too. It is possible to access the object variables of obj. Furthermore, if interactors are expanded, the necessary management infos will be entered into dlg. (However, there is currently a conceptual bug which makes it impossible to expand ui:form elements in a reasonable way.)

Template API (Perl)

Web Path: WDialog / Reference / Templates / Template API (Perl)

18 The Template API for Perl

XXX

Session management and security

Web Path: WDialog / Reference / Session management and security

19 Session Managers

The task of the session manager is to manage the link from the current request to the current dialog (i.e. the current session, which is always a dialog for WDialog). By default, the *instant session manager* is used that does not store the session externally, but simply includes it literally into the current request. The hidden form field `uiobject_session` contains the data describing the current dialog as BASE64-encoded string. This has one major advantage: You do need to set up an external data store for the sessions, instant sessions work "out of the box". In the rest of this chapter, I will try to convince you that it is better not to trust this session manager, and that the additional work necessary to instantiate the *database session manager*, *memory session manager*, or *daemon session manager* is worth-while.

19.1 The Database Session Manager

The database session manager puts the session data into a data store (that can be freely chosen), and the web request only contains a reference to the stored record. Obviously, the web requests and responses become smaller (sometimes much smaller), and your web application becomes faster (sometimes much faster), as fewer bytes need to be transferred over the network. Another advantage is not that obvious, and because of this I am talking about it: your web application becomes much more secure. I hope you understand that this is the real point behind the more complex database managers; the Internet is full of attackers today, and is highly likely that a public application is "tested" by hackers.

Of course, using a database session manager does not ensure that your application cannot be hacked. First, there are other potential weaknesses in the application, and second, sessions can still be "hijacked", although this becomes more complicated.

19.2 Using the database session manager

First, you need to instantiate the class `Wd_dialog.database_session_manager` by passing functions that allocate rows in your database, insert rows, update rows, and lookup rows. These functions must perform the real database accesses, such that the class `database_session_manager` remains generic with respect to the type of the database system. The functions are exactly described in the module interface of *Wd_dialog* (→ 187).

The second step is to pass the new session manager to the WDialog engine. In the case you use `Wd_run_cgi`, `Wd_run_fcgi`, or `Wd_run_jserv`, the entry points of the engine `run` and `create_request_handler` accept the session manager as optional argument. In the case you call `Wd_cycle.process_request` directly, this function accepts the session manager, too.

A complete example can be found in the directory `examples/list-dbm-ml` of the source distribution. This example stores the sessions in an NDBM database.

The database session manager never deletes sessions. It simply does not know when a session has been dropped by the user, and can now be removed from the database. A strategy to get rid of dropped sessions is to delete all sessions that have not been used for a long time.

19.3 The in Memory Session Manager

The memory session manager is an easy to set up, secure alternative to the database session manager, which can only be used by applications which stay memory resident between requests, and which exist within a single process image. In practical terms, it can be used by sequential or threaded FastCGI and Jserv applications.

Its implementation is very simple. It generates an opaque random session identifier, which is sent out to the client as the session id, and is also used as the index of the real session in a hash table. When the client send the id back, the real session is looked up in the hashtable. Because of this simplicity, it does not require any special functions to be written, and only requires the user to configure two paramaters, for how long sessions are valid, and how often to look for and purge from memory invalid sessions.

Besides ease of use, it has one additional advantage over the database session manager. It generates a new session for each request, and it DOES NOT throw away the old application state. This allows it to interoperate with the browser back button where that is possible, just as the instant session manager does.

19.4 Using the in Memory Session Manager

The function `new` `Wd_inmemory_session.inmemory_session_manager` Will create an instance of the memory session manager, all you need to do then is pass it to the WDialog engine.

19.5 The Daemon Session Manager

The daemon session manager is very much like the memory session manager, however it removes several annoying restrictions.

- It works with concurrent applications which use more than one process. It is therefore usable with all connectors (including CGI), and all process models.
- It maintains session state even if the application crashes, or needs to be restarted.
- It performs session garbage collection in parallel which the application, and is therefore usually faster than the memory session manager.

The daemon session manager is implemented in two parts. There is a small rpc server, which stores session data in its memory image, and there is a wdialog session manager class which talks to it. The session manager class works much like the in memory session manager, except that it takes 3 additional arguments, hostname, user, and password, which are used to contact the session daemon.

19.6 Using the Daemon Session Manager

WDialog comes with a both the daemon session manager class, and an implementation of the session daemon, which is called wdstated. The first step in using the session daemon is to configure a user name and password for each application which uses the daemon. This is done in the wdstated.conf file, which is included with the distribution. These credentials ensure that session data is only available to authorized applications. Next you should start wdstated as an unprivlidged user. You can pass the location of the config file as an argument to wdstated if it fails to find it. In your application, create an instance of `Wd_daemon_session.daemon_session_manager`, and pass it to the WDialog engine. Your application should now be storing sessions in the state daemon.

20 The security problems of the instant session manager

As already pointed out, this manager includes the session data literally into web requests. The data contain essentially marshalled O'Caml values describing the current dialog, and that means all dialog variables except the temporary ones. Because of this we assume that any low-skilled hacker can

- read the contents of the variables
- modify the contents of the variables.

Since WDialog-2.1, the marshalling format has been changed to make it secure against attacks based on malformed strings.

21 The security benefits of the database session manager, the in memory session manager, and the daemon session manager

These managers do not put the session data into the web requests, but only a reference to them, while the sessions are stored in some sort of database. The references are triples `(id,key,checksum)` where the `id` is a predictable integer identifying the session, and the `key` is a non-predictable hash value for the same purpose. Finally, the `checksum` is a hash value of the session data.

Obviously, it is no longer possible to read the dialog variables by just looking at the value of the session reference. However, a hacker can still modify a dialog variable, because all variables are writeable by default. This is allowed because an interactor might be bound to a variable, and the update of the variable implies the right to modify the variable.

The *ui:variable* (→ 170) element can declare a variable as *protected* (`protected="yes"`). In this case, the variable cannot be modified by web requests at all, even not by binding an interactor to the variable. (Of course, protected variables are only sensible if the database session manager is used, because the instant session manager allows a hacker to replace the whole session by a different one.)

The question remains whether it is possible to attack the application in a completely different way. One idea is to "hijack" the session of another user by stealing his session reference. In theory, this is possible, but it is both difficult to get such a reference and to abuse it. The reference is only included as hidden form field, and it is only transferred as part of a POST request. This means that the reference is normally not written to files (e.g. cookie file, log files, cache files), but exists only in volatile memory. So access to the disks of a computer does not help. However, browsers do (or might) contain so-called cross-site scripting bugs. These bugs sometimes allow the attackers to read form fields such as the reference without having the permission to do so by executing scripts in the browser. Unfortunately, nothing can be really done against these bugs on the server side, one can only try to minimize the damages. WDialog makes it hard to abuse a stolen session reference. First, the session contains the IP address of the browser. Often, this is already a very high hurdle to jump over for an attack. Of course, it may be possible that the attacker has access to the same web proxy as the victim, and so has the same IP address. Second, the `checksum` of the session changes frequently, but the attacker needs the checksum to take the session over.

Summarized, hijacking a session is possible, but difficult. Normally, the attacker needs at least two other security exploits, one to steal the session reference, one to have access to the IP address the session is bound to.

For a highly secure application, I would additional recommend to secure the network channels, too, i.e. to use SSL/TLS. This makes it much harder to steal the session reference as it is only transmitted in encrypted form.

Another attack one can think of is to simply guess the session reference. This is practically impossible as the `key` part of the reference is practically unpredictable, and the search space for a brute-force attack is too large (128 bits). Furthermore, you need the `checksum`, too, which is as difficult to guess as the key.

Of course, it cannot really be excluded that a working brute-force method is found. For a highly secure application, it is recommended to install attack detectors, e.g. by counting the trials for every real key stored in the database, and to drop the session if too much trials happen in short time. This should effectively protect against guessing the checksum.

Internationalization

Web Path: WDialog / Reference / Internationalization

22 Internationalization

Internationalization (I18N) is the ability to support several languages, character encodings, and local notation conventions. Currently, WDialog has some support for that, but I18N is not yet done. In particular, the following features already work:

- *UTF-8 support:* It is possible to represent all character data as UTF-8 strings. By default, character data are encoded as ISO-8859-1 (Latin-1) strings. UTF-8 does not only support many languages, but also special characters such as mathematical symbols. In order to switch from ISO-8859-1 to UTF-8, pass the `~charset:'Enc_utf8` argument to the `Wd_run_cgi.run` function (or `Wd_run_jserv.run`).

  The consequence is that all strings are now UTF-8 strings, and that the generated HTML output is UTF-8, too.

  The XML file(s) that contain the UI definition can be encoded differently, however. These files begin with `<?xml version='1.0' encoding='ENC'?>` where `ENC` is the name of a character encoding, for example ISO-8859-1, or UTF-8. Many other encodings of the ISO-8859 series are possible, too. The characters read from these files are automatically recoded to the character set demanded by the `charset` option. This means that your text editor need not to support UTF-8.

  As alternative to the direct inclusion of characters, the notation `&#n;` (where `n` is a decimal number) can be used to denote the character number n in the input text (the Unicode character set is assumed). For the default ISO-8859-1 encoding the number n must be less than 256, but for UTF-8 all valid Unicode character codes are allowed.

- *The 'language' variable:* Every dialog object may contain a language variable that selects the language of the user interface. The name of this variable must be declared in the `ui:dialog` element, for example:

```
<ui:dialog name="foo" lang-variable="current-language">
  ...
  <ui:variable name="current-language">
    <ui:string-value>en</ui:string-value>
  </ui:variable>
  ...
</ui:dialog>
```

  Here, the variable is initialized with the string "en". It is recommended to store two-letter ISO language codes as abbreviations for the meant languages into the variable. The language variable can be used as any other variable, but there are some statements that access the variable automatically. Read on for more.

- *Language conditions:* The *ui:iflang* (→ 122) element can be used to expand text only if the language variable has a certain value. An application are message catalogs like:

```
<ui:cond>
  <ui:iflang xml:lang="en">
    This is English.
  </ui:iflang>
  <ui:iflang xml:lang="de">
    Dies ist Deutsch.
```

```
  </ui:iflang>
</ui:cond>
```

Of course, the `ui:iflang` conditions are evaluated sequentially, and such "catalogs" should only be used if the number of languages is small. Nevertheless, this construct seems to be reasonable, and there is even an abbreviation to avoid notation overhead:

```
<ui:cond>
  <l:en>
    This is English.
  </l:en>
  <l:de>
    Dies ist Deutsch.
  </l:de>
</ui:cond>
```

This means exactly the same as the example above.

- *Language-sensitive templates:* Another way to select UI code depending on the language are templates that are defined differently for different languages. For instance:

```
<ui:template name="salutation" xml:lang="en" from-caller="user title">
  Good morning, $title $user!
</ui:template>

<ui:template name="salutation" xml:lang="de" from-caller="user title">
 Guten Morgen, $title $user!
</ui:template>
```

  If you invoke the template, one of the definitions is selected depending on the current state of the language variable. You can also define a "fallback" version without `xml:lang` attribute, it is used when no special version for the template is defined.

  In reality, the above templates have the full names `salutation#en` and `salutation#de`, i.e. the language code is appended to the name after the hash mark. Because of this, the language code is sometimes also called language suffix. Of course, you can call the template by its full name, and bypass the automatic selection rules, e.g. `<ui:use template="salutation#de">`.

And these features would be nice to have, but are not yet available:

- *More character sets:* Currently, only ISO-8859-1 and UTF-8 are possible. Other character sets should be supported, too.
- *Localized enumerations:* It is not possible to define enumerations differently depending on the selected language.
- *Accepted languages:* There should be a library function to determine the languages accepted/preferred by the browser.

What is really missing is a good example. I have not yet enough experience to say whether we need more features or not.

Last but not least some interesting code snippets. There are already some tricks that I should mention, as they simplify localization a lot.

Localized attributes: Imagine you want to have a button with localized labels. The string for the label is passed as XML attribute, and the question is: how to select attributes in a language-dependent way?

The straight-forward solution simply repeats the button:

```
<ui:cond>
  <l:en>
    <ui:button name="foo" label="Press here"/>
  </l:en>
  <l:de>
    <ui:button name="foo" label="Hier drücken"/>
  </l:de>
</ui:cond>
```

The drawback is that you must repeat the whole button element although only one attribute is different. Is there a better solution?

The idea is to define a template for the button, and to pass different labels. There are several possibilities:

- (1)

```
<ui:template name="button_foo" from-caller="$label">
  <ui:button name="foo" label="$label"/>
</ui:template>
...
<ui:cond>
  <l:en>
    <t:button_foo label="Press here"/>
  </l:en>
  <l:de>
    <t:button_foo label="Hier drücken"/>
  </l:de>
</ui:cond>
```

- (2)

```
<ui:template name="button_foo" from-caller="$label">
  <ui:button name="foo" label="$label"/>
</ui:template>
...
<t:button_foo>
  <p:label
     ><ui:cond
        ><l:en>Press Here</l:en><l:de>Hier drücken</l:de></ui:cond></p:label>
</t:button_foo>
```

- (3)

```
<ui:template name="button_foo" from-caller="$label">
```

```
    <ui:default param="label"
      ><ui:cond
        ><l:en>Press Here</l:en><l:de>Hier drücken</l:de><ui:cond></ui:default>
    <ui:button name="foo" label="$label"/>
  </ui:template>
  ...
  <t:button_foo/>
```

Unfortunately, these tricks work only for template invocations, so we must define `button_foo` and cannot do the same with `ui:button` directly.

Using enumerations for localization: If the string to localize is constant, an alternative for condition testing may be an enumeration. Continuing the last example, another solution is:

```
<ui:template name="button_foo" from-caller="$label">
  <ui:button name="foo" label="$label"/>
</ui:template>
...
<ui:enumeration name="label_for_foo">
  <ui:enum internal="en" external="Press here"/>
  <ui:enum internal="de" external="Hier drücken"/>
</ui:enumeration>
...
<t:button_foo>
  <p:label
    ><ui:translate type="label_for_foo"
                   internal="$[language()]"/></p:label>
</t:button_foo>
```

Here, `$[language()]` expands to the current language code, and the *ui:translate* (→ 165) element finds the corresponding external string for this code.

With enumerations it is even possible to avoid the extra template definition completely:

```
<ui:enumeration name="label_for_foo">
  <ui:enum internal="en" external="Press here"/>
  <ui:enum internal="de" external="Hier drücken"/>
</ui:enumeration>
...
<ui:button name="foo" label="$[translate(enum(label_for_foo),language())]"/>
```

The `enum` expression is a special form that returns the declaration of the named enumeration. The `translate` function does the same as ui:translate, but can be used in a *bracket expression* (→ 176).

Output encodings

Web Path: WDialog / Reference / Output encodings

23 Output encodings

This section explains how character data in the generated HTML output are encoded. The are various aspects of this theme, and it is quite easy to get totally confused. Because of this, I will first explain how character data change their encoding type during the processing steps by default, and later, how this behaviour can be modified.

24 The standard way of encoding characters

24.1 Phase 1: Parsing XML, and the internal representation

Most of the character data are read from the XML files containing the UI definition, but some strings are also dynamically added by the program (e.g. read from a database, or another background store). The XML data are parsed, and the result is an in-memory representation as XML tree. This tree can be seen as a reference point of the various recoding steps as it expresses what is meant. This becomes clearer by an example:

```
<ui:variable name="company">
  <ui:string-value>Meyer &amp; Son</ui:string-value>
</ui:variable>
```

This literal XML fragment is parsed, and represented as a tree:

```
  |
  ui:variable
      |
      +-- attribute "name" has value "company"
      +-- ui:string-value
              |
              +-- text "Meyer & Son"
```

Especially, the ampersand is now represented as ampersand, and does not need any escaping notation.

Of course, there are many more data structures than just XML trees. We have declared a variable here, and this creates a container for the variable. The important point is that the initial value of the variable can be directly taken from the XML tree, here it is "Meyer & Son". If the value is later changed (e.g. overwritten by some database record), no encoding changes are necessary. The general idea is that the internal representation never escapes characters.

24.2 Phase 2: Internal processing

In order to get HTML output, the XML tree needs to be transformed, for example, template calls must be expanded. The transformation never changes the way character data are encoded.

24.3 Phase 3: Writing the HTML output

The result of the transformation step is an HTML tree that must be written as text stream. There are essentially two major cases:

- *Element context:* This simply means that the HTML node to write occurs within an outer HTML node as sub element. HTML tags are printed with the normal tag syntax: `<tag>...</tag>`. Character data are HTML-escaped, i.e. `<` is printed as `<` etc.

  For example, this HTML tree

  ```
  |
  b
  |
  +-- text "Meyer & Son"
  ```

  is printed as

  ```
  <b>Meyer &amp; Son</b>
  ```

  It is also possible that the HTML node has attributes. These are HTML-escaped, too, e.g. if the `value` attribute has the value `"Meyer & Son"`, the whole `input` element is printed as:

  ```
  <input type="button" value="Meyer &amp; Son">
  ```

- *Attribute context:* Here, the HTML node to write occurs inside the attribute value of an outer HTML node. What? Well, this is a consequence of the template expansion algorithm. For example:

  ```
  <ui:template name="bold_meyer">
    <b>Meyer &amp; Son</b>
  </ui:template>
  ...
  <ui:template name="make_button" from-caller="value">
    <input type="button" value="$value"/>
  </ui:template>
  ...
  <t:make_button>
    <p:value><t:bold_meyer/></p:value>
  </t:make_button>
  ```

  Here, the HTML subtree `<b>Meyer & Son</b>` is finally inserted as the value of the `value` attribute! The tree looks like:

```
|
input
|
+-- attribute "type" has value "button"
+-- attribute "value":
      |
      +-- b
          |
          +-- text "Meyer & Son"
```

This case is handled in two steps. First, the HTML subtree within the attribute is linearized into a single string. Second, the string is printed as attribute, and this is the same algorithm as above, i.e. HTML meta characters are escaped.

Linearization: HTML elements are printed in tag notation. Text nodes are simply left as they are, i.e. *no HTML-escaping happens in this step*.

In the example, the result of the linearization is the string `"<b>Meyer & Son</b>"`, and this string is printed as attribute, leading to the final result

```
<input type="button" value="&lt;b&gt;Meyer &amp; Son &lt;/b&gt;">
```

I know that it is a bit surprising that this case exists, but I think it is treated in a straight-forward way.

25 How to modify the way output is encoded

25.1 Forcing the algorithm for attribute context

One drawback of the normal output encoding is that it is impossible to generate raw HTML dynamically. Imagine you have a database containing HTML pages. How do you include the pages into your generated output?

Let us assume the variable `html_page` contains the page. If you include it by

```
<ui:dynamic variable="html_page"/>
```

the ui:dynamic statement expands to a text node, and the normal encoding escapes all HTML meta characters. The result is that the browser displays the code of the page as such, but does not interpret it.

It is possible to force the algorithm that is used for attribute context. The important point is that this algorithm does not escape within text nodes. The *ui:special* (→ 156) element selects this algorithm, e.g.

```
<ui:special>
  <ui:dynamic variable="html_page"/>
</ui:special>
```

Now the HTML meta characters are left as they are, without any escaping. The browser interprets the HTML code.

25.2 Additional output encodings

The HTML pre tag preserves the formatting of the inner character block. Sometimes it would be nice to simulate the effect of pre without using it, by replacing spaces with , newlines with <br>, and by expanding tabs. The *ui:encode* (→ 106) element allows one to add an escaping algorithm to the current active set of encoders:

```
<ui:encode enc="pre">
This is the first line.
Second line.
</ui:encode>
```

The two lines are first encoded by the HTML-escaping algorithm, the default algorithm. The ui:encode element takes the result of this, and applies pre-style escaping to it. The printed HTML code is:

```
This is the first line.<br>
Second line.<br>
```

Another example: You want to generate a Javascript function that pops up an alert box on the screen:

```
<ui:template name="alert" from-caller="body">
  <script type="text/javascript">
    <ui:special>
window.alert("${body/js}");
    </ui:special>
  </script>
</ui:template>
```

The ui:special element makes that HTML-escaping is turned off. The /js notation applies the js encoding to the value of body. This encoding escapes characters that cannot occur in Javascript strings literally, e.g. the quotation mark itself.

25.3 The list of defined output encodings

The following names can be used in ui:encode, and when expanding parameters (${param/encname}) and in bracket expressions ($[expr/encname]):

- html: The HTML-escaping algorithm substitutes < for <, > for >, " for ", and & for &.
- pre: This encoding substitutes   for spaces, <br> for newline characters, and expands tabs (tab width is 8).
- para: Multiple newline characters are replaced by <p>.
- js: The characters \, ", ', <, % and control characters are escaped according to the Javascript rules such that the string can be used inside a Javascript string literal.
- jslong: A problem of js is that Javascript interpreters do not like long lines. To be on the safe side, jslong should be used instead. It puts "+\n+" sequences into the string to avoid that the resulting lines become too long.

You can define your own encodings by calling the method `add_output_encoding` of the application object.

The encodings can be referred to at a number of places:

- *ui:encode:* The element *ui:encode* (→ 106) applies the encoding to what is printed for the subelements.
- *Parameters:* The syntax `${param/enc}` applies the encoding `enc` to the value of the template parameter `param`.
- *Bracket expressions:* The syntax `$[expr/enc]` applies the encoding `enc` to the result of the *bracket expression* (→ 176) `expr`.

Processing instructions

Web Path: WDialog / Reference / Processing instructions

26 Processing instructions

Processing instructions can be used to modify the way the UI definition is interpreted. Currently, all processing instructions must be directly contained in the *ui:application* (→ 86) element.

- `<?wd-debug-mode?>`: In debug mode, the generated HTML code contains a number of comments showing the current state of the dialog. Although these comments are not rendered by browsers, you can view them by selecting the "View source code" function of the browser.

  The state is shown as a long XML expression, even including some inner comments to improve the readability. There are two styles to encode the XML expression:

  - `<?wd-debug-mode partially-encoded?>`: This is style is intended for browsers that do not decode text occuring in comments (e.g. Internet Explorer, Mozilla). Only "-->" is converted to "==>" to avoid that the outer comment is closed prematurely. This style is the default.
  - `<?wd-debug-mode fully-encoded?>`: This is style is intended for browsers that do decode text within comments (e.g. Netscape 4). The characters <, >, and & are converted to their escaped forms <, >, and &, respectively.

- `<?wd-prototype-mode?>`: In prototype mode, missing definitions for dialog classes are automatically added to the universe of dialog classes. These definitions are "empty", i.e. nothing is done in `prepare_page`, and nothing happens in `handle`. The prototype mode is useful to design dialogs without changing the program.

- `<?wd-onstartup-call-handle?>`: This mode forces that the `handle` method is called even if no event has been recognized. Normally, the `handle` method is not called in such cases.

  When the application is started, i.e. the user has typed in the URL, there is no event, and because of this normally the first action is to prepare the first page to display. This processing instruction changes the behaviour, and the first invoked method is `handle`, followed by `prepare_page`.

  Another possible situation is that Javascript statements submit the current form, and the WDialog routines do not detect that one of the defined events (from ui:button elements etc.) have happened. Normally, the invocation of `handle` is omitted unless this processing instruction forces its execution.

The UI language

Web Path: WDialog / Reference / The UI language

27 The UI Language

The UI language is defined in terms of XML-1.0. The subsections explain every element in detail; together they cover the complete DTD of the language.

27.1 The hierarchy of the UI elements

The following picture shows the hierarchy. *The picture is active,* you can click at the elements to go immediately to their descriptions!

27.2 Other syntactic objects

For many reasons, the above `ui:name` elements are not the best choice for every kind of notation. Because of this, a number of further syntactic objects have been defined:

- *t:*, q:*, and p:** (→ 173) (Elements like `<t:name>`, `<q:name>`, and `<p:name>`)
- *l:** (→ 174) (Elements like `<l:name>`)
- *$param* (→ 175) (Template parameters)
- *$[expr]* (→ 176) (Bracket expressions)
- *Dot notation (v1.v2)* (→ 180)

27.3 The semantic levels, and the incompleteness of the formal declarations

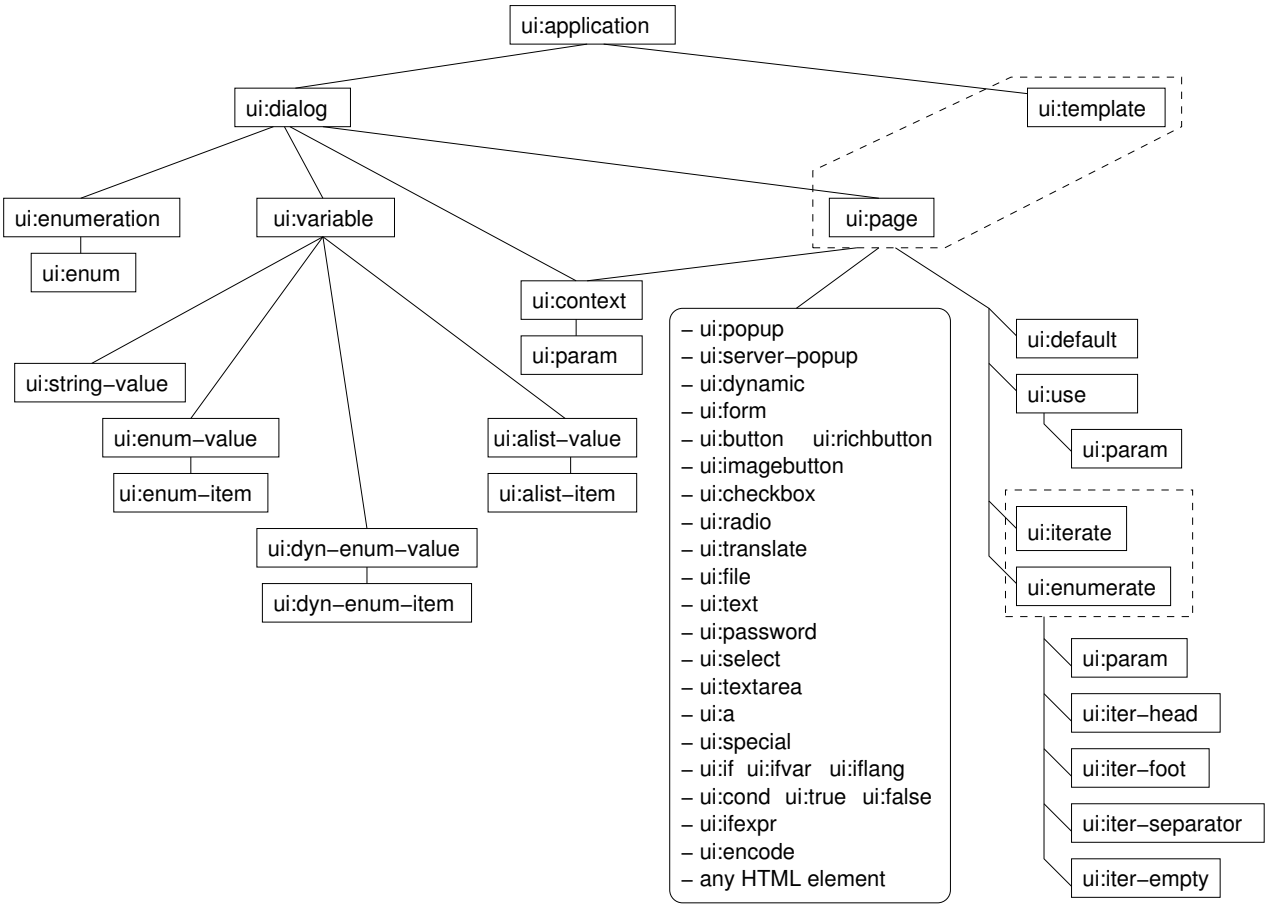
The elements shown in picture 5 belong to one of three semantic levels. The topmost element `ui:application` and its children are the elements that determine the *dialog structure*, i.e. which dialogs exist, and what are their static properties. The elements inside `ui:page` are either *generative elements*, or *control structures*. The former generate output directly, while the latter control the expansion process (e.g. "if" conditions).

I have tried to give a formal XML declaration for every element. Unfortunately, the XML DTDs are not powerful enough to specify the relations between the elements fully. I found a number of problems, and they are often related to difficulties how to describe the relations between the semantic levels.

Many control structures are declared with a content model of `ANY`, i.e. the DTD does not restrict the type of the children elements. There are actually two reasons for this. First, the replacement for `ANY` would be an enumeration of all control structures, and all generative elements. But the latter are not completely known for the scope of the UI language, as all HTML elements are generative, too, and it is intentionally avoided to include HTML as sublanguage into the UI language.

The second reason is that the relations between control structures and generative elements cannot be expressed by the DTD. The control structures are evaluated at expansion time, and change the order of the generative elements dynamically. For example, is it allowed that the `TD` HTML element is a child of `ui:template`, a control structure? This depends on where the template is called. If this happens inside `TR`, the usage of the template will be legal because `TD` must finally

Picture 5: The element hierarchy



occur inside `TR` according to the HTML language. However, this condition cannot be expressed in a DTD. You need the power of a type system to check the validity of element relations across semantic levels.

Note that even the attribute declarations are incomplete. Often, the elements of the UI language are only variants of the corresponding fundamental HTML elements. For example, the `ui:button` element generates an `INPUT` HTML element with `TYPE=SUBMIT`. Of course, it is allowed to add HTML attributes such as `CLASS` or `ONCLICK` to `ui:button` as they can be simply copied to the generated `INPUT` element. In the declarations, these HTML attributes are omitted to avoid dependencies on certain HTML versions.

In the following sections, the semantic level of the element is given in addition to the formal declaration. This helps a lot to find out where (and how) an element can be used: Both control structures and generative elements can only be used in page context, i.e. when the creation of an output page is described. The control structures determine the algorithm how the generative elements are arranged. Finally, the page consists only of generative elements, and these must have a sound structure that is correct with respect to the HTML definition.

ui:a

Web Path: WDialog / Reference / The UI language / ui:a

28 The element ui:a

This element displays a hyperlink referring to a position or function within the application. The generated HTML code consists of an `A` element with a "javascript:..." `href`, and some hidden fields which are set from the Javascript code bound to the hyperlink. When the hyperlink is clicked, these fields are filled, and the form is submitted. The system recognizes that the hyperlink has been clicked by checking for these special fields.

Note: This element works only for browsers which are capable of executing Javascript code!

When the user clicks on the hyperlink, a *hyperlink* event is generated (unless the `index` attribute is specified; see below); the `handle` callback method of the dialog object can check whether the current event is the event associated with this hyperlink, and the method can execute code depending on the result of this check. For a description of possible events see *Events* (→ 41). The following example illustrates hyperlink events:

```
<ui:dialog name="sample" start-page="p1">
  <ui:page name="p1">
    <html>
      <body>
        <h1>Hyperlink test</h1>
        This is a <ui:a name="b">hyperlink</ui:a>.
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

Here, the hyperlink event has the name "b". In order to check whether this event occured in the `handle` method, the following piece of code is recommended. Note that hyperlink events are actually handled in exactly the same way as button events. O'Caml:

```
method handle() =
  match self # event with
    Button "b" ->             (* yes, it's also Button for hyperlinks! *)
      ... (* Do whatever you want to do *)
  | ... (* other cases *)
```

- Perl:

```
sub handle {
    my ($self) = @_;
    my ($e, $name) = $self->event;
    if ($e eq 'BUTTON' && $name eq 'b') {   # "BUTTON" works also for hyperlinks
```

```
        ... # Do whatever you want to do
    } elsif ... # other cases
    ;
    return undef;
}
```

If the `ui:a` element sets the `index` attribute, the hyperlink is identified by the pair `(name,index)`. When the user clicks on such an indexed link, an *indexed hyperlink* event is generated. The index value can be used to distinguish between several instances of hyperlinks of the same type. For instance, a book store may offer the customer several books:

```
<ui:dialog name="sample" start-page="view_records">
  <ui:page name="view_records">
    <html>
      <body>
        <h1>View books</h1>
        <table>
          <tr>
            <th>Author</th>
            <th>Title</th>
            <th>Action</th>
          </tr>
          <tr>
            <td>Damon Runryon</td>
            <td>Guys and Dolls</td>
            <td><ui:a name="view" index="4523">View Details</ui:a></td>
          </tr>
          <tr>
            <td>William S. Burroughs</td>
            <td>Naked Lunch</td>
            <td><ui:a name="view" index="8612">View Details</ui:a></td>
          </tr>
        </table>
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

Here, the index value is the database ID of the record. The typical code to check for such a hyperlink in the `handle` callback is - O'Caml:

```
method handle() =
  match self # event with
    Indexed_button("view", index) ->
      ... (* Do whatever you want to do *)
  | ... (* other cases *)
```

- Perl:

```
sub handle {
    my ($self) = @_;
    my ($e, $name, $index) = $self->event;
    if ($e eq 'INDEXED_BUTTON' && $name eq 'view') {
        ... # Do whatever you want to do
    } elsif ... # other cases
    ;
    return undef;
}
```

Note that the transport mechanism for the strings specified for name and/or index is 8 bit clean (at least if `cgi="auto"`). This means that the name and index strings may be composed of all characters of the character set.

28.1 Declaration

Level: Generative

```
<!ELEMENT ui:a ANY>
<!ATTLIST ui:a
          name  NMTOKEN     #REQUIRED
          index CDATA       #IMPLIED
          goto  NMTOKEN     #IMPLIED
          cgi   (auto|keep) "auto"
>
```

Additionally, `ui:a` must only occur inside `ui:form`. `ui:a` must not contain another `ui:a` element.

28.2 Attributes

The following attributes have a special meaning:

- `name`: Specifies the name of the hyperlink.
- `index`: Specifies the index value of the link. If this attribute is present, the hyperlink becomes an indexed link; otherwise the hyperlink is a plain link.
- `goto`: Specifies which page is the next page if the hyperlink is clicked. The variable containing the next page is initialized with the name specified here before the `handle` method is invoked. This means that the action of the link is to go to this page, unless the action is overridden in the `handle` method.
- `cgi`: The value "auto" means that the name of the CGI variable associated with the hyperlink is selected automatically. This works perfectly unless you want to refer to this variable from Javascript or from some other manually written event decoder. The value "keep" causes that the name of the CGI variable is predictable: it is `anchor_` concatenated with the name of the link. However, it is not allowed to specify "keep" if there is also an index value. Furthermore, the hyperlink name should only contain alphanumeric characters, because not all punctuation characters can be safely transported over the CGI protocol.

If there are any other attributes, these are added to the generated `A` HTML element. This means that especially the `onclick`, `onmouseover`, and `onmouseout` attributes may be specified. It is possible to set also the `target` value, but I do not recommend this (it may have strange effects).

28.3 Sub elements

The contents of `ui:a` are rendered as hyperlink zone.

28.4 Generated HTML code

The `ui:a` element generates HTML code which roughly looks as follows:

```
<a href="javascript:...">...</a>
<input type="hidden" name="..." value="...">
```

ui:alist-value and ui:alist-item

Web Path: WDialog / Reference / The UI language / ui:alist-value and ui:alist-item

29 The elements ui:alist-value and ui:alist-item

The element `ui:alist-value` represents a literal for associative lists that can be used to set the initial value of a *ui:variable* (→ 170). The element `ui:alist-item` represents one association pair.

29.1 Declaration

Level: Dialog structure

```
<!ELEMENT ui:alist-value (ui:alist-item)* >

<!ELEMENT ui:alist-item %value-literal; >

<!ATTLIST ui:alist-item
          index  CDATA #REQUIRED>
```

For the definition of `%value-literal;` see *ui:variable* (→ 170).

Restriction: All items must contain literals of the same type. Furthermore, it is (currently) not allowed that an item contains another `alist-value`.

29.2 Attributes

- `index`: The index of the item.

29.3 Sub elements

The `ui:alist-item` contains the literal that corresponds to the `index`

29.4 Example

```
<ui:enumeration name="fruit">
  <ui:enum internal="apple"  external="I like apples"/>
  <ui:enum internal="orange" external="I like oranges"/>
  <ui:enum internal="banana" external="I like bananas"/>
</ui:enumeration>

<ui:variable name="preference" type="fruit" associative="yes">
  <ui:alist-value>
```

```
    <ui:alist-item index="John">
      <!-- John's preferred fruit: -->
      <ui:enum-value>
        <ui:enum-item internal="orange"/>
        <ui:enum-item internal="banana"/>
      </ui:enum-value>
    </ui:alist-item>
    <ui:alist-item index="Mary">
      <!-- Mary's preferred fruit: -->
      <ui:enum-value>
        <ui:enum-item internal="apple"/>
        <ui:enum-item internal="banana"/>
      </ui:enum-value>
    </ui:alist-item>
  </ui:alist-value>
</ui:variable>
```

ui:application

Web Path: WDialog / Reference / The UI language / ui:application

30 The element ui:application

This is the top-level element of an application. It contains all *ui:dialog* (→ 98) and *ui:template* (→ 158) elements.

30.1 Declaration

Level: Dialog structure

```
<!ELEMENT ui:application ( ui:dialog | ui:template )+ >

<!ATTLIST ui:application
          start-dialog NMTOKEN #REQUIRED
>
```

30.2 Attributes

- *start-dialog:* This required attribute determines the start dialog of the application, i.e. the dialog to create first when the user starts the application (enters its URL into the browser).

30.3 Example

```
<ui:application start-dialog="main">
  <ui:dialog name="main" start-page="portal">
    <ui:page name="portal">
      <html>
        <body>
          <h1>This is the first page of the application!</h1>
        </body>
      </html>
    </ui:page>
  </ui:dialog>
</ui:application>
```

ui:button

Web Path: WDialog / Reference / The UI language / ui:button

31 The element ui:button

This element displays a button. The generated HTML code consists of an `INPUT` element with `TYPE=SUBMIT`, whose `name` attribute is set to a special identifier which is recognized by the system when the form is submitted.

When the user clicks on the button, a `Button` event is generated (unless the `index` attribute is specified; see below); the `handle` callback method of the dialog object can check whether the current event is the event associated with this button, and the method can execute code depending on the result of this check. For a description of possible events see *Events* (→ 41). The following example illustrates button events:

```
<ui:dialog name="sample" start-page="p1">
  <ui:page name="p1">
    <html>
      <body>
        <h1>Button test</h1>
        This is a <ui:button name="b" label="Button"/>
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

Here, the button event has the name "b". In order to check in the `handle` method whether this event occured, the following piece of code is recommended. O'Caml:

```
method handle =
  match self # event with
    Button "b" ->
      ... (* Do whatever you want to do *)
  | ... (* other cases *)
```

- Perl:

```
sub handle {
    my ($self) = @_;
    my ($e, $name) = $self->event;
    if ($e eq 'BUTTON' && $name eq 'b') {
        ... # Do whatever you want to do
    } elsif ... # other cases
    ;
    return undef;
```

```
}
```

If the `ui:button` element has the `index` attribute, the button is identified by the pair `(name,index)`. When the user clicks on such an indexed button, an `Indexed_button` event is generated. The index value can be used to distinguish between several instances of buttons of the same type. For example, a book store may offer the customer several books:

```
<ui:dialog name="sample" start-page="view_records">
  <ui:page name="view_records">
    <html>
      <body>
        <h1>View books</h1>
        <table>
          <tr>
            <th>Author</th>
            <th>Title</th>
            <th>Action</th>
          </tr>
          <tr>
            <td>Damon Runryon</td>
            <td>Guys and Dolls</td>
            <td><ui:button name="view" label="View Details" index="4523"/></td>
          </tr>
          <tr>
            <td>William S. Burroughs</td>
            <td>Naked Lunch</td>
            <td><ui:button name="view" label="View Details" index="8612"/></td>
          </tr>
        </table>
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

Here, the index value is the database ID of the record. The typical code to check for such a button in the `handle` callback is - O'Caml:

```
method handle =
  match self # event with
    Indexed_button("view", index) ->
      ... (* Do whatever you want to do *)
  | ... (* other cases *)
```

- Perl:

```
sub handle {
    my ($self) = @_;
    my ($e, $name, $index) = $self->event;
```

```
    if ($e eq 'INDEXED_BUTTON' && $name eq 'view') {
        ... # Do whatever you want to do
    } elsif ... # other cases
    ;
    return undef;
}
```

Note that the transport mechanism for the strings specified for name and/or index is 8 bit clean (at least if cgi="auto"). This means that the name and index strings may be composed of all characters of the character set.

31.1 Declaration

Level: Generative element

```
<!ELEMENT ui:button EMPTY>
<!ATTLIST ui:button
          name  NMTOKEN     #REQUIRED
          index CDATA       #IMPLIED
          label CDATA       #IMPLIED
          goto  NMTOKEN     #IMPLIED
          cgi   (auto|keep) "auto"
>
```

Additionally, `ui:button` must occur inside `ui:form`.

31.2 Attributes

The following attributes have a special meaning:

- `name`: Specifies the name of the button.
- `index`: Specifies the index value of the button. If this attribute is present, the button becomes an indexed button; otherwise the button is a plain button.
- `label`: Specifies the text appearing on the button.
- `goto`: Specifies which page is the next page if the button is pressed. The variable containing the next page is initialized with the name specified here before the `handle` method is invoked. This means that the action of the button is to go to this page, unless the action is overridden in the `handle` method.
- `cgi`: The value "auto" means that the name of the CGI variable associated with the button is selected automatically. This works perfectly unless you want to refer to this variable from Javascript or from some other manually written event decoder. The value "keep" causes that the name of the CGI variable is predictable: it is `button_` concatenated with the name of the button. However, it is not allowed to specify "keep" if there is also an index value. Furthermore, the button name should only contain alphanumeric characters, because not all punctuation characters can be safely transported over the CGI protocol.

If there are any other attributes, these are added to the generated `INPUT` HTML element. This means that especially the `onclick` attribute may be specified.

31.3 Sub elements

`ui:button` elements do not have sub elements.

31.4 Generated HTML code

The `ui:button` element generates HTML code which roughly looks as follows:

```
<input type="SUBMIT" name="..." value="...">
```

ui:checkbox

Web Path: WDialog / Reference / The UI language / ui:checkbox

32 The element ui:checkbox

This element displays a checkbox. The generated HTML code consists of an `INPUT` element with `TYPE=CHECKBOX`, whose name attribute is set to a special identifier which is recognized by the system when the form is submitted.

The checkbox must be tied to an enumerator variable (either a declared one, or a dynamic enumerator); the name of the variable must be specified in the `variable` attribute. Furthermore, there must be a `value` attribute determining which value is visualized by the checkbox. The rule is as follows: The checkbox is in the state "checked" iff the specified value occurs in the set of values currently stored in the specified variable.

The checkbox widget will be initialized to the state given by this rule when the current page is displayed. Furthermore, any state change of the widget caused by user interaction will be propagated back to the enumerator variable when the current page is submitted. This means that if the user checks the box the specified value will be added to the specified enumerator, and that conversely if the user releases the box the specified value will be deleted from the specified enumerator variable. However, other members of the enumerator variable than the specified one remain unchanged.

Of course, the specified value is an internal value with respect to the difference between internal and external values.

In the following example, the customer can select which kind of fruit he orders. The variable `customer_wish` is initialized with the set {"apple"}, and because of this, the page appears initially with a checked "apple" box and unchecked "banana" and "ananas" boxes. The checkboxes simply visualize the current state of the variable. When the customer has selected his items and presses the "OK" button, the variable `customer_wish` is automatically updated from the state of the input widgets, and reflects again the current state of the boxes. From the `handle` callback method, one can read the variable `customer_wish` and interpret the contents.

```
<ui:dialog name="sample" start-page="sample_page">
  <ui:enumeration name="fruit">
    <ui:enum internal="apple"  external="Apple"/>
    <ui:enum internal="banana" external="Banana"/>
    <ui:enum internal="ananas" external="Ananas"/>
  </ui:enumeration>

  <ui:variable name="customer_wish" type="fruit">
    <ui:enum-value>
      <ui:enum-item internal="apple"/>
    </ui:enum-value>
  </ui:variable>

  <ui:page name="sample_page">
    <html>
      <body>
        Please select what you want:
        <ul>
          <li><ui:checkbox variable="customer_wish" value="apple"/>
              Apples</li>
          <li><ui:checkbox variable="customer_wish" value="banana"/>
```

```
                Bananas</li>
            <li><ui:checkbox variable="customer_wish" value="ananas"/>
                Ananas</li>
          </ul>
          <ui:button name="ok" label="OK"/>
        </body>
      </html>
    </ui:page>
  </ui:dialog>
```

32.1 Declaration

Level: Generative

```
<!ELEMENT ui:checkbox EMPTY>
<!ATTLIST ui:checkbox
          variable NMTOKEN     #REQUIRED
          index    CDATA       #IMPLIED
          value    NMTOKEN     #REQUIRED
          cgi     (auto|keep)  "auto"
>
```

Additonally, `ui:checkbox` must only occur inside `ui:form`.

32.2 Attributes

The following attributes have a special meaning:

- `variable`: Specifies the variable of the current dialog object to which the checkbox is tied. Unless the `index` attribute is present, the variable must be a declared enumerator or a dynamic enumerator. If there is an `index` attribute, the variable must be an associative list of either declared or dynamic enumerators.
- `index`: Specifies the index value of the element of the associative variable to which the checkbox is tied.
- `value`: Specifies the internal value whose presence in the enumerator is represented by the checkbox.
- `cgi`: The value "auto" means that the name of the CGI variable associated with the checkbox is selected automatically. This works perfectly unless you want to refer to this variable from Javascript or from some other manually written event decoder. The value "keep" causes that the name of the CGI variable is predictable: it is `var_` concatenated with the name of the variable. However, it is not allowed to specify "keep" if there is also an index value. Furthermore, the variable name should only contain alphanumeric characters, because not all punctuation characters can be safely transported over the CGI protocol.

If there are any other attributes, these are added to the generated `INPUT` HTML element. This means that especially the `onclick` attribute may be specified.

32.3 Sub elements

`ui:checkbox` does not have sub elements.

32.4 Tips

Often, it is desired to iterate over all defined values of an enumerator, and to output a checkbox for every item. The following code demonstrates how `ui:checkbox` works in conjunction with *ui:enumerate* (→ 108); it is a another version of the fruit example:

```
<ui:template name="list_item" from-caller="int ext">
  <li>
    <ui:checkbox variable="customer_wish" value="$int"/> $ext
  </li>
</ui:template>

<ui:dialog name="sample" start-page="sample_page">
  <ui:enumeration name="fruit">
    <ui:enum internal="apple"  external="Apple"/>
    <ui:enum internal="banana" external="Banana"/>
    <ui:enum internal="ananas" external="Ananas"/>
  </ui:enumeration>

  <ui:variable name="customer_wish" type="fruit">
    <ui:enum-value>
      <ui:enum-item internal="apple"/>
    </ui:enum-value>
  </ui:variable>

  <ui:page name="sample_page">
    <html>
      <body>
        Please select what you want:
        <ul>
          <ui:enumerate template="list_item"
                        type="fruit"/>
        </ul>
        <ui:button name="ok" label="OK"/>
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

ui:cond

Web Path: WDialog / Reference / The UI language / ui:cond

33 The element ui:cond

This element contains several conditional branches. The first branch whose condition code evaluates to `true` is expanded. If no branch is `true`, nothing will be expanded.

33.1 Declaration

Level: Control structure

```
<!ELEMENT ui:cond ( (ui:if | ui:ifexpr | ui:ifvar | ui:iflang | ui:true | ui:false)+
)>
```

33.2 Sub elements

The following sub elements are possible. All are conditional elements setting the condition code.

- *ui:if* (→ 119)
- *ui:ifexpr* (→ 121)
- *ui:ifvar* (→ 124)
- *ui:iflang* (→ 122)
- *ui:true* (→ 167)
- *ui:false* (→ 113)

33.3 Example

```
<ui:cond>
  <ui:ifvar variable="v" value="1">
    ...branch 1...
  </ui:ifvar>
  <ui:ifvar variable="v" value="2">
    ...branch 2...
  </ui:ifvar>
  <ui:true>
    ...branch 3...
  </ui:true>
</ui:cond>
```

If the variable `v` has the value 1, the first branch will be expanded. If it has the value 2, the second branch will be expanded. In all other cases, the third branch will be expanded.

33.4 Hints

There is a certain order in which the various types of expansions are performed. First, template parameters and bracket expressions are evaluated and replaced by the resulting values. After that, the conditions are tested, and the right conditional branches are selected. This particular order may lead to problems when the test conditions are used to ensure preconditions of expressions, like in:

```
<ui:if value1="$x" value2="0" op="int-gt">
  $[div(100,$x)]   <!-- Runtime Error! -->
</ui:if>
```

Here, the division is carried out before the test whether `$x` is positive, and because of this a runtime error can happen. The solution is to put the bracket expression into a template, as templates are expanded only on demand:

```
<ui:template name="divide" from-caller="x">
  $[div(100,$x)]
</ui:template>

...

<ui:if value1="$x" value2="0" op="int-gt">
  <t:divide x="$x"/>
</ui:if>
```

ui:context

Web Path: WDialog / Reference / The UI language / ui:context

34 The element ui:context

This elements extends the set of current context parameters while the body of `ui:context` is expanded. Context parameters are one way to pass values to *templates* (→ 49).

If a parameter is added to the context by `ui:context`, but it happens that a parameter with the same name is already member of the context, the existing member is suspended for the time the body of `ui:context` is expanded, and the new parameter becomes part of the context instead.

34.1 Declaration

Level: Control structure

```
<!ELEMENT ui:context ANY >
```

The subelements of `ui:context` must match the informal rule `( ui:param*, %context-body;* )` where `%context-body;` stands symbolically for the sub elements of the body. Note that whitespace between the `%context-body;` elements counts, but at the other places it does not count.

34.2 Sub elements

The *ui:param* (→ 140) subelements must be placed at the beginning of the inner nodes. For every `ui:param` a new parameter is added to the context.

The elements following `ui:param` are the *body* of `ui:context`.

ui:default

Web Path: WDialog / Reference / The UI language / ui:default

35 The element ui:default

The element can be used at the beginning of *ui:page* (→ 135) and *ui:template* (→ 158) to define default values for template parameters that are not passed to the template.

35.1 Declaration

Level: Control structure

```
<!ELEMENT ui:default ANY>

<!ATTLIST ui:default
          name     NMTOKEN    #REQUIRED
>
```

35.2 Attributes

- `name`: The name of the parameter

35.3 Sub elements

All page body elements may occur in `ui:default`.

35.4 Known Bug

Template parameters inside `ui:default` are currently not expanded. This is likely to change in the future.

Example that does not work yet:

```
<ui:template name="foo" from-caller="a b">
  <ui:default name="a">44</ui:default>
  <ui:default name="b">$a + $a = 88</ui:default>
  ...
</ui:template>
```

ui:dialog

Web Path: WDialog / Reference / The UI language / ui:dialog

36 The element ui:dialog

An `ui:dialog` element describes the object behind a series of interactions (*dialog* (→ 22)) that base on shared state variables. The object has the following properties:

- The object consists of a number of instance variables. These variables must be declared with the *ui:variable* (→ 170) directive. Every variable has a type and a value. It is possible to set a default value to which the variable is initialized when the object is being created.
- The object can be visualized as one of the defined pages (see *ui:page* (→ 135)). The elements describing the visualization can access the instance variables of the object, either reading them only (such as in *ui:dynamic* (→ 102)), or associating them with interactor elements which can be modified by the user of the application (such as in *ui:text* (→ 160)).
- The object remembers which page has been displayed last; this page is called the current page.
- When the user clicks on a hyperlink (see *ui:a* (→ 80)) or on a button (see *ui:button* (→ 87)), an event is triggered and sent to the object.

The programmer can associate an O'Caml or Perl class to the object, and this class is then treated as extension to the default behaviour of the object. You can find detailed descriptions of the dialog properties in the chapter about *dialogs* (→ 22).

36.1 Declaration

Level: Dialog structure

```
<!ELEMENT ui:dialog ( ( ui:enumeration |
                        ui:variable |
                        ui:context |
                        ui:page )* ) >

<!ATTLIST ui:dialog
          name          NMTOKEN #REQUIRED
          start-page    NMTOKEN #REQUIRED
          lang-variable NMTOKEN #IMPLIED
>
```

Restrictions: There must at most only one `ui:context` sub element. There must be at least the `ui:page` mentioned by `start-page`.

36.2 Attributes

- `name`: The name of the object which must be unique among all objects of the application.
- `start-page`: The name of the page to which the current page property is initialized when the object is created.
- `lang-variable`: The name of a string variable that contains the selected language. It is required that this variable is declared by `ui:variable`. For more information, see the chapter *Internationalization* (→ 67).

36.3 Sub elements

The following sub elements may be contained in `ui:dialog` in arbitrary order:

- *ui:enumeration* (→ 111): Declarations of enumerator types
- *ui:variable* (→ 170): Declarations of instance variables
- *ui:context* (→ 96): Optionally, a default binding for dynamic template parameters
- *ui:page* (→ 135): One or more pages describing possible visualizations of the dialog. At least the page defined by `start-page` must exist.

36.4 Example

```
<ui:dialog name="name_dialog" start-page="change_name">
  <ui:variable name="first_name" type="string"/>
  <ui:variable name="last_name" type="string"/>

  <ui:page name="change_name">
    <html>
      <body>
        <h1>Please enter your name here:</h1>

        <table>
          <tr>
            <td>First name:</td>
            <td><ui:text variable="first_name"/></td>
          </tr>
          <tr>
            <td>Last name:</td>
            <td><ui:text variable="last_name"/></td>
          </tr>
        </table>

        <p><ui:button name="name_changed" label="Done" goto="show_name"/></p>
      </body>
    </html>
  </ui:page>

  <ui:page name="show_name">
    <html>
```

```
      <body>
        <h1>Welcome!</h1>

        Welcome, <ui:dynamic variable="first_name"/>
<ui:dynamic variable="last_name"/>!
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

ui:dyn-enum-value and ui:dyn-enum-item

Web Path: WDialog / Reference / The UI language / ui:dyn-enum-value and ui:dyn-enum-item

37 The elements ui:dyn-enum-value and ui:dyn-enum-item

The element `ui:dyn-enum-value` represents a literal for dynamic enumerators that can be used to set the initial value of a *ui:variable* (→ 170). The element `ui:dyn-enum-item` represents one enumerated item.

37.1 Declaration

Level: Dialog structure

```
<!ELEMENT ui:dyn-enum-value (ui:dyn-enum-item)* >

<!ELEMENT ui:dyn-enum-item EMPTY>

<!ATTLIST ui:dyn-enum-item
          internal  NMTOKEN  #REQUIRED
          external  CDATA    #IMPLIED >
```

37.2 Attributes

- `internal`: The internal value identifying the enumerated item
- `external`: The corresponding external value for display purposes. If omitted, the internal value is also used as external value.

37.3 Example

```
<ui:variable name="preference" type="dynamic-enumerator">
  <ui:dyn-enum-value>
    <ui:dyn-enum-item internal="orange" external="I like oranges"/>
    <ui:dyn-enum-item internal="banana" external="I like bananas"/>
  </ui:dyn-enum-value>
</ui:variable>
```

ui:dynamic

Web Path: WDialog / Reference / The UI language / ui:dynamic

38 The element ui:dynamic

The element `ui:dynamic` is replaced by the current value of the variable it refers to. The replacement algorithm can quote the value according to several quoting styles; see below.

One task of `ui:dynamic` is to show the values of variables as constant, immutable texts. For example, the following dialog displays the values of the current data fields in the page `show_record`:

```
<ui:dialog name="sample" start-page="show_record">
  <ui:variable name="title"/>
  <ui:variable name="author"/>
  <ui:variable name="isbn"/>
  <ui:variable name="price"/>

  <ui:page name="show_record">
    <html>
      <body>
        <h1>The selected record</h1>
        <dl>
          <dt>Title:</dt>
          <dd><ui:dynamic variable="title"/></dd>
          <dt>Author:</dt>
          <dd><ui:dynamic variable="author"/></dd>
          <dt>ISBN number:</dt>
          <dd><ui:dynamic variable="isbn"/></dd>
          <dt>price:</dt>
          <dd><ui:dynamic variable="price"/></dd>
        </dl>
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

Of course, one has to load the values into the displayed variables. This can be done in the `prepare_page` method of the dialog object which is called just before the current page is printed as HTML document.

Another way to apply `ui:dynamic` is to insert an already generated HTML fragment at the current location into the output stream. In this case, the attribute `special` must be set to indicate that the variable contains already HTML and that no further quoting is required. A tiny example:

```
<ui:dialog name="sample" start-page="p">
  <ui:variable name="x">
    <ui:string-value>This is &lt;b&gt;bold&lt;/b&gt; text!</ui:string-value>
```

```
  </ui:variable>

  <ui:page name="p">
    <html>
      <body>
        The value of x:
        <ui:dynamic variable="x" special="yes"/>
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

Here, the variable `x` is initialized with the value "This is bold text!", and this value is verbatim included into the generated HTML code.

38.1 Declaration

Level: Generative element

```
<!ELEMENT ui:dynamic EMPTY>
<!ATTLIST ui:dynamic
          variable  NMTOKEN  #REQUIRED
          index     CDATA    #IMPLIED
          special   (yes|no) "no"
          enc       NMTOKENS ""
>
```

38.2 Attributes

- `variable`: Specifies the name of the variable to display. This must be a string variable.
- `index`: Specifies the selected index if the variable is an associative string variable. This attribute is required if the variable is associative.
- `special`: This attribute determines if the usual output encoding is applied (`special="no"`) or not (`special="yes"`). Normally, the characters <, >, &, and " in the generated HTML text are replaced by their corresponding entities, so they are displayed "as they are meant". The attribute `special="yes"` turns the output encoding off (like the element *ui:special* (→ 156) for bigger sections of code).
- `enc`: Defines a list of additional output encodings that are applied *before* the normal HTML output encoding. See *Output encodings* (→ 71) for a list of encodings. It is recommended to define `enc` only in conjuction with `special="yes"` to get full control over the order in which the encodings are applied.

38.3 Special case: Using ui:dynamic in parameter values

Or: How to set attributes dynamically

When ui:dynamic occurs in values of template parameters (i.e. within *ui:param* (→ 140)), the following behaviour can be expected. If the parameter is referred to from character data context, the ui:dynamic element is just passed through to the template expansion text as any other element. For example, if the template definition of t1 is

```
<ui:template name="t1" from-caller="x">
  The text is: <b>$x</b></ui:template>
```

and the template is called as in

```
<ui:use template="t1">
  <ui:param name="x"><ui:dynamic variable="v"/></ui:param>
</ui:use>
```

the expanded template will be:

```
The text is: <b><ui:dynamic variable="v"/></b>
```

This is nothing special. However, it is also possible to use parameters in attribute context. In this case, the ui:dynamic element will be immediately replaced by the (optionally quoted) contents of the called variable; this is different from most other ui: elements which are just ignored in attribute context. If in our example t2 is defined as

```
<ui:template name="t2" from-caller="x">
  <ui:button name="b_$x" label="Label of: $x"/></ui:template>
```

and called in the same way as t1, the expansion text will be computed as follows:

```
<ui:button name="b_<v>" label="Label of: <v>"/>
```

where <v> is replaced by the current contents of the variable v.

38.4 The bracket notation

Or: How to set attributes dynamically in a better way

In pages and templates, the notation `$[name]` can be used to insert the current value of the named instance variable, just like `ui:dynamic` does. This notation can also be used inside atttibutes, so the last example can be better written as

```
<ui:button name="b_$[v]" label="Label of: $[v]"/>
```

To get the effect of `special="yes"` you have to put the bracket into a *ui:special* (→ 156) element:

```
<ui:special>$[v]</ui:special>
```

To get the effect of `enc`, just add the encodings after a slash, and separate them by slashes:

```
<ui:special>$[v/html/js]</ui:special>
```

In recent versions of WDialog, the bracket notation has been generalized, and it is now allowed to write more complex expressions inside the brackets. See the chapter about *$[expr]* (→ 176).

ui:encode

Web Path: WDialog / Reference / The UI language / ui:encode

39 The element ui:encode

This element changes the output encoding for the time its sub elements are expanded. By default, WDialog uses the `html` output encoding, as if the whole page were embraced by

```
<ui:encode enc="html">
  ...
</ui:encode>
```

The output encoding can be changed to a different style with this element. In particular, it is possible to add encodings to the currently working list of encodings, so that in addition to the `html` encoding the encodings specified by the `enc` attribute are in effect.

It is also possible to put `ui:encode` inside *ui:special* (→ 156) elements.

39.1 Declaration

Level: Control structure

```
<!ELEMENT ui:encode ANY>

<!ATTLIST ui:encode
          enc     NMTOKENS    #REQUIRED>
```

39.2 Attributes

- `enc`: A space-separated list of encoding tokens. See *Output encodings* (→ 71) for the defined tokens and their meaning.

39.3 Sub elements

All page body elements may occur in `ui:encode`.

ui:enum-value and ui:enum-item

Web Path: WDialog / Reference / The UI language / ui:enum-value and ui:enum-item

40 The elements ui:enum-value and ui:enum-item

The element `ui:enum-value` represents an enumerator literal that can be used to set the initial value of a *ui:variable* (→ 170). The element `ui:enum-item` represents one enumerated item.

40.1 Declaration

Level: Dialog structure

```
<!ELEMENT ui:enum-value (ui:enum-item)* >

<!ELEMENT ui:enum-item EMPTY>

<!ATTLIST ui:enum-item
          internal  NMTOKEN  #REQUIRED>
```

40.2 Attributes

- `internal`: The internal value identifying the enumerated item

40.3 Example

```
<ui:enumeration name="fruit">
  <ui:enum internal="apple"  external="I like apples"/>
  <ui:enum internal="orange" external="I like oranges"/>
  <ui:enum internal="banana" external="I like bananas"/>
</ui:enumeration>

<ui:variable name="preference" type="fruit">
  <ui:enum-value>
    <ui:enum-item internal="orange"/>
    <ui:enum-item internal="banana"/>
  </ui:enum-value>
</ui:variable>
```

ui:enumerate

Web Path: WDialog / Reference / The UI language / ui:enumerate

41 The element ui:enumerate

This elements instantiates a template several times. It is possible to enumerate the members of a declared enumerator type, and it is possible to enumerate the values of an enumerator variable (declared or dynamic). For every member the template is expanded, and the resulting texts are concatenated.

41.1 $int and $ext

The element iterates over the members of the given type or variable. For every member, the referenced template is called, and the lexical parameters `$int` and `$ext` identify the member. These parameters are always passed to the template (and additionally, the parameters given by `ui:param`, and the current context parameters are passed, too).

The parameter `$int` is set to the internal value of the member, and the parameter `$ext` is set to the corresponding external value of the member while the template is being expanded for the corresponding member.

41.2 Declaration

Level: Control structure

```
<!ELEMENT ui:enumerate ( ( ui:param )*,
                         ( ui:iter-empty )?,
                         ( ui:iter-head )?,
                         ( ui:iter-foot )?,
                         ( ui:iter-separator )?
                       )
>

<!ATTLIST ui:enumerate
          template NMTOKEN   #REQUIRED
          type     NMTOKEN   #IMPLIED
          variable NMTOKEN   #IMPLIED
          index    CDATA     #IMPLIED
>
```

41.3 Attributes

- `template`: Names the template to instantiate. Note that the current language of the dialog may also influence which template is selected.
- `type`: The type to enumerate. There must be an *ui:enumeration* (→ 111) for this type. If `type` is present, neither `variable` nor `index` must be specified.

- `variable`: Selects the variable containing the enumerated values. This variable must be either a declared or a dynamic enumerator. If `variable` is present, the attribute `type` must not be specified.
- `index`: If the variable is an associative list over enumerators, this attribute selects the component to use.

41.4 Sub elements

- *ui:param* (→ 140): Passes additional lexical parameters to the called template.
- *ui:iter-empty* (→ 131): If existing, and if the iterated variable is empty, the nodes below `ui:iter-empty` are expanded instead of the template.
- *ui:iter-head* (→ 131): If existing, and the iterated variable is non-empty, the nodes below `ui:iter-head` are prepended to the concatenated template instances.
- *ui:iter-foot* (→ 131): If existing, and the iterated variable is non-empty, the nodes below `ui:iter-foot` are appended to the concatenated template instances.
- *ui:iter-separator* (→ 131): If existing, and the iterated variable contains more than one component, the nodes below `ui:iter-separator` are placed between the template instances.

41.5 Example

```
<ui:dialog name="preferred-colors" start-page="input-color">
  <ui:enumeration name="color">
    <ui:enum internal="ff0000" external="red"/>
    <ui:enum internal="00ff00" external="green"/>
    <ui:enum internal="0000ff" external="blue/>
  </ui:enumeration>

  <ui:variable name="prefcol" type="color"/>

  <ui:page name="input-color">
    <html>
      <body>
        Select your preferred colors:
        <ui:select variable="prefcol" multiple="yes"/>
        <ui:button name="ok" label="OK" goto="print-color"/>
      </body>
    </html>
  </ui:page>

  <ui:page name="print-color">
    <html>
      <body>
        You have selected:
        <ui:enumerate variable="prefcol" template="print">
          <ui:iter-empty>Nothing!</ui:iter-empty>
          <ui:iter-sep>, </ui:iter-sep>
        </ui:enumerate>
      </body>
    </html>
```

```
  </ui:page>
</ui:dialog>

<ui:template name="print" from-caller="ext">
  "$ext"
</ui:template>
```

In this example, the variable `prefcol`, a declared enumerator of type `color` is iterated. For all selected colors the template `print` is called which puts the external names into double quotes. Furthermore, the separator is a comma followed by a space character. If no color is selected, the special string "Nothing!" is printed instead.

Assumed that the user has selected red and blue, the output of this example is:

```
You have selected: "red", "blue"
```

ui:enumeration and ui:enum

Web Path: WDialog / Reference / The UI language / ui:enumeration and ui:enum

42 The elements ui:enumeration and ui:enum

The element ui:enumeration declares a new enumerator type; for an introduction to enumerator types see *Data Types* (→ 27).

Once an enumerator type has been declared, it can be referred to in variable declarations (see *ui:variable* (→ 170)) by setting the `type` attribute of the variable to the name of the enumerator type. Furthermore, the new type can be used in the *ui:translate* (→ 165) element to map internal tokens to external representations in the user interface. Another application of enumerator types is the possibility to enumerate their values by the *ui:enumerate* (→ 108) element.

The declaration looks like:

```
<ui:enumeration name="sample_enum">
  <ui:enum internal="int1" external="ext1"/>
  <ui:enum internal="int2" external="ext2"/>
  ...
</ui:enumeration>
```

The `ui:enum` elements specify the possible values of the enumerator. Every value has an internal and an external representation; the internal value is used to identify the value in programs while the external value is taken to display the value in the user interface. If the external value is missing, it defaults to the same value as the internal value.

Variables of the declared enumerator have set values: The set is a subset of the values specified in the enumerator declaration. For example, the variable

```
<ui:variable name="sample_var" type="sample_enum"/>
```

has possible values: {}, {"int1"}, {"int2"}, {"int1","int2"}, and so on.

The order of the ui:enum declarations determines the order of the enumerated values. Especially, this order is used when enumerators are visualized (for example, by *ui:select* (→ 150) interactors).

42.1 Declaration

Level: Dialog structure

```
<!ELEMENT ui:enumeration ( ui:enum )*>

<!ATTLIST ui:enumeration
          name NMTOKEN #REQUIRED>
```

```
<!ELEMENT ui:enum EMPTY>

<!ATTLIST ui:enum
          internal NMTOKEN  #REQUIRED
          external CDATA    #IMPLIED>
```

42.2 Attributes

The `ui:enumeration` element has only one attribute:

- `name`: The name of the enumerator type

The `ui:enum` element has these attributes:

- `internal`: Specifies the internal token of an enumerated value
- `external`: Optionally, this attributes specifies the external representation of an enumerated value. If missing, the external value defaults to the same string as the internal value.

42.3 Example

```
<ui:dialog name="print_list" ...>
  <ui:enumeration name="list_length_type">
    <ui:enum internal="10" external="10 items per page"/>
    <ui:enum internal="20" external="20 items per page"/>
    <ui:enum internal="50" external="50 items per page"/>
    <ui:enum internal="100" external="100 items per page"/>
  </ui:enumeration>
  ...
  <ui:variable name="list_length" type="list_length_type">
    <!-- The default value of this variable: -->
    <ui:enum-value>
      <ui:enum-item internal="10"/>
    </ui:enum-value>
  </ui:variable>
  ...
  <ui:page name="select_list_length">
    <html>
      <body>
        Please select the number of items per page:
        <ui:select variable="list_length"/>
        ...
      </body>
    </html>
  </ui:page>
  ...
</ui:dialog>
```

ui:false

Web Path: WDialog / Reference / The UI language / ui:false

43 The element ui:false

This element ignores its inner nodes without expanding them. Furthermore, this element sets the condition code to `false`. The condition code can be evaluated by *ui:cond* (→ 94).

43.1 Declaration

Level: Control structure

```
<!ELEMENT ui:false ANY>
```

43.2 Sub elements

All page body elements may occur in `ui:false`.

43.3 Example

```
<ui:false>
  ...material to ignore...
</ui:false>
```

43.4 Hints

Note that template parameters and bracket expressions within `ui:false` are evaluated. See *ui:cond* (→ 94) for a discussion of the consequences of this fact.

ui:file

Web Path: WDialog / Reference / The UI language / ui:file

44 The element ui:file

The `ui:file` element displays a file upload widget. The generated HTML output consists of an `INPUT` element with `TYPE=FILE` whose name attribute is set to a special identifier which is recognized by the system when the form is submitted.

The name of the `ui:file` box is specified by the `name` attribute. The dialog object provides access methods to find out whether a file has been uploaded, and if yes, where it is stored. The file is a temporary file being automatically deleted after the `handle` callback method has returned to the caller. It is allowed to move the file away to a different location in the filesystem. An example:

```
<ui:dialog name="sample" start-page="sample_page">
  <ui:page name="sample_page">
    <html>
      <body>
        <ui:file name="my_upload"/>
        <ui:button label="OK" name="ok"/>
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

The corresponding code of the `handle` method - O'Caml:

```
method handle() =
  match self # event with
    Button "ok" ->
      (* Somebody pressed "OK", so we check if there is an uploaded file: *)
      ( match self # lookup_uploaded_file "my_upload" with
          None ->
            (* No file! *)
            ...
        | Some arg ->
            (* There is a file encoded as arg : Netcgi_types.cgi_argument.
             * See the documentation of the Netcgi_types module contained
             * in the netstring/ocamlnet package for details.
             *)
          ( match arg # representation with
              `Memory -> assert false        (* Impossible case *)
            | `File filename ->
                (* "filename" is the file where the contents of the uploaded
                 * file are currently stored. For example:
                 *)
```

```
                Sys.rename filename "/other/location";
            )
        )
    | ... ->
        (* Other cases. You need not to check whether there is a temporary
         * upload file to delete, as this is done automatically.
         *)
```

Here is the same as Perl code:

```
sub handle {
    my ($self) = @_;
    my ($e, $name) = $self->event();
    if ($e eq 'BUTTON' && $name eq 'ok') {
        # Somebody pressed "OK", so we check if there is an uploaded file:
        my ($username, $mimetype, $filename) = $self->uploaded_file("my_upload");
        if ($filename eq '') {
            # No file!
            ...
        } else {
            # $filename is the file where the contents of the uploaded
            # file are currently stored. For example:
            rename($filename, "/other/location");
        }
    } elsif (...) {
        # Other cases. You need not to check whether there is a temporary
        # upload file to delete, as this is done automatically.
        ...
    }
```

44.1 Declaration

Level: Generative

```
<!ELEMENT ui:file EMPTY>
<!ATTLIST ui:file
          name     NMTOKEN     #REQUIRED
          cgi     (auto|keep)  "auto"
>
```

Additionally, `ui:file` must only occur inside `ui:form`.

44.2 Attributes

- `name`: Specifies the name of the file upload widget.

- `cgi:` The value "auto" means that the name of the CGI variable associated with the widget is selected automatically. This works perfectly unless you want to refer to this variable from Javascript or from some other manually written event decoder. The value "keep" causes that the name of the CGI variable is predictable: it is `upload_` concatenated with the name of the widget. In this case the widget name should only contain alphanumeric characters, because not all punctuation characters can be safely transported over the CGI protocol.

If there are any other attributes, these are added to the generated `INPUT` HTML element. However, there is rarely an application for this possibility.

44.3 Sub elements

`ui:file` does not have sub element.

44.4 Generated HTML code

The `ui:file` element generates HTML code which roughly looks as follows:

```
<input type="FILE" name="...">
```

ui:form

Web Path: WDialog / Reference / The UI language / ui:form

45 The element ui:form

This element begins a form that may contain form elements like input boxes and buttons. The generated HTML code consists of a form element and many hidden input fields keeping the current state of the dialog.

In order to include form elements, an `ui:form` element is required, and all form elements must only occur within `ui:form`. Furthermore, an `ui:form` is required, too, if the page contains hyperlinks (*ui:a* (→ 80)). Furthermore, an `ui:form` is required if the page refers to popup pages.

It is recommended to put the `ui:form` element into a table, because many browsers render the table only once it is completely loaded. This avoids that the user presses buttons before the `ui:form` element is loaded which causes incorrect behaviour of the system. However, newer browsers supporting DOM level 2 render tables even if they are incomplete such that the trick no longer works. (Note that submitting clearly incomplete forms should be considered as a bug of the browsers.)

45.1 Declaration

Level: Generative element

```
<!ELEMENT ui:form ANY>

<!ATTLIST ui:form
          action-suffix CDATA "">
```

The children may be any elements that are allowed in page context (see *ui:page* (→ 135) for an overview); however, it is not allowed that there is a second `ui:form` element anywhere in the current page.

45.2 Attributes

- `action-suffix`: This string is appended to the automatically generated ACTION attribute. The string should begin with a slash character.

The element `ui:form` defines almost no attributes; however, if there are attributes these are added to the generated HTML `form` element. Especially `onsubmit` and `onreset` work. Note that the attributes `name`, `action`, `method`, `accept-charset`, and `enctype` are generated and if these occur inside `ui:form` they will be ignored.

45.3 Sub elements

All elements allowed in a page body can occur within `ui:form`.

45.4 Generated HTML code

The `ui:form` element generates HTML code which roughly looks as follows:

```
<form name="uiform" action="..." method="post" enctype="multipart/form-encoded"
      accept-charset="...">
  <input type="hidden" ...>
  <input type="hidden" ...>
  ... (Sub elements)
  <input type="hidden" ...>
  <input type="hidden" ...>
</form>
```

Note that the generated form element has always the name `uiform` such that it is possible to access the element from Javascript code. - If there are attributes in the `ui:form` element, these are added to the generated form element.

Sometimes, `ui:form` generates a second form element

```
<form name="uialtform" ...>
  ...
</form>
```

immediately after `uiform`.

45.5 Example

A page with a button:

```
<ui:page name="sample">
  <html>
    <body>
      <ui:form>
        <h1>A sample page</h1>
        You can press on this
        <ui:button name="sample_button" label="Button"/>
      </ui:form>
    </body>
  </html>
</ui:page>
```

ui:if

Web Path: WDialog / Reference / The UI language / ui:if

46 The element ui:if

This element compares two values. If the result is true, the inner nodes are expanded and the condition code is set to `true`; otherwise the inner nodes are ignored, and the conditon code is set to `false`.

46.1 Declaration

Level: Control structure

```
<!ELEMENT ui:if ANY>

<!ATTLIST ui:if
          value1   CDATA         #REQUIRED
          value2   CDATA         #REQUIRED
          op       CDATA         "eq"
>
```

46.2 Attributes

- `value1`: The first operand of the comparison
- `value2`: The second operand of the comparison
- `op`: The comparison operator

Since WDialog-2.1 it is allowed to pass more than two operands to the comparison function, although there is not any application for this feature in WDialog itself. User-defined comparison functions may take advantage from this, however. Additional operands are passed as attributes `value3`, `value4` and so on.

46.3 Sub elements

All page body elements may occur as sub elements.

46.4 Operators

The following operators are defined for `ui:if`:

- `eq`: string equality
- `ne`: string inequality

- `match`: the first operand matches the regular expression denoted by the second operand (see notes on regular expressions below)
- `nomatch`: the first operand does not match the regular expression denoted by the second operand (see notes on regular expressions below)
- `int-eq`: integer equality
- `int-ne`: integer inequality
- `int-lt`: the first operand is less than the second operand
- `int-le`: the first operand is less or equal than the second operand
- `int-gt`: the first operand is greater than the second operand
- `int-ge`: the first operand is greater or equal than the second operand

Since WDialog-2.1, it is possible to call any function defined for bracket expressions. The function is called with two (or more) strings as arguments, and it is expected that the function returns a number indicating a boolean result: 0 is considered as the false value, and a non-zero number is considered as the true value. Currently, there aren't any other functions that would make sense to be called. The user may, however, define further functions.

46.5 Regular expressions

The engine matching regular expressions is PCRE (Perl-compatible regular expressions), so the Perl syntax is used. The expressions are not anchored by default, so have to write `^` and `$$` to force anchoring. `^` matches the beginning of the string, and `$$` matches the end of the string ("single-line expressions"). Note that you have to write double dollars `$$` because the dollar character is the escape character for template parameters.

46.6 Hints

Note that template parameters and bracket expressions within `ui:if` are unconditionally evaluated. See *ui:cond* (→ 94) for a discussion of the consequences of this fact.

46.7 Example

```
<ui:if value1="$[n]" value2="1" op="int-gt">
  The result has more than one solution: ...
</ui:if>
```

ui:ifexpr

Web Path: WDialog / Reference / The UI language / ui:ifexpr

47 The element ui:ifexpr

The element expects in the `expr` attribute a number which is interpreted as boolean value. If the number is non-zero, the inner nodes of the element are expanded and the condition code is set to `true`; otherwise the inner nodes are ignored, and the conditon code is set to `false`.

47.1 Declaration

Level: Control structure

```
<!ELEMENT ui:ifexpr ANY>

<!ATTLIST ui:ifexpr
          expr CDATA #REQUIRED
>
```

47.2 Attributes

- `expr`: An integer number. A non-zero value as considered as true boolean value, and the number zero is considered as false boolean value. The number is usually the result of the evaluation of a bracket expression.

47.3 Sub elements

All page body elements may occur as sub elements.

47.4 Hints

Note that template parameters and bracket expressions within `ui:if` are unconditionally evaluated. See *ui:cond* (→ 94) for a discussion of the consequences of this fact.

47.5 Example

```
<ui:ifexpr expr="$[int-gt(n,1)]">
  The result has more than one solution: ...
</ui:ifexpr>
```

ui:iflang

Web Path: WDialog / Reference / The UI language / ui:iflang

48 The element ui:iflang

This element checks whether the current language is the language specified by `xml:lang`. If the result is true, the inner nodes are expanded and the condition code is set to `true`; otherwise the inner nodes are ignored, and the conditon code is set to `false`.

For an overview about multi-language support, see the chapter about *Internationalization* (→ 67).

48.1 Declaration

Level: Control structure

```
<!ELEMENT ui:iflang ANY>

<!ATTLIST ui:iflang
          xml:lang CDATA #REQUIRED>
```

48.2 Attributes

- `xml:lang`: Specifies the language to check for.

48.3 Sub elements

All page body elements may occur as sub elements.

48.4 The l namespace

This element can be abbreviated as follows: Instead of

```
<ui:iflang xml:lang="TOKEN">TREE</ui:iflang>
```

one can write

```
<l:TOKEN>TREE</l:TOKEN>
```

48.5 Example

```
<ui:cond>
  <l:en>One</l:en>
  <l:de>Eins</l:de>
  <l:it>Uno</l:it>
  <l:es>Uno</l:es>
</ui:cond>
```

Due to the implementation it is currently not recommended to use the combination `ui:cond/ui:iflang` as substitute for message catalogues if the number of languages is bigger than a small number (three or four). (May be improved in the future, however.)

48.6 Hints

Note that template parameters and bracket expressions within `ui:iflang` are unconditionally evaluated. See *ui:cond* (→ 94) for a discussion of the consequences of this fact.

ui:ifvar

Web Path: WDialog / Reference / The UI language / ui:ifvar

49 The element ui:ifvar

This element compares a dialog variable with a value. If the result is true, the inner nodes are expanded and the condition code is set to `true`; otherwise the inner nodes are ignored, and the conditon code is set to `false`.

The domain of comparisons that can be performed by `ui:ifvar` overlaps with *ui:if* (→ 119). Especially the comparison of a non-associative variable with a value can be expressed by both elements. In doubt, use the shorter and more common *ui:if* (→ 119).

49.1 Declaration

Level: Control structure

```
<!ELEMENT ui:ifvar ANY>

<!ATTLIST ui:ifvar
          variable NMTOKEN           #REQUIRED
          index    NMTOKEN           #IMPLIED
          value    CDATA             #REQUIRED
          op       CDATA             "eq"
>
```

49.2 Attributes

- `variable`: The name of the variable that is compared with the value.
- `index`: If the variable is associative, the index can select the component. However, it is also possible to compare associative variables as such if one of the list operators is applied.
- `value`: The second operand of the comparison besides the value of the variable.
- `op`: Selects the comparison operator

49.3 Sub elements

All page body elements may occur as sub elements.

49.4 Operators

The following operators are defined for `ui:if` (we call the value of the variable the "first operand", and the value denoted by the `value` attribute the "second operand"):

- *Operators for string variables*
  - `eq`: string equality
  - `ne`: string inequality
  - `match`: the first operand matches the regular expression denoted by the second operand (see notes on regular expressions below)
  - `nomatch`: the first operand does not match the regular expression denoted by the second operand (see notes on regular expressions below)
  - `int-eq`: integer equality
  - `int-ne`: integer inequality
  - `int-lt`: the first operand is less than the second operand
  - `int-le`: the first operand is less or equal than the second operand
  - `int-gt`: the first operand is greater than the second operand
  - `int-ge`: the first operand is greater or equal than the second operand
- *Operators for enumerators (declared or dynamic), and for associative variables*
  - `contains`: The first operand contains the second operand as component (key). For enumerators, this means that the first operand has an internal value equal to the second operand. For associations, this means that the first operand has a component indexed by the second operand.
  - `mentions`: The first operand contains the second operand as component (value). For dynamic enumerators, this means that the first operand has an external value equal to the second operand (the operation is not defined for declared enumerators). For associations, this means that the first operand has a component in whose value the second operand occurs.
  - `card-eq`, `card-ne`, `card-lt`, `card-le`, `card-gt`, `card-ge`: The cardinality of the first operand is compared with the number in the second operand.
- *Operators for variables of type* `dialog`
  - `dialog-exists`: The first operand is of dialog type, and the second operand is either `yes` or `no`. The comparison yields a true result, if the first operand contains a dialog and the second operand is `yes`, or if the first operand does not contain a dialog, and the second operand is `no`.

Since WDialog-2.1, it is possible to call any function defined for bracket expressions. The function is called with two arguments, and it is expected that the function returns a number indicating a boolean result: 0 is considered as the false value, and a non-zero number is considered as the true value. Currently, there aren't any other functions that would make sense to be called. The user may, however, define further functions.

49.5 Regular expressions

The engine matching regular expressions is PCRE (Perl-compatible regular expressions), so the Perl syntax is used. The expressions are not anchored by default, so have to write `^` and `$$` to force anchoring. `^` matches the beginning of the string, and `$$` matches the end of the string ("single-line expressions"). Note that you have to write double dollars `$$` because the dollar character is the escape character for template parameters.

49.6 Example

```
<ui:dialog start-page="poll">
  <ui:enumeration name="fruit">
    <ui:enum internal="apple" external="I like apples"/>
    <ui:enum internal="orange" external="I like oranges"/>
    <ui:enum internal="bananas" external="I like bananas"/>
    <ui:enum internal="ananas" external="I like ananas"/>
  </ui:enumeration>

  <ui:variable name="pref" type="fruit"/>

  <ui:page name="poll">
     ...
     Please click at your preferred fruit:
     <ui:select variable="pref" multiple="yes"/>
     ...
  </ui:page>

  <ui:page name="check">
    ...
    <ui:ifvar variable="pref" value="0" op="card-eq">
      You don't have selected any fruit sort. ...
    </ui:ifvar>
    ...
  </ui:page>
</ui:dialog>
```

49.7 Hints

Note that template parameters and bracket expressions within `ui:ifvar` are unconditionally evaluated. See *ui:cond* (→ 94) for a discussion of the consequences of this fact.

ui:imagebutton

Web Path: WDialog / Reference / The UI language / ui:imagebutton

50 The element ui:imagebutton

This element displays an imagebutton, i.e. a button rendered as an image. The generated HTML code consists of an `INPUT` element with `TYPE=IMAGE`, whose `name` attribute is set to a special identifier which is recognized by the system when the form is submitted.

When the user clicks on the button, an `Image_button` event is generated (unless the `index` attribute is specified; see below); the `handle` callback method of the dialog object can check whether the current event is the event associated with this button, and the method can execute code depending on the result of this check. For a description of possible events see *Events* (→ 41). The following example illustrates image button events:

```
<ui:dialog name="sample" start-page="p1">
  <ui:page name="p1">
    <html>
      <body>
        <h1>Button test</h1>
        This is a <ui:imagebutton name="b" src="button.gif"/>
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

Here, the image button event has the name "b". In order to check whether this event occured in the `handle` method, the following piece of code is recommended. O'Caml:

```
method handle =
  match self # event with
    Image_button("b",x,y) ->
      ... (* Do whatever you want to do *)
  | ... (* other cases *)
```

- Perl:

```
sub handle {
    my ($self) = @_;
    my ($e, $name, $x, $y) = $self->event;
    if ($e eq 'IMAGE_BUTTON' && $name eq 'b') {
        ... # Do whatever you want to do
    } elsif ... # other cases
    ;
```

```
    return undef;
}
```

In both cases, the variables x and y contain the position of the click relative to the origin of the button.

If the `ui:imagebutton` element sets the `index` attribute, the button is identified by the pair `(name,index)`. When the user clicks on such an indexed imagebutton, an `Indexed_image_button` event is generated. The index value can be used to distinguish between several instances of buttons of the same type. For instance, a book store may offer the customer several books:

```
<ui:dialog name="sample" start-page="view_records">
  <ui:page name="view_records">
    <html>
      <body>
        <h1>View books</h1>
        <table>
          <tr>
            <th>Author</th>
            <th>Title</th>
            <th>Action</th>
          </tr>
          <tr>
            <td>Damon Runryon</td>
            <td>Guys and Dolls</td>
            <td><ui:imagebutton name="view" src="view.gif" index="4523"/></td>
          </tr>
          <tr>
            <td>William S. Burroughs</td>
            <td>Naked Lunch</td>
            <td><ui:imagebutton name="view" src="view.gif" index="8612"/></td>
          </tr>
        </table>
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

Here, the index value is the database ID of the record. The typical code to check for such a button in the `handle` callback is - O'Caml:

```
method handle =
  match self # event with
    Indexed_image_button("view", index, x, y) ->
      ... (* Do whatever you want to do *)
  | ... (* other cases *)
```

- Perl:

```
sub handle {
```

```
    my ($self) = @_;
    my ($e, $name, $index, $x, $y) = $self->event;
    if ($e eq 'INDEXED_IMAGE_BUTTON' && $name eq 'view') {
        ... # Do whatever you want to do
    } elsif ... # other cases
    ;
    return undef;
}
```

Note that the transport mechanism for the strings specified for name and/or index is 8 bit clean (at least if `cgi="auto"`). This means that the name and index strings may be composed of all characters of the character set.

50.1 Declaration

Level: Generative

```
<!ELEMENT ui:imagebutton EMPTY>
<!ATTLIST ui:imagebutton
          name  NMTOKEN     #REQUIRED
          index CDATA       #IMPLIED
          src   CDATA       #REQUIRED
          align CDATA       #IMPLIED
          goto  NMTOKEN     #IMPLIED
          cgi   (auto|keep) "auto"
>
```

Additionally, `ui:imagebutton` must only occur inside `ui:form`.

50.2 Attributes

The following attributes have a special meaning:

- `name`: Specifies the name of the button.
- `index`: Specifies the index value of the button. If this attribute is present, the button becomes an indexed button; otherwise the button is a plain button.
- `src`: Specifies the image file containing the image bitmap.
- `align`: Specifies the `align` attribute of the generated HTML element. Defaults to `BOTTOM` according to HTML standards.
- `goto`: Specifies which page is the next page if the button is pressed. The variable containing the next page is initialized with the name specified here before the `handle` method is invoked. This means that the action of the button is to go to this page, unless the action is overridden in the `handle` method.
- `cgi`: The value "auto" means that the name of the CGI variable associated with the button is selected automatically. This works perfectly unless you want to refer to this variable from Javascript or from some other manually written event decoder. The value "keep" causes that the name of the CGI variable is predictable: it is `imagebutton_`

concatenated with the name of the button. However, it is not allowed to specify "keep" if there is also an index value. Furthermore, the button name should only contain alphanumeric characters, because not all punctuation characters can be safely transported over the CGI protocol.

If there are any other attributes, these are added to the generated `INPUT` HTML element. This means that especially the `onclick` attribute may be specified.

50.3 Sub elements

`ui:imagebutton` elements do not have sub elements.

50.4 Generated HTML code

The `ui:imagebutton` element generates HTML code which roughly looks as follows:

```
<input type="IMAGE" name="..." src="..." align="...">
```

ui:iter-*

*Web Path: WDialog / Reference / The UI language / ui:iter-**

51 The elements ui:iter-empty, ui:iter-head, ui:iter-foot, and ui:iter-separator

These elements must be used inside *ui:iterate* (→ 132), or *ui:enumerate* (→ 108), and are explained there.

51.1 Declaration

Level: Control structure

```
<!ELEMENT ui:iter-empty ANY>
<!ELEMENT ui:iter-head ANY>
<!ELEMENT ui:iter-foot ANY>
<!ELEMENT ui:iter-separator ANY>
```

51.2 Sub elements

All page body elements may occur inside the `ui:iter` particles.

ui:iterate

Web Path: WDialog / Reference / The UI language / ui:iterate

52 The element ui:iterate

This elements instantiates a template several times. For every component of a compound variable the template is expanded, and the resulting texts are concatenated. The following types of variables can be iterated:

- *Dynamic enumerators:* The members of the set are iterated in turn
- *Associations:* The elements of the associative variable are iterated in turn
- *Strings:* Even strings can be iterated. The string is regarded as a list of words; the words must be separated by white space.

For an overview over templates, see the chapter *Templates* (→ 49).

See also the similar element *ui:enumerate* (→ 108) that works for declared enumerators.

52.1 $int and $ext

The element iterates over the components of the given variable. For every component, the named template is called, and the lexical parameters `$int` and `$ext` identify the components. These parameters are always passed to the template (and additionally, the parameters given by `ui:param`, and the current context parameters are passed, too).

If the variable is a dynamic enumerator, `$int` will be set to the internal values of the components, and `$ext` will be set to the external values of the components.

If the variable is associative, `$int` will be set to the keys of the components, and `$ext` will be set to the associated values of the components, if possible. The latter is only defined for associative string variables. For other types of associations, the parameter `$ext` is simply empty.

If the variable is a string (list of words), `$int` will be set to the index number of the words (0 for the first word, 1 for the second word, etc.). The parameter `$ext` will be set to the words themselves.

52.2 Declaration

Level: Control structure

```
<!ELEMENT ui:iterate ( ( ui:param )*,
                       ( ui:iter-empty )?,
                       ( ui:iter-head )?,
                       ( ui:iter-foot )?,
                       ( ui:iter-separator )?
                     )
>
```

```
<!ATTLIST ui:iterate
          template NMTOKEN   #REQUIRED
          variable NMTOKEN   #REQUIRED
          index    CDATA     #IMPLIED
>
```

52.3 Attributes

- `template`: Names the template to instantiate. Note that the current language of the dialog may also influence which template is selected (see below).
- `variable`: Selects the variable containing the iterated values. This variable must be either a dynamic enumerator, or an association, or a string.
- `index`: If the variable is an associative list over dynamic enumerators or strings, this attribute selects the component to use.

52.4 Sub elements

- *ui:param* (→ 140): Passes additional lexical parameters to the called template.
- *ui:iter-empty* (→ 131): If existing, and if the iterated variable is empty, the nodes below `ui:iter-empty` are expanded instead of the template.
- *ui:iter-head* (→ 131): If existing, and the iterated variable is non-empty, the nodes below `ui:iter-head` are prepended to the concatenated template instances.
- *ui:iter-foot* (→ 131): If existing, and the iterated variable is non-empty, the nodes below `ui:iter-foot` are appended to the concatenated template instances.
- *ui:iter-separator* (→ 131): If existing, and the iterated variable contains more than one component, the nodes below `ui:iter-separator` are placed between the template instances.

52.5 Internationalization

If a certain language is selected for the dialog, this also affects the template system. In particular, it is first checked if the used template is defined for this language, and if so, this version of the template will be used. Otherwise, it is checked whether there is a template without `xml:lang` attribute, and if it can be found, this version will be used.

For more information, see the chapter about *Internationalization* (→ 67).

52.6 Example

```
<ui:dialog name="sample" start-page="show">
  <ui:variable name="fruit_index">
    <ui:string-value>0 3</ui:string-value>
  </ui:variable>

  <ui:variable name="fruit" type="dynamic-enumerator">
    <ui:dyn-enum-value>
      <ui:dyn-enum-item internal="0" external="apple"/>
```

```
      <ui:dyn-enum-item internal="1" external="ananas"/>
      <ui:dyn-enum-item internal="2" external="pineapple"/>
      <ui:dyn-enum-item internal="3" external="banana"/>
    </ui:dyn-enum-value>
  </ui:variable>

  <ui:page name="show">
    <html>
      <body>
        <p>Available fruit:
          <ui:iterate variable="fruit" template="display-fruit"/>
        </p>

        <p>Selected fruit according to index:
          <ui:iterate variable="fruit_index" template="display-fruit-idx"/>
        </p>
      </body>
    </html>
  </ui:page>
</ui:dialog>

<ui:template name="display-fruit" from-caller="ext">
  <tt>$ext</tt>
</ui:template>

<ui:template name="display-fruit-idx" from-caller="int">
  <tt>$[translate(fruit,$int)]</tt>
</ui:template>
```

ui:page

Web Path: WDialog / Reference / The UI language / ui:page

53 The element ui:page

Every dialog object consists of one or several pages, which are the HTML documents visualizing the state of the object. When the web browser sends a request to the WDialog system, the reply is one of the pages of the current dialog object. Because of this, pages are the units of communications to the browser.

For example, the following page defines a minimal HTML document:

```
<ui:page name="sample">
  <html>
    <head>
      <title>This is a sample page</title>
    </head>
    <body>
      This is the body of the sample page.
    </body>
  </html>
</ui:page>
```

The dialog object stores the name of the current page, and this name is used to select the page replied to the browser. In detail, the dialog object distinguishes between the *current page* and the *next page*. This can be explained by illustrating the actions taken by the WDialog system to display the next page:

- When the user clicks on a button or a hyperlink, this event causes that the `handle` callback of the dialog object is invoked. At this time, the current page is still the old page, and the next page is the designated new page (either the again the old page, or the page specified by the `goto` attribute of the button or hyperlink element). The callback method can now change the name of the next page.
- Now the page change occurs: The name of the next page is stored in the variable containing the name of the current page.
- The `prepare_page` callback is invoked, and at this moment, the current page is the new page. (You can still ask the dialog object for the "next page"; however it will simply return the current page as there is currently no next page.)
- The final HTML document is created by expanding the current page until all templates are instantiated. This document is sent back to the browser.

For instance, there is a possible page change from sample1 to sample2 in the following example:

```
<ui:page name="sample1">
  <html>
    <head>
      <title>This is sample page 1</title>
```

```
    </head>
    <body>
      <ui:a name="mylink" goto="sample2">Click here to go to page 2</ui:a>
    </body>
  </html>
</ui:page>

<ui:page name="sample2">
  <html>
    <head>
      <title>This is sample page 2</title>
    </head>
    <body>
      This is the body of the sample page 2.
    </body>
  </html>
</ui:page>
```

It is assumed that the dialog object starts with sample1. This page contains a hyperlink (*ui:a* (→ 80)) with a `goto` attribute pointing to sample2. When the user clicks on this hyperlink, the current page is sample1, and the designated next page is sample2. The `handle` callback of the dialog object can now change the name of the next page if this is required for some reason. Unless this actually happens, the next page remains to be sample2, and the current page changes to sample2. The `prepare_page` callback will already find that sample2 is the current page. Finally, sample2 is displayed as the next page on the browser window.

The first page of a dialog object can be specified in the `start-page` attribute of the *ui:dialog* (→ 98) element. Example:

```
<ui:dialog name="sampleobject" start-page="sample1">
  <!-- Now ui:pages sample1 and sample2 -->
  ...
</ui:dialog>
```

A page is a template that is called by WDialog. It may contain template parameters, and it can declare parameters using `from-caller` and `from-context`. For example, the following page displays the sentence specified in the parameter `s` five times:

```
<ui:page name="repeater" from-caller="s">
  <ui:default name="s">
    I have to write this sentence five times.
  </ui:default>
  <html>
    <body>
      $s $s $s $s $s
    </body>
  </html>
</ui:page>
```

Note that the usual rules for template parameters apply: The `ui:default` element defines the default value of the parameter which is used if there is no directly passed parameter value. As pages are automatically called, it is not possible to

pass parameters directly to pages. Because of this, the default value is always taken by the system.

This simply means: In the context of pages, it is possible to bind a parameter `p` locally to a value `v` by including the statement

```
<ui:default name="p">v</ui:default>
```

at the beginning of the ui:page element, and by declaring `p` using `from-caller="p"`.

It is also possible that several pages share parameters. The shared parameters must be defined in the *ui:context* (→ 96) section of the dialog object, and the parameters must be imported by `from-context`. Example:

```
<ui:dialog name="sample" start-page="sample1">
  <ui:context>
    <ui:param name="bgcolor">#342312</ui:param>
  </ui:context>

  <ui:page name="sample1" from-context="bgcolor">
    <html>
      <body bgcolor="$bgcolor">
         ...
      </body>
    </html>
  </ui:page>

  <ui:page name="sample2" from-context="bgcolor">
    <html>
      <body bgcolor="$bgcolor">
         ...
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

In general, the `ui:context` section specifies the parameter context valid for the whole dialog; i.e. when the instantiation procedure begins, the current context for this procedure is initialized with the parameter values defined in `ui:context`. Because of this early binding rule, the parameters can also be imported into pages. - Note that the `ui:context` section affects all template instantiations within pages, too; i.e. context parameters can be imported by any template invoked directly or indirectly from the page.

For general information about templates and the instantiation mechanism, see the *Templates* (→ 49) section.

53.1 Declaration

Level: Both dialog structure and control structure

```
<!ELEMENT ui:page ANY>
<!ATTLIST ui:page
```

```
            name         NMTOKEN  #REQUIRED
            from-caller  NMTOKENS #IMPLIED
            from-context NMTOKENS #IMPLIES
            popup        (yes|no) "no"
    >
```

The subelements of `ui:page` must match the informal rule `( ui:default*, %page-body;* )` where `%page-body;` stands symbolically for all allowed sub elements. Note that whitespace between the `%page-body;` elements counts, but at the other places it does not count.

53.2 Attributes

- `name:` Specifies the name of the page. The name must be unique among all page of the current dialog.
- `from-caller:` Lists the parameters with lexical scope that occur inside the page body.
- `from-context:` Lists the parameters with dynamic scope that occur inside the page body.
- `popup:` If "yes", it is allowed that this page is the target of a *ui:popup* (→ 141) or *ui:server-popup* (→ 154).

53.3 Sub elements

The sub elements `%page-body;` of `ui:page` are either HTML elements, or the following ui elements that generate HTML elements.

- *ui:a* (→ 80)
- *ui:button* (→ 87)
- *ui:checkbox* (→ 91)
- *ui:cond* (→ 94)
- *ui:context* (→ 96)
- *ui:dynamic* (→ 102)
- *ui:encode* (→ 106)
- *ui:enumerate* (→ 108)
- *ui:false* (→ 113)
- *ui:file* (→ 114)
- *ui:form* (→ 117)
- *ui:if* (→ 119)
- *ui:ifexpr* (→ 121)
- *ui:iflang* (→ 122)
- *ui:ifvar* (→ 124)
- *ui:imagebutton* (→ 127)

- *ui:iterate* (→ 132)
- *ui:popup* (→ 141)
- *ui:radio* (→ 144)
- *ui:richbutton* (→ 147)
- *ui:select* (→ 150)
- *ui:server-popup* (→ 154)
- *ui:special* (→ 156)
- *ui:text and ui:password* (→ 160)
- *ui:textarea* (→ 163)
- *ui:translate* (→ 165)
- *ui:true* (→ 167)
- *ui:use* (→ 168)

ui:param

Web Path: WDialog / Reference / The UI language / ui:param

54 The element ui:param

This element defines a parameter to be passed to a template. The element can be used inside *ui:use* (→ 168), *ui:context* (→ 96), *ui:iterate* (→ 132), and *ui:enumerate* (→ 108).

54.1 Declaration

Level: Control structure

```
<!ELEMENT ui:param ANY>

<!ATTLIST ui:param
          name  NMTOKEN   #REQUIRED
>
```

54.2 Attributes

- `name`: The name of the parameter

54.3 Sub elements

The sub elements are the value of the parameter. All page body elements can occur.

ui:popup

Web Path: WDialog / Reference / The UI language / ui:popup

55 The element ui:popup

This element generates a Javascript function that opens another page as popup window. The contents of this page are generated at the time the `ui:popup` element is expanded, which means that the popop window can only contain simple dialogs that are constant from the perspective of the main window. The element *ui:server-popup* (→ 154) is able to create fully dynamic popup windows, however.

The generated Javascript function has the name `popup_<page>` where `<page>` is the identifier found in the `page` attribute (see declaration below). The function takes a string argument which is the window option list known from the Javascript `window.open` function; it specifies the visual properties of the new window. By calling the generated function, the window pops up and displays the constant material found in the referenced page. For example, the following button shows the page `sample` in a new popup window with the specified width and height:

```
<ui:popup page="sample"/>
<input type="button" onclick="open_sample('width=100,height=100')"/>
```

The page, here `sample`, must set the attribute `popup` to `yes`:

```
<ui:page name="sample" popup="yes">
  ...
</ui:page>
```

This is necessary because the generated HTML code is slightly different from that of normal pages.

The popup window can include interactors like text boxes, selection lists, etc., as well as buttons and hyperlinks. The interactors are handled as if they occured on the main window. When pressed, the buttons and hyperlinks close the popup window, and submit the main window. This is very important: Popup windows are not independent of the main window, but they are rather an extension of the main window that is first hidden, and only exhibited on request of the user. Both windows form a unit, and can only be processed as a whole by the WDialog toolkit.

It is only possible to have one open popup window at the same time. Trials to open more than one popup window are silently ignored.

The popup window is automatically closed when any submit button or hyperlink of the main window is pressed, or when the page currently displayed in the main window is left by other means. In these cases, the user interactions in the popup window are ignored.

It is possible to change this behavior by additional Javascript statements:

- You can close the popup window by calling `window.close()` from the popup window, or by calling `close_popup()` from the main window. The user interactions in the popup window will be ignored.
- You can force submission of the popup window by calling `uiform_submit()` from the popup window.

- You can lock the submission of the main window while a popup window is open by setting the `ONSUBMIT` handler of the `ui:form` element of the main window to `return lock_popup()`

Note that WDialog generates an `ONUNLOAD` handler for the main window and a `ONSUBMIT` handler for the popup window.

55.1 Declaration

Level: Generative element

```
<!ELEMENT ui:popup EMPTY>

<!ATTLIST ui:popup
          page  NMTOKEN  #REQUIRED>
```

It is required that there is a `ui:form` in the current page; however, the `ui:popup` element can occur outside the `ui:form` element.

55.2 Attributes

- `page`: The name of the page to display in the popup window. It is required that the `popup` attribute of this page is set to "yes". The Javascript function gets the name `open_` plus the name of the opened page, e.g. `open_menu` if the page is called `menu`.

55.3 Example

```
<ui:dialog name="enter-your-name" start-page="main">
  <ui:variable name="your-name"/>

  <ui:page name="main">
    <html>
      <ui:popup page="popup"/>
      <body>
        <ui:form onsubmit="return lock_popup()">
          Your name is: $[your-name]
          <input type="button" value="Change"
                 onclick="open_popup('width=200,height=200')"/>
          <br/>
          <ui:button name="main_ok" label="OK"/>
        </ui:form>
      </body>
    </html>
  </ui:page>

  <ui:page name="popup" popup="yes">
    <html>
      <body>
        <ui:form>
```

```
            Enter your name:
            <ui:text variable="your-name"/><br/>
            <ui:button name="popup_ok" label="OK"/>
            <input type="button" value="Cancel" onclick="window.close()"/>
          </ui:form>
        </body>
      </html>
    </ui:page>
  </ui:dialog>
```

In this example, the main page shows the current name of the user only as string constant. In order to change it, the user must press the "Change" button, which opens the popup window containing the text box. When the user presses "OK" in the popup window, the new user name is put into the variable, the popup window is closed, and the main window is redisplayed (because of form submission). When the user presses "Cancel" in the popup window, the popup window is closed, and the new name is ignored.

When the popup window is open, the "OK" button of the main window is locked because of the `ONSUBMIT` handler.

55.4 Further Questions

I want that the popup window behaves differently depending on user interactions in the main window. How do I do this? Either you can program the special behavior fully in Javascript, or you use *ui:server-popup* (→ 154) instead. The latter element causes that the popup window is generated after the `open_popup` function is called, and goes back to the server for this.

Can I define my own `ONSUBMIT` *handler for the popup window?* It is not allowed to set the `ONSUBMIT` attribute of the `ui:form` element in popup windows. This attribute is reserved for WDialog. However, you can modify the handler after WDialog has set its own handler. Execute after the whole `ui:form` element these Javascript statements:

```
var wd_onsubmit = document.uiform.onsubmit;
document.uiform.onsubmit = my_handler;
```

Furthermore, program your own handler `my_handler`:

```
function my_handler () {
   ...
   // If you do not want form submission:
   return false;
   ...
   // If you do want form submission:
   return wd_onsubmit();
}
```

ui:radio

Web Path: WDialog / Reference / The UI language / ui:radio

56 The element ui:radio

This element displays a radiobox. The generated HTML code consists of an `INPUT` element with `TYPE=RADIO`, whose name attribute is set to a special identifier which is recognized by the system when the form is submitted.

The radiobox must be tied to an enumerator variable (either a declared one, or a dynamic enumerator), or a string variable. The name of the variable must be specified in the `variable` attribute. Furthermore, there must be a `value` attribute determining which value is visualized by the radiobox. The rule is as follows: The radiobox is in the state "checked" iff the specified value occurs in the set of values currently stored in the specified variable. A string variable is considered as a one-element set for this purpose.

The radiobox widget will be initialized to the state given by this rule when the current page is displayed. All radioboxes referring to the same variable form a group of boxes, and only one of the boxes can be checked at the same time. If the contents of the variable would cause that more than one box is checked, the browser enforces that only one box remains checked (but it is unspecified which box is selected).

Any state change of the widget caused by user interaction will be propagated back to the enumerator variable when the current page is submitted. This means that if the user checks the box the specified value will be added to the specified enumerator, and that conversely if the user releases the box the specified value will be deleted from the specified enumerator variable. In principle, other values contained in the enumerator variable than the specified one remain unchanged; however, the browser will automatically deselect all other radioboxes of the same group if one radiobox is checked, such that normally the other values of the enumerator are deleted.

Of course, the specified value is an internal value with respect to the difference between internal and external values.

In the following example, the user can answer a yes/no question. The variable `user_answer` is initialized with the set {"yes"}, and because of this, the page appears initially with a checked "Yes" box and an unchecked "No" box. The radioboxes simply visualize the current state of the variable. When the customer has given the answer and presses the "OK" button, the variable `user_answer` is automatically updated, and reflects again the current state of the boxes. From the `handle` callback method, one can read the variable `user_answer` and interpret the contents.

```
<ui:dialog name="sample" start-page="sample_page">
  <ui:enumeration name="yesno">
    <ui:enum internal="yes" external="Yes"/>
    <ui:enum internal="no"  external="No"/>
  </ui:enumeration>

  <ui:variable name="user_answer" type="yesno">
    <ui:enum-value>
      <ui:enum-item internal="yes"/>
    </ui:enum-value>
  </ui:variable>

  <ui:page name="sample_page">
    <html>
      <body>
        What is your answer?
```

```
          <ul>
            <li><ui:radio variable="user_answer" value="yes"/>
                Yes</li>
            <li><ui:radio variable="user_answer" value="no"/>
                No</li>
          </ul>
          <ui:button name="ok" label="OK"/>
        </body>
      </html>
    </ui:page>
  </ui:dialog>
```

Note that only the empty set and single-valued sets are reasonable values for the `user_answer` variable. Even if we initialize the variable with multi-valued sets (such as {"yes","no"}), the browser will enforce that only one of the boxes is checked; however, it is unspecified which box remains checked.

56.1 Declaration

Level: Generative

```
<!ELEMENT ui:radio EMPTY>
<!ATTLIST ui:radio
          variable NMTOKEN     #REQUIRED
          index    CDATA       #IMPLIED
          value    NMTOKEN     #REQUIRED
          cgi     (auto|keep)  "auto"
>
```

Additionally, `ui:radio` must only occur inside `ui:form`.

56.2 Attributes

The following attributes have a special meaning:

- `variable`: Specifies the variable of the current dialog object to which the radiobox is tied. Unless the `index` attribute is present, the variable must be a declared enumerator, a dynamic denumerator, or a string. If there is an `index` attribute, the variable must be an associative list of one of the mentioned types.
- `index`: Specifies the index value of the element of the associative variable to which the radiobox is tied.
- `value`: Specifies the internal value whose presence in the enumerator is represented by the radiobox.
- `cgi`: The value "auto" means that the name of the CGI variable associated with the radiobox is selected automatically. This works perfectly unless you want to refer to this variable from Javascript or from some other manually written event decoder. The value "keep" causes that the name of the CGI variable is predictable: it is `var_` concatenated with the name of the variable. However, it is not allowed to specify "keep" if there is also an index value. Furthermore, the variable name should only contain alphanumeric characters, because not all punctuation characters can be safely transported over the CGI protocol.

If there are any other attributes, these are added to the generated `INPUT` HTML element. This means that especially the `onclick` attribute may be specified.

56.3 Sub elements

`ui:radio` does not have sub elements.

56.4 Tips

Often, it is desired to iterate over all defined values of an enumerator, and to output a radiobox for every item. The following code demonstrates how `ui:radio` works in conjunction with *ui:enumerate* (→ 108); it is a another version of the yes/no example:

```
<ui:template name="list_item" from-caller="int ext">
  <li>
    <ui:radio variable="user_answer" value="$int"/> $ext
  </li>
</ui:template>

<ui:dialog name="sample" start-page="sample_page">
  <ui:enumeration name="yesno">
    <ui:enum internal="yes" external="Yes"/>
    <ui:enum internal="no"  external="No"/>
  </ui:enumeration>

  <ui:variable name="user_answer" type="yesno">
    <ui:enum-value>
      <ui:enum-item internal="yes"/>
    </ui:enum-value>
  </ui:variable>

  <ui:page name="sample_page">
    <html>
      <body>
        What is your answer?
        <ul>
          <ui:enumerate template="list_item"
                        type="yesno"/>
        </ul>
        <ui:button name="ok" label="OK"/>
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

ui:richbutton

Web Path: WDialog / Reference / The UI language / ui:richbutton

57 The element ui:richbutton

This element displays an HTML4-style richly rendered button. The generated HTML code consists of an `BUTTON` element with `TYPE=SUBMIT`, whose `name` attribute is set to a special identifier which is recognized by the system when the form is submitted.

When the user clicks on the button, a `Button` event is generated (unless the `index` attribute is specified; see below); the `handle` callback method of the dialog object can check whether the current event is the event associated with this button, and the method can execute code depending on the result of this check. For a description of possible events see *Events* (→ 41). The way events are generated is very similar to the *ui:button* (→ 87) element. Nevertheless, here is an example:

```
<ui:dialog name="sample" start-page="p1">
  <ui:page name="p1">
    <html>
      <body>
        <h1>Button test</h1>
        This is a <ui:richbutton name="b"><i>Button</i></ui:richbutton>
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

Here, the button event has the name "b". In order to check whether this event occured in the `handle` method, the following piece of code is recommended. O'Caml:

```
method handle =
  match self # event with
    Button "b" ->
      ... (* Do whatever you want to do *)
  | ... (* other cases *)
```

- Perl:

```
sub handle {
    my ($self) = @_;
    my ($e, $name) = $self->event;
    if ($e eq 'BUTTON' && $name eq 'b') {
        ... # Do whatever you want to do
    } elsif ... # other cases
    ;
```

```
    return undef;
}
```

If the `ui:richbutton` element has the `index` attribute, the button is identified by the pair `(name,index)`. When the user clicks on such an indexed button, an `Indexed_button` event is generated. The index value can be used to distinguish between several instances of buttons of the same type.

57.1 Declaration

Level: Generative element

```
<!ELEMENT ui:richbutton ANY>
<!ATTLIST ui:richbutton
          name  NMTOKEN     #REQUIRED
          index CDATA       #IMPLIED
          goto  NMTOKEN     #IMPLIED
          cgi   (auto|keep) "auto"
>
```

Additionally, `ui:richbutton` must only occur inside `ui:form`.

57.2 Attributes

The following attributes have a special meaning:

- `name`: Specifies the name of the button.
- `index`: Specifies the index value of the button. If this attribute is present, the button becomes an indexed button; otherwise the button is a plain button.
- `goto`: Specifies which page is the next page if the button is pressed. The variable containing the next page is initialized with the name specified here before the `handle` method is invoked. This means that the action of the button is to go to this page, unless the action is overridden in the `handle` method.
- `cgi`: The value "auto" means that the name of the CGI variable associated with the button is selected automatically. This works perfectly unless you want to refer to this variable from Javascript or from some other manually written event decoder. The value "keep" causes that the name of the CGI variable is predictable: it is `button_` concatenated with the name of the button. However, it is not allowed to specify "keep" if there is also an index value. Furthermore, the button name should only contain alphanumeric characters, because not all punctuation characters can be safely transported over the CGI protocol.

If there are any other attributes, these are added to the generated `BUTTON` HTML element. This means that especially the `onclick` attribute may be specified.

57.3 Sub elements

The `ui:richbutton` elements may contain any HTML code, or any UI language code that expands to HTML code. The inner elements are rendered as the surface of the button.

57.4 Generated HTML code

The `ui:richbutton` element generates HTML code which roughly looks as follows:

```
<button type="SUBMIT" name="..." value="...">
  inner elements
</button>
```

57.5 Known Problems

As the underlying `BUTTON` element is a recent addition to HTML, not every browser supports it (well). Problems include:

- The `BUTTON` element is not recognized at all (example: Netscape 4 browsers)
- The `BUTTON` element is rendered but the wrong events are generated (example: all Internet Explorers)

Nevertheless, many browsers support this element very well (e.g. Mozilla, Opera, Lynx), and it is only a matter of time until this element can be recommended for web sites. Until then, I would not use it unless there is some strategy how to avoid the problems. For example, server-side browser sniffing can be used to detect whether the element is supported, and if so, a better-looking HTML page is generated by using `ui:richbutton` instead of `ui:button`.

ui:select

Web Path: WDialog / Reference / The UI language / ui:select

58 The element ui:select

This element displays a selection box offering the user a number of choices, and the user can select one or multiple items of the list. The generated HTML code consists of a `SELECT` element, whose name attribute is set to a special identifier which is recognized by the system when the form is submitted.

It is important to distinguish between two sets of values: The *base set* contains all items of the list, whereas the *active set* is the smaller set enumerating only the selected items of the list.

There are three ways to specify these sets:

- One can bind the selection list to a declared enumerator variable. In this case, the base set are all values declared in the *ui:enumeration* (→ 111), and the active set are the current contents of the variable.
- One can explicitly specify an enumerator variable in the `base` attribute; the current contents of this variable will be taken as base set. In this case, it is possible to bind the selection list to a dynamic enumerator (for which no declaration exists that could set the base set implicitly). The active set are the current contents of this variable.
- Alternatively, the selection list can also be bound to a string variable when `base` is an enumerator. This is very useful for selections of the type 1 of N. The string variable contains the currently selected item of the base set, i.e. the active set has exactly one element.

As all input elements, the selection list must be bound to a variable (specified by the `variable` attribute). When the page is displayed, the base set determines the items of the selection list, and the current contents of the variable = the active set determines which of the items are selected. When the page is submitted, the active set is written back to the variable. Note that it is not possible to modify the base set by user interactions, and so there is no mechanism that writes such modifications back to a base set variable (if there is any). This means that (currently[7]) only the active set of the selection list is tied to a variable.

58.1 Declaration

Level: Generative

```
<!ELEMENT ui:select EMPTY>
<!ATTLIST ui:select
          variable  NMTOKEN      #REQUIRED
          index     CDATA        #IMPLIED
          base      NMTOKEN      #IMPLIED
          baseindex CDATA        #IMPLIED
          multiple  (yes|no)     "no"
          size      CDATA        #IMPLIED
          cgi       (auto|keep)  "auto"
```

[7] This is a restriction of HTML. I can imagine a selection list to which the user can add missing items, and I think such an extension would be practical in many situations. Perhaps the WWW Consortium adds this feature some day.

```
>
```

Additionally, `ui:select` must only occur inside `ui:form`.

58.2 Attributes

The following attributes have a special meaning:

- `variable`: Specifies the variable to which the active set of the selection list is tied. This must be a declared or dynamic enumerator variable, or a string variable. If the `index` attribute is also specified, this variable must be an associative enumerator, or an associative string.
- `index`: Specifies the index value of the element of the associative variable to which the active set is tied.
- `base`: Specifies the variable determining the base set of the selection list. This must be a declared or dynamic enumerator variable. If the `baseindex` attribute is also specified, this variable must be an associative enumerator. If the `base` attribute is omitted, and the variable specified by `variable` is a declared enumerator, the set of all declared values of this enumerator will be used as base set. It is an error to omit this attribute if the variable specified by `variable` is not a declared enumerator.
- `baseindex`: Specifies the index value of the element of the associative variable specified by the `base` attribute.
- `multiple`: Specifies whether the selection list allows multiple selections ("yes") or not ("no").
- `size`: Specifies the size of the visual layout of the selection list, i.e. the number of items that can be manipulated without scrolling. If omitted, and if multiple="no", the selection list will be usually rendered as drop-down menu.
- `cgi`: The value "auto" means that the name of the CGI variable associated with the selection list is determined automatically. This works perfectly unless you want to refer to this variable from Javascript or from some other manually written event decoder. The value "keep" causes that the name of the CGI variable is predictable: it is `var_` concatenated with the name of the variable (of the active set). However, it is not allowed to specify "keep" if there is also an index value. Furthermore, the variable name should only contain alphanumeric characters, because not all punctuation characters can be safely transported over the CGI protocol.

If there are any other attributes, these are added to the generated `SELECT` HTML element. This means that especially `onblur`, `onchange`, `onfocus`, and `onselect` may be specified.

58.3 Sub elements

The `ui:select` element does not have sub elements.

58.4 Generated HTML code

The `ui:select` element generates HTML code which roughly looks as follows:

```
<select name="..." ...>
  <option value="..." [selected]>...
  ...
</select>
```

58.5 Example 1: Usage with a declared enumerator

Here, the base set is { "0", "1" }, i.e. simply the values declared for the enumeration `bool`. This means that the selection list will offer the values "0" and "1" (but the user will see "false" and "true" because the external values are the visible ones). The active set is tied to the variable `b`, and `b` is initialized with { "0" }. When the page is first displayed, the selection list will show "false". Any changes resulting from user interaction will be written back to `b`; i.e. if the user selects "true", the active set becomes { "1" }, and if he goes back to "false", the active set will again be { "0" }.

```
<ui:dialog name="sample" start-page="sample_page">
  <ui:enumeration name="bool">
    <ui:enum internal="0" external="false"/>
    <ui:enum internal="1" external="true"/>
  </ui:enumeration>

  <ui:variable name="b" type="bool">
    <ui:enum-value>
      <ui:enum-item internal="0">
  </ui:variable>

  <ui:page name="sample_page">
    <html>
      <body>
        Please select your boolean value:
        <ui:select variable="b"/>
        <ui:button name="ok" label="OK"/>
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

58.6 Example 2: Usage with a dynamic enumerator

In this example, the base set is considered dynamic, for example it might be initialized from a database. However, the following fragment simply sets the base set `candidates` to a fixed list of candidates; I hope you can imagine that this could also be done by additional code in a really dynamic way. Consequently, the active set must be considered as dynamic, too, because the active set is always a subset of the base set. The active set `vote` is empty at the beginning, and the selection list will show only unselected items. After the user has clicked "OK", the selection will be written back to `vote`, which will be either empty or contain one candidate name.

```
<ui:dialog name="sample" start-page="sample_page">
  <ui:variable name="candidates" type="dynamic-enumerator">
    <ui:dyn-enum-value>
      <ui:dyn-enum-item internal="1234" external="Smith, Joe"/>
      <ui:dyn-enum-item internal="763"  external="Jackson, Dave"/>
      <ui:dyn-enum-item internal="128"  external="Miller, Jack"/>
    </ui:dyn-enum-value>
  </ui:variable>

  <ui:variable name="vote" type="dynamic-enumerator"/>
```

```
  <ui:page name="sample_page">
    <html>
      <body>
        Please vote for your favourite candidate:
        <ui:select variable="vote" base="candidates" size="3"/>
        <ui:button name="ok" label="OK"/>
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

58.7 Example 3: Usage with a string

The task is the same as in example 2. As you can only select one of the candidates, it is also possible to declare `vote` as string. This string should be initialized to one of the possible values, otherwise it is left to the browser (and unspecified) to initialize the dropdown list.

```
<ui:dialog name="sample" start-page="sample_page">
  <ui:variable name="candidates" type="dynamic-enumerator">
    <ui:dyn-enum-value>
      <ui:dyn-enum-item internal="1234" external="Smith, Joe"/>
      <ui:dyn-enum-item internal="763"  external="Jackson, Dave"/>
      <ui:dyn-enum-item internal="128"  external="Miller, Jack"/>
    </ui:dyn-enum-value>
  </ui:variable>

  <ui:variable name="vote" type="string">
    <ui:string-value>763</ui:string-value>
  </ui:variable>

  <ui:page name="sample_page">
    <html>
      <body>
        Please vote for your favourite candidate:
        <ui:select variable="vote" base="candidates" size="3"/>
        <ui:button name="ok" label="OK"/>
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

ui:server-popup

Web Path: WDialog / Reference / The UI language / ui:server-popup

59 The element ui:server-popup

This element generates a Javascript function that opens another page as popup window. The contents of this page are generated after the function has been called, and it is even possible to intercept the popup request from O'Caml/Perl because a `Popup_request` event is triggered just before page generation. There is also a simpler variant of popup windows that does not go to the server to get the popup page: *ui:popup* (→ 141).

The generated Javascript function has the name `popup_<page>` where `<page>` is the identifier found in the `page` attribute (see declaration below). The function takes two string arguments. The first argument is the window option list known from the Javascript `window.open` function; it specifies the visual properties of the new window. The second argument is an arbitrary string that is attached to the `Popup_request` event. By calling the generated function, the window pops up and requests the referenced page from the server. For example, the following button shows the page `sample` in a new popup window with the specified width and height, and attaches the string `fourty-two` to the popup request event:

```
<ui:popup page="sample"/>
<input type="button" onclick="open_sample('width=100,height=100', 'forty-two')"/>
```

The page, here `sample`, must set the attribute `popup` to `yes`:

```
<ui:page name="sample" popup="yes">
  ...
</ui:page>
```

This is necessary because the generated HTML code is slightly different from that of normal pages.

After the Javascript function has been called, the following actions happen in turn:

- The current dialog object is copied, and the duplicate is used to generate the popup window. Note that the dialog variables are not updated from the interactors, i.e. the copied dialog has the same state as the original had when the main window was generated.
- The `Popup_request` event is sent to the copied dialog, and it can be caught from the `handle` method. It is allowed that the `handle` method changes the name of the popup page to be displayed (by raising `Change_page`, or calling `set_next_page`). It is not allowd that the `handle` method changes the dialog object, though.
- Of course, the `prepare_page` method is invoked, too.
- Finally, the popup page is generated.

Keep in mind that the dialog object is copied for the time of the popup generation. If the main window is submitted, the original object will continue to be the active object, and the duplicate is silently dropped. If the popup window is submitted, however, the original object will be dropped, and the copy will be used instead for all future interactions!

The popup window can include interactors like text boxes, selection lists, etc., as well as buttons and hyperlinks. If the popup window is submitted, the variables of the duplicated dialog are updated from the interactors of the popup page, the popup window is closed, the main window is discarded (!), the trigerred event is sent to the dialog, and the resulting page will be displayed in the main window using the copied dialog object as current object. Note that these actions are very different from what happens for submitted popup windows generated by `ui:popup`. Especially, the interactors of the main page are not used to update any dialog variables.

If the main window is submitted when there is an open popup window, the popup window is closed, and the underlying duplicated dialog object is discarded.

It is only possible to have one open popup window at the same time. Trials to open more than one popup window are silently ignored.

The popup window is automatically closed when any submit button or hyperlink of the main window is pressed, or when the page currently displayed in the main window is left by other means. In these cases, the user interactions in the popup window are ignored, and the copied dialog object is dropped.

It is possible to change this behavior by additional Javascript statements:

- You can close the popup window by calling `window.close()` from the popup window, or by calling `close_popup()` from the main window. The user interactions in the popup window will be ignored.
- You can force submission of the popup window by calling `uiform_submit()` from the popup window.
- You can lock the submission of the main window while a popup window is open by setting the `ONSUBMIT` handler of the `ui:form` element of the main window to `return lock_popup()`

Note that WDialog generates an `ONUNLOAD` handler for the main window and a `ONSUBMIT` handler for the popup window.

59.1 Declaration

Level: Generative element

```
<!ELEMENT ui:server-popup EMPTY>

<!ATTLIST ui:server-popup
          page  NMTOKEN  #REQUIRED>
```

It is required that there is a `ui:form` in the current page; however, the `ui:server-popup` element can occur outside the `ui:form` element.

59.2 Attributes

- `page:` The name of the page to display in the popup window. It is required that the `popup` attribute of this page is set to "yes". The Javascript function gets the name `open_` plus the name of the opened page, e.g. `open_menu` if the page is called `menu`.

ui:special

Web Path: WDialog / Reference / The UI language / ui:special

60 The element ui:special

The inner nodes of the `ui:special` element are expanded without applying the default output encoding. Normally, every character data node is HTML-encoded before included into the output stream such that the characters <, >, &, and " are properly quoted. Inside `ui:special` this quoting is turned off.

60.1 Declaration

Level: Control structure

```
<!ELEMENT ui:special ANY>
```

60.2 Sub elements

All page body elements are allowed.

60.3 Example

```
<ui:page name="sample">
  <ui:special>This outputs &amp;nbsp;, a non-breakable space.</ui:special>
</ui:page>
```

Note that we have to write `&` because the XML parser requires this. This page generates the string "This outputs , a non-breakable space." Without the `ui:special` element, the page would be expanded to "This outputs , a non-breakable space."

ui:string-value

Web Path: WDialog / Reference / The UI language / ui:string-value

61 The element ui:string-value

This element represents a string literal that can be used to set the initial value of a *ui:variable* (→ 170).

61.1 Declaration

Level: Dialog structure

```
<!ELEMENT ui:string-value (#PCDATA)* >
```

61.2 Example

```
<ui:variable name="sample" type="string">
  <ui:string-value>the initial value</ui:string-value>
</ui:variable>
```

ui:template

Web Path: WDialog / Reference / The UI language / ui:template

62 The element ui:template

This element defines a template. For an overview, see the chapter about *Templates* (→ 49).

62.1 Declaration

Level: Control structure

```
<!ELEMENT ui:template ANY>

<!ATTLIST ui:template
          name               NMTOKEN  #REQUIRED
          from-caller        NMTOKENS #IMPLIED
          from-context       NMTOKENS #IMPLIED
          xml:lang           NMTOKEN  #IMPLIED
>
```

The subelements of `ui:template` must match the informal rule `( ui:default*, %page-body;* )` where `%page-body;` stands symbolically for all allowed sub elements. Note that whitespace between the `%page-body;` elements counts, but at the other positions it is ignored.

62.2 Attributes

- `name`: The name of the template. Template names are globally known.
- `from-caller`: The space-separated list of parameters with lexical scope.
- `from-context`: The space-separated list of parameters with dynamic scope.
- `xml:lang`: If present, this attribute defines the language suffix of the template name.

62.3 Sub elements

The sub element *ui:default* (→ 97) has the special task to define the default values for parameters that are used if the parameter has not been passed by the caller (lexical scope), or the parameter cannot be found in the context (dynamic scope). `ui:default` elements must only occur at the beginning of the template.

The other sub elements may be arbitrary page body elements.

62.4 Internationalization

If no `xml:lang` attribute exists, the template is defined with exactly the name as specified by the `name` attribute.

If there is a `xml:lang` attribute, the template gets a compound name. The `name` attribute is the first part, and the `xml:lang` attribute is the suffix; the parts are separated by a `#` character. For example, the template

```
<ui:template name="foo" xml:lang="en">...</ui:template>
```

defines the template `foo#en`.

62.5 Example

See the chapter *Templates* (→ 49).

ui:text and ui:password

Web Path: WDialog / Reference / The UI language / ui:text and ui:password

63 The elements ui:text and ui:password

The element `ui:text` displays a one-line text input box. The element `ui:password` displays a password input box. The difference is that the contents of the password box are invisible (only a string of asterisks). - The generated HTML code consists of an `INPUT` element with `TYPE=TEXT` or `TYPE=PASSWORD`, whose name attribute is set to a special identifier which is recognized by the system when the form is submitted.

The text or password box must be tied to a string variable. The contents of the widget is initialized with the current contents of the variable when the page is displayed. Conversely, the contents of the widget are transferred back to the variable when the page is submitted.

In the following example, the user can enter a file name. The variable `file_name` is initialized with the value "sample.txt", and because this is the intial value of the variable, this string will also be the initial value of the text widget.

```
<ui:dialog name="sample" page="sample_page">
  <ui:variable name="file_name" type="string">
    <ui:string-value>sample.txt</ui:string-value>
  </ui:variable>

  <ui:page name="sample_page">
    <html>
      <body>
        Please input the file name:
        <ui:text variable="file_name"/>
        <ui:button name="ok" label="OK"/>
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

63.1 Declaration

Level: Generative

```
<!ELEMENT ui:text EMPTY>
<!ELEMENT ui:password EMPTY>

<!ATTLIST ui:text
          variable  NMTOKEN     #REQUIRED
          index     CDATA       #IMPLIED
          maxlength CDATA       #IMPLIED
          size      CDATA       #IMPLIED
```

```
          cgi       (auto|keep) "auto"
>
<!ATTLIST ui:password
          variable  NMTOKEN     #REQUIRED
          index     CDATA       #IMPLIED
          maxlength CDATA       #IMPLIED
          size      CDATA       #IMPLIED
          cgi       (auto|keep) "auto"
>
```

Additionally, `ui:text` and `ui:password` must only occur inside `ui:form`.

63.2 Attributes

The following attributes have a special meaning:

- `variable`: Specifies the variable of the current dialog object to which the text/password box is tied. Unless the `index` attribute is present, the variable must have string type. If there is an `index` attribute, the variable must be an associative list of strings.
- `index`: Specifies the index value of the element of the associative variable to which the text/password box is tied.
- `maxlength`: Specifies the maximum number of characters the input box can accepts.
- `size`: Specifies the length of the input box in characters.
- `cgi`: The value "auto" means that the name of the CGI variable associated with the text box is selected automatically. This works perfectly unless you want to refer to this variable from Javascript or from some other manually written event decoder. The value "keep" causes that the name of the CGI variable is predictable: it is `var_` concatenated with the name of the variable. However, it is not allowed to specify "keep" if there is also an index value. Furthermore, the variable name should only contain alphanumeric characters, because not all punctuation characters can be safely transported over the CGI protocol.

If there are any other attributes, these are added to the generated `INPUT` HTML element. This means that especially `onblur`, `onchange`, `onfocus`, and `onselect` may be specified.

63.3 Sub elements

Neither `ui:text` nor `ui:password` have sub elements.

63.4 Generated HTML code

The `ui:text` element generates HTML code which roughly looks as follows:

```
<input type="TEXT" name="..." value="..." maxlength="..." size="...">
```

The `ui:password` element generates something like that:

```
<input type="PASSWORD" name="..." value="..." maxlength="..." size="...">
```

ui:textarea

Web Path: WDialog / Reference / The UI language / ui:textarea

64 The element ui:textarea

The element `ui:textarea` displays a multi-line text input box. The generated HTML code consists of a `TEXTAREA` element, whose name attribute is set to a special identifier which is recognized by the system when the form is submitted.

The text box must be tied to a string variable. The contents of the widget is initialized with the current contents of the variable when the page is displayed. Conversely, the contents of the widget are transferred back to the variable when the page is submitted.

Input boxes generated by ui:textarea are very similar to the boxes generated by *ui:text* (→ 160); the only major difference is that the boxes accept multi-line texts.

64.1 Declaration

Level: Generative

```
<!ELEMENT ui:textarea EMPTY>
<!ATTLIST ui:textarea
          variable  NMTOKEN          #REQUIRED
          index     CDATA            #IMPLIED
          rows      CDATA            #IMPLIED
          cols      CDATA            #IMPLIED
          wrap      (off|hard|soft)  "off"
          cgi       (auto|keep)      "auto"
>
```

Additionally, `ui:textarea` must only occur inside `ui:form`.

64.2 Attributes

The following attributes have a special meaning:

- `variable`: Specifies the variable of the current dialog object to which the text box is tied. Unless the `index` attribute is present, the variable must have string type. If there is an `index` attribute, the variable must be an associative list of strings.
- `index`: Specifies the index value of the element of the associative variable to which the text box is tied.
- `rows`: Specifies the number of rows the input box displays. Note that this affects only the visual layout of the box, and does not limit the number of lines the user can enter.
- `cols`: Specifies the number of columns the input box displays. Note that this affects only the visual layout of the box, and does not limit the number of columns the user can enter.

- `wrap`: Specifies the wrapping mode of the box. See HTML documentation for the meaning of the modes.
- `cgi`: The value "auto" means that the name of the CGI variable associated with the text box is selected automatically. This works perfectly unless you want to refer to this variable from Javascript or from some other manually written event decoder. The value "keep" causes that the name of the CGI variable is predictable: it is `var_` concatenated with the name of the variable. However, it is not allowed to specify "keep" if there is also an index value. Furthermore, the variable name should only contain alphanumeric characters, because not all punctuation characters can be safely transported over the CGI protocol.

If there are any other attributes, these are added to the generated `TEXTAREA` HTML element. This means that especially `onblur`, `onchange`, `onfocus`, and `onselect` may be specified.

64.3 Sub elements

`ui:textarea` has no subelements (unlike the corresponding HTML element).

64.4 Generated HTML code

The `ui:textarea` element generates HTML code which roughly looks as follows:

```
<textarea name="..." rows="..." cols="..." wrap="...">
...
</textarea>
```

ui:translate

Web Path: WDialog / Reference / The UI language / ui:translate

65 The element ui:translate

This element is replaced in the generated HTML output by the corresponding external value of the specified internal value of a declared enumerator.

The external value is HTML-quoted before being processed; i.e. the characters <, >, &, and " are substituted by the corresponding HTML entities <, >, &, and ", respectively.

65.1 Declaration

Level: Generative

```
<!ELEMENT ui:translate EMPTY>
<!ATTLIST ui:translate
          type     NMTOKEN #REQUIRED
          internal NMTOKEN #REQUIRED>
```

65.2 Attributes

- `type`: Specifies the name of a declared enumerator type (see *ui:enumeration* (→ 111)).
- `internal`: Selects the internal value of the declared enumerator type.

65.3 Sub elements

`ui:translate` does not have sub elements.

65.4 Example

The following piece of code simply displays "Two":

```
<ui:dialog name="sample" start-page="sample_page">
  <ui:enumeration name="smallnumbers">
    <ui:enum internal="1" external="One"/>
    <ui:enum internal="2" external="Two"/>
    <ui:enum internal="3" external="Three"/>
  </ui:enumeration>

  <ui:page name="sample_page">
```

```
    <html>
      <body>
        <ui:translate type="smallnumbers" internal="2"/>
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

ui:true

Web Path: WDialog / Reference / The UI language / ui:true

66 The element ui:true

This element expands simply its inner nodes without transforming them. Furthermore, this element sets the condition code to `true`. The condition code can be evaluated by *ui:cond* (→ 94).

66.1 Declaration

Level: Control structure

```
<!ELEMENT ui:true ANY>
```

66.2 Sub elements

This element may contain all page body elements.

66.3 Example

```
<ui:cond>
  <ui:if value1="$x" value2="$[v]">
     ...do this...
  </ui:if>
  <ui:if value1="$x" value2="$[w]">
     ...do that...
  </ui:if>
  <ui:true>
     ...else do this...
  </ui:true>
</ui:cond>
```

Because `ui:true` sets the condition code always to `true`, it can be used as default branch in `ui:cond`. Its contents will be expanded if all the previous conditions happen to be false.

ui:use

Web Path: WDialog / Reference / The UI language / ui:use

67 The element ui:use

This element instantiates a template. For an overview, see the chapter about *Templates* (→ 49).

67.1 Declaration

Level: Control structure

```
<!ELEMENT ui:use ( ui:param )* >

<!ATTLIST ui:use
          template NMTOKEN #REQUIRED>
```

67.2 Attributes

- `template`: Names the template to instantiate. Note that the current language of the dialog may also influence which template is selected (see below).

67.3 Sub elements

The *ui:param* (→ 140) elements define the actual values for the lexical parameters

67.4 Internationalization

If a certain language is selected for the dialog, this also affects the template system. In particular, it is first checked if the used template is defined for this language, and if so, this version of the template will be used. Otherwise, it is checked whether there is a template without `xml:lang` attribute, and if it can be found, this version will be used.

For more information, see the chapter about *Internationalization* (→ 67).

67.5 The t namespace

Because `ui:use` is a quite long notation, there are two ways to abbreviate it. Instead of

```
<ui:use template="x">
  <ui:param name="p1">t1</ui:param>
  ...
  <ui:param name="pN">tN</ui:param>
```

```
</ui:use>
```

you can also write

```
<t:x>
  <p:p1>t1</p:p1>
  ...
  <p:pN>tN</p:pN>
</t:x>
```

Furthermore, the parameters can also be passed by attributes if they only consist of unstructured text:

```
<t:x p1="t1" ... pK="tK">
  <p:pJ>tJ</p:pJ>
  ...
  <p:pN>tN</p:pN>
</t:x>
```

67.6 The q namespace

The other way to abbreviate `ui:use` is the `q` namespace. Instead of writing

```
<ui:use template="x">
  <ui:param name="p1">t1</ui:param>
  ...
  <ui:param name="pN">tN</ui:param>
  <ui:param name="body">tBODY</ui:param>
</ui:use>
```

(note the fixed name `body`) it is also possible to call the template by:

```
<q:x p1="t1" ... pK="tK">
  tBODY
</q:x>
```

67.7 Example

See *Templates* (→ 49).

ui:variable

Web Path: WDialog / Reference / The UI language / ui:variable

68 The element ui:variable

`ui:variable` defines an instance variable for the current dialog object. The variable has a name, a type, and an initial value (the value the variable is automatically set to when the object is created). The variable is assignable ("mutable" in O'Caml terminology).

It is possible to refer to the current value of variables from the following places:

- *Before the HTML page is generated:* In the `prepare_page` callback method of the current dialog object one can get and set the value of variables by the inherited methods `variable` and `set_variable` (and some derivates of these); see *Dialogs* (→ 22).
- *While the HTML page is being generated:* There are lots of elements which can refer to variables. Especially, *ui:dynamic* (→ 102) allows it to insert the current value of a string value at the current generation position. The interactor elements such as *ui:text* (→ 160) are tied to variables, i.e. they reflect the current value of variables, and when the user changes the value of the interactor, the value of the variable will be changed, too.
- *When the user event is evaluated:* After the user has clicked on a button or hyperlink, one can again get and set the values of variables by calling the methods `variable` and `set_variable` in the `handle` callback.

Unless the value is explicitly modified, variables retain their values across page changes (like many other properties of dialog objects).

For an introduction into the type system, see *Data types* (→ 27).

68.1 Declaration

Level: Dialog structure

```
<!ENTITY % value-literal "(ui:string-value | ui:enum-value | ui:dyn-enum-value |
                           ui:alist-value)" >

<!ELEMENT ui:variable ( %value-literal; )? >

<!ATTLIST ui:variable
          name        NMTOKEN   #REQUIRED
          type        NMTOKEN   "string"
          associative (yes|no)  "no"
          temporary   (yes|no)  "no"
          protected   (yes|no)  "no"
>
```

There are some restrictions concerning the possible sub elements of `ui:variable`, see below.

68.2 Attributes

- `name`: Specifies the name of the variable
- `type`: Specifies the base type of the variable: Either `string`, `dialog`, `dynamic-enumerator`, or the name of a declared enumerator (see *ui:enumeration* (→ 111)). The default is "string".
- `associative`: Whether the variable is associative or not; the default is that the variable is not associative.
- `temporary`: Whether the variable is temporary or not. A temporary variable is reset to its initial value during a page change (unless the variable is tied to an interactor). This means that you can still set the variable in the `prepare_page` method and refer to it from elements; but in the following `handle` invocation the variable will be found reset to its initial value. The intention of this option is to avoid that large variables are transported from one page to another in hidden HTML fields although the variables are only needed to generate dynamic contents. Note that interactor variables (i.e. variables tied to an interactor element) are already transported by the interactor, and so the `temporary` declaration has no effect.

  The default is that variables are not temporary.
- `protected`: A protected variable cannot be changed by CGI arguments. Sometimes it is necessary to prevent the users from changing variables because they contain trusted data. The recommended method to achieve this requires two steps: First, the variables must be declared with `protected="yes"`. Second, a session manager must be used that protects the state from user manipulations (note that the session manager that is enabled by default does not provide such protection).

68.3 Sub elements

The sub element of the `ui:variable` contains the initial value of the variable. The initial value is the value the variable is initialized with when the containing dialog object is created. If no initial value is specified, the following values apply:

- For non-associative string variables: The empty string
- For non-associative enumerator variables: The empty enumerator (empty set)
- For non-associative dynamic enumerator variables: The empty enumerator (empty set)
- For non-associative dialog variables: The value `None`
- For associative variables: The empty associative list

If an initial value is specified, the sub element defining the value must correspond to the type of the variable:

- For non-associative string variables: The default value must be defined by *ui:string-value* (→ 157).
- For non-associative enumerator variables: The default value must be defined by *ui:enum-value* (→ 107).
- For non-associative dynamic enumerator variables: The default value must be defined by *ui:dyn-enum-value* (→ 101).
- For non-associative dialog variables: It is not possible to define default values for dialog variables.
- For associative variables: The default value must be defined by *ui:alist-value* (→ 84).

68.4 Example

```
<ui:dialog name="sample" ...>
  <ui:variable name="person">
    <ui:string-value>Peter</ui:string-value>
  </ui:variable>
  ...
  <ui:page name="sample">
    <html>
      <body>
        My name is <ui:dynamic variable="person"/>.
      </body>
    </html>
  </ui:page>
</ui:dialog>
```

t:*, q:*, and p:*

Web Path: WDialog / Reference / The UI language / t:, q:*, and p:**

69 The namespaces t, q, and p

Elements like `t:name`, `q:name`, and `p:name` (where `name` is an arbitrary identifier) can be used as abbreviations for the *ui:use* (→ 168) and *ui:param* (→ 140) elements. See there for explanations and examples.

l:*

*Web Path: WDialog / Reference / The UI language / l:**

70 The namespace l

Elements like `l:name` (where `name` is an arbitrary identifier) can be used as abbreviations for *ui:iflang* (→ 122) elements. See there for explanations and examples.

$param

Web Path: WDialog / Reference / The UI language / $param

71 Template parameters $param

Template parameters can be written as `$param` (i.e. an identifier prefixed by a dollar character), or as `${param}` (i.e. additional braces). There is also the variant `${param/enc}` containing an additional output encoding. All these parameters must be imported into the current template or page definition by `from-caller` or `from-context` attributes, see the elements *ui:template* (→ 158) and *ui:page* (→ 135). For an overview how template parameters work see the chapter about *Templates* (→ 49).

The parameters *$int* and *$ext* play a special role in the context of iterations. They need not to be imported into the current context, as they are automatically set for every iteration. See the elements *ui:iterate* (→ 132) and *ui:enumerate* (→ 108) for details.

$[expr]

Web Path: WDialog / Reference / The UI language / $[expr]

72 Bracket expressions $[...]

Bracket expressions allow the UI developer to access dialog variables from page or template definitions (i.e. everywhere the dollar character is recognized as meta symbol). For example, `$[v]` would expand to the current value of the dialog variable `v`. Of course, this works only if this variable is a string variable, as it would be unclear what to do with a non-string value (e.g. an enumeration).

So far, so simple. In recent versions of WDialog, the brackets cannot only contain variable names but whole expressions that are computed when the bracket is expanded. For instance, the expression `$[card(v)]` expands to the number of words in the string `v`. There are a number of functions that can be applied to variables (see below). The syntax of function calls always includes parantheses, so `x` is a variable, and `x()` is a function call. Functions may have several arguments, separated by commas, e.g. `f(x1,x2,x3)`.

Of course, the card function is not only reasonable for strings. We can also define the cardinality of enumerator values, and of associative lists. Because of this it is allowed to write `$[card(v)]` when `v` is an enumerator or alist. In general, the intermediate values during evaluation may be of any type that can be stored in a dialog variable; however the final value must be a string as the result is inserted into the XML tree[8]. Typing is dynamic, and although the functions usually only accept certain types as arguments it is not tried to verify that by a type checker.

It is not only possible to access variables, but also template parameters. The dollar character must prefix the parameter name, e.g. `$[$p]`. Braces are allowed, and even output encodings can be specified: `$[${p}]` and `$[${p/html}]` are legal expressions. The parameters are evaluated to strings (as if they would occur in attribute context). A number of UI control elements are expanded allowing one to mix template-level and dialog-level evaluation strategies. For example, the template `t2` expands to the *size* of the expanded template `t1`:

```
<ui:template name="t1">
  This is a text
</ui:template>

<ui:template name="t2" from-caller="which">
  <ui:default param="which"><ui:use template="t1"/></ui:default>
  $[card($which)]
</ui:template>
```

The expanded text is `4`, the number of words in `t1`.

We have mentioned variables, functions, parameters. Is it possible to include constant values into bracket expressions? Yes, but the current implementation is quite limited. Numbers can be used literally, e.g. `$[add(n,2)]`. It is also possible to construct strings of words using the `words` form, e.g. `$[words(a,b,c)]` is the string "a b c". The words must be syntactically names. It is not (yet) possible to include arbitrary string constants or any non-string constants. These may be added later.

[8] It may be possible to also allow XML tree types here, but such types do not occur in the rest of the UI language. This is an interesting idea for a future extension of the language, though.

Another important syntactic note: *Bracket expressions must not contain white space!* They are not even recognized by the parser if they do.

There are also some boolean functions. The value 0 is considered as the false value, and numbers other than 0 are considered as true values.

Now the list of functions that are defined by default:

- `id(expr)`: Just returns `expr` (identity)
- `length(str)`: Returns the number of characters of the string `str`
- `card(expr)`: If `expr` is a string, this function returns the number of words. If `expr` is an enumerator or associative list, this function returns the number of elements. This definition of cardinality is just the same as that of ui:iterate.
- `size(expr)`: *Deprecated!* If `expr` is a string, this function returns the number of characters. If `expr` is an enumerator or associative list, this function returns the number of elements.
- `add(num1,...,numN)`: Adds all the numbers passed as arguments and returns the sum.
- `sub(num1,...,numN)`: Subtracts the second and all following arguments from the first argument, and returns the difference.
- `mul(num1,...,numN)`: Multiplies all the numbers passed as arugments and returns the product.
- `div(num1,...,numN)`: Divides the first number through the second number and all following numbers (in turn), and returns the quotient.
- `modulo(num1,...,numN)`: Divides the first number through the second number and takes the modulus, and continues with the following numbers in turn, and returns the final modulus.
- `assoc(alist,str)`: Looks up the `str` argument in the associative list `alist`, and returns the value that corresponds to the `str` key. Of course, `str` must be a string. It is an error if `str` does not occur in `alist`.
- `nth(alist,num)`: Returns the `num`th value of the associative list `alist`, i.e. `num` is the ordinal number of the value to return. The first value has the ordinal number 0. It is an error if `num` is greater or equal than the number of elements of `alist`.
- `contains(container,str)`: Returns 1 if the first argument contains the second argument, and 0 otherwise. The second argument must be a string value. The function is defined as follows: If the `container` is a string, it is split up into a list of words, and it is tested whether `str` occurs in the list of words. If the `container` is a declared enumerator, it is tested whether the value includes `str` as item. If the `container` is a dynamic enumerator, it is tested whether the value includes `str` as internal item. If the `container` is an associative list, it is tested whether the value includes `str` as index.
- `mentions(container,str)`: Returns 1 if the first argument contains the second argument, and 0 otherwise. The second argument must be a string value. The function is defined as follows: If the `container` is a dynamic enumerator, it is tested whether the value includes `str` as external item. If the `container` is an associative list, it is tested whether the value includes `str` as mapped value.
- `translate(dynenum,str)`: Returns the external value that corresponds to the internal value `str` in the dynamic enumerator `dynenum`. It is an error if `str` does not occur as internal value in `dynenum`.
- `translate(enum(name),str)`: Returns the external value that corresponds to the internal value `str` in the declared enumerator type `name` (i.e. the name in a `ui:enumeration` declaration). It is an error if `str` does not occur as internal value in the enumeration. See below for explanations of the `enum` form.
- `rev-translate(dynenum,str)`: Returns the (first) internal value that corresponds to the external value `str` in the dynamic enumerator `dynenum`. It is an error if `str` does not occur as external value in `dynenum`.

- `rev-translate(enum(name),str)`: Returns the (first) internal value that corresponds to the external value `str` in the declared enumerator type `enum` (i.e. the name in a `ui:enumeration` declaration). It is an error if `str` does not occur as external value in the enumeration. See below for explanations of the `enum` form.
- `eq(str1,str2)`: Returns 1 if both strings are equal, and 0 otherwise
- `ne(str1,str2)`: Returns 1 if the strings are not equal, and 0 otherwise
- `match(str1,str2)`: Returns 1 if the string `str1` matches the regular expression `str2`, and 0 otherwise
- `nomatch(str1,str2)`: Returns 1 if the string `str1` does not match the regular expression `str2`, and 0 otherwise
- `substring(str,num1)`: Returns the substring of `str` starting at character position `num1` until the end of the string.
- `substring(str,num1,num2)`: Returns the substring of `str` starting at character position `num1` with length `num2` (the length can be negative).
- `concat(str1,...,strN)`: Concatenates the strings `str1` to `strN`.
- `int-eq(num1,num2)`: Returns 1 if both numbers are equal, and 0 otherwise
- `int-ne(num1,num2)`: Returns 1 if the numbers are not equal, and 0 otherwise
- `int-lt(num1,num2)`: Returns 1 if `num1` < `num2`, and 0 otherwise
- `int-le(num1,num2)`: Returns 1 if `num1` <= `num2`, and 0 otherwise
- `int-gt(num1,num2)`: Returns 1 if `num1` > `num2`, and 0 otherwise
- `int-ge(num1,num2)`: Returns 1 if `num1` >= `num2`, and 0 otherwise
- `int-min(num,...)`: Returns the minimum of all passed numbers
- `int-max(num,...)`: Returns the maximum of all passed numbers
- `int-abs(num)`: Returns the absolute value
- `int-sign(num)`: Returns the sign of the number
- `card-eq(container,num)`, `card-ne(container,num)`, `card-lt(container,num)`, `card-le(container,num)`, `card-gt(container,num)`, `card-ge(container,num)`: These functions compare the cardinality of `container` with the number. The cardinality is defined as for the `card` function.
- `height(str)`: Returns the number of lines the string `str` consists of. A line is separated from the next one by either LF, CR, or CRLF bytes. The number of lines is the number of these line separators plus 1.
- `width(str)`: Returns the number of characters the longest line in string `str` consists of. The line separators are not counted for the width.
- `dialog_exists(dlg)`: Returns 1 if the dialog exists, 0 otherwise
- `and(bool,...)`: Returns 1 if all arguments are non-zero integers, and 0 otherwise. This function is evaluated lazily.
- `or(bool,...)`: Returns 1 if there is a non-zero integers as argument, and 0 otherwise. This function is evaluated lazily.
- `not(bool)`: Returns 1 if the argument is 0, and 0 if the argument is non-zero.
- `true()`: Returns 1
- `false()`: Returns 0

- `if(bool,true_arg,false_arg)`: If `bool` is a non-zero number, the value of `true_arg` is returned. If 0, the value of `false_arg` is returned. This function is evaluated lazily.
- `var(str)`: Returns the contents of the dialog variable called `str`. (This function can be used to access variables indirectly.)
- `dialog()`: Returns the name of the current dialog.
- `self()`: Returns the current dialog.
- `page()`: Returns the name of the current page.
- `language()`: Returns the current language, or the empty string if none is selected.
- `self-base-url()`: Return the URL pointing to the current script (omitting any URL parameters)
- `session-id()`: Returns the session ID without checksum
- `create-anchor-event(str)`: Add the anchor event source called `str`, and return the HTML-level identifier.
- `create-xanchor-event(str1,str2)`: Add the anchor event source called `str1` for index `str2`, and return the HTML-level identifier.

Furthermore, there are the following special forms, i.e. syntactic elements evaluated in a special way:

- `type(var)`: Returns the name of the type of the variable `var`. The argument is not expanded before evaluation of the form, but taken literally, e.g. `type(x)` returns the type of the variable `x`, and not the type of the variable whose name is stored in the variable `x`. The return value is a string:
  - `"string"` is the type name of string variables
  - `"dialog"` is the type name of dialog variables
  - `"dynamic-enumerator"` is the type name of dynamic enumerator variables
  - The name of the enumeration is the type name of enumerator variables

  The type name is what the *ui:variable* (→ 170) element declares with the `type` argument.

  It does not matter whether the variable is associative or not.
- `is-associative(var)`: Returns whether the type of the variable `var` is associative or not. Like the `type` form, the argument is taken literally. The return value is either the string `"yes"` or `"no"`.
- `default(var)`: Returns the default value used to initialize the variable `var` when the dialog object is created. Like the `type` form, the argument is taken literally. The return value is a value of appropriate type.
- `enum(name)`: Returns the declaration of the enumeration `enum` as a dynamically enumerated value. The argument is taken literally.
- `words(name,...)`: Returns the string containing the names literally (i.e. the concatenated names separated by spaces). The arguments are taken literally.

Numbers are represented as decimal strings.

As strings can contain multi-byte characters, the question arises whether the string functions take the number of bytes or the number of characters as "position" or "length". Of course, characters are counted, so the user does not have to take care of the character encoding.

Dot notation (v1.v2)

Web Path: WDialog / Reference / The UI language / Dot notation (v1.v2)

73 The dot notation

Variables can be declared with type `dialog`, and because of this it is possible that a dialog stores a subdialog in a variable. The question arises how you can access subdialogs like in:

```
<ui:dialog name="calling_dialog" start-page="...">
  ...
  <ui:variable name="v" .../>
  ...
  <ui:page name="calling_page">
    ...
    As in many other dialogs, you can now go to our
    <ui:a name="call_event">special dialog</ui:a>.
    ...
  </ui:page>
</ui:dialog>

<ui:dialog name="called_dialog" start-page="called_page">
  ...
  <ui:variable name="caller" type="dialog"/>
  ...
  <ui:page name="called_page">
    ...
    You can now do ... this ... and ... that.
    <ui:a name="return_event">Return to previous dialog.</ui:a>
    ...
  </ui:page>
</ui:dialog>
```

(See the section about *Data types* (→ 27), it discusses the same example from another view.) Of course, when the calling dialog instantiates the called dialog, it stores itself into the dialog variable `caller`, so the previous dialog is preserved, and can be reactivated when the user wants to return. The variable `caller` is not only an abstract container for the original dialog, because it is possible to access the contents of the calling dialog from the called dialog by using the *dot notation*. For example, the variable `v` is available under the name `caller.v`, i.e. the name is prefixed by the name of the variable storing the dialog, separated by a dot. You can use the dot notation everywhere a variable is referenced, e.g.

```
<ui:dynamic variable="caller.v"/>
```

or just

```
${caller.v}
```

It is even possible to access variables from O'Caml (or Perl) by the dot notation:

```
let v = dlg # string_variable "caller.v" in
...
```

Note that a dialog variable is either empty (or `None` in O'Caml), or filled (`Some dlg` in O'Caml). Because of this, the "dot" access can cause an error condition when the dialog variable is empty.

The standard UI library

Web Path: WDialog / Reference / The standard UI library

74 The standard UI library

There are a number of predefined templates that can be used by application programmers.

- `wd-null`: This template expands to the empty string.
- `wd-int`: This template expands to the value of the parameter `$int`. It is intended to be used in iterators like ui:iterate:

  ```
  <ui:iterate template="wd-int" variable="x">
    <ui:iter-separator>, </ui:iter-separator>
  </ui:iterate>
  ```

  This example iterates over the contents of `x` (either a dynamic enumerator or an associative variable), and prints the internal values, separated by commas.
- `wd-ext`: This template expands to the value of the parameter `$ext`. It is intended to be used in iterators like ui:iterate:

  ```
  <ui:iterate template="wd-ext" variable="x">
    <ui:iter-separator>, </ui:iter-separator>
  </ui:iterate>
  ```

  This example iterates over the contents of `x` (either a dynamic enumerator or an associative variable), and prints the external values, separated by commas.
- `wd-doctype-html401-strict`: Expands to the DOCTYPE line for the strict HTML 4.01 DTD.
- `wd-doctype-html401-transitional`: Expands to the DOCTYPE line for the transitional HTML 4.01 DTD.
- `wd-doctype-html401-frameset`: Expands to the DOCTYPE line for the HTML 4.01 frameset DTD.
- `wd-doctype-html40-strict`: Expands to the DOCTYPE line for the strict HTML 4.0 DTD.
- `wd-doctype-html40-transitional`: Expands to the DOCTYPE line for the transitional HTML 4.0 DTD.
- `wd-doctype-html40-frameset`: Expands to the DOCTYPE line for the HTML 4.0 frameset DTD.
- `wd-doctype-html32`: Expands to the DOCTYPE line for the HTML 3.2 DTD.
- `wd-doctype-html20`: Expands to the DOCTYPE line for the HTML 2.0 DTD.

WDialog API (O’Caml)

Web Path: WDialog / Reference / WDialog API (O’Caml)

75 WDialog API for O’Caml

This part of the manual has been generated by Maxence Guesdon’s excellent documentation tool *ocamldoc*. You find the interesting part under *Modules* (→ 184).

Modules

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules

76 Modules

The important modules for application programmers are:

- *Wd_types* (→ 205): Defines the fundamental types
- *Wd_dialog* (→ 187): Contains the base class `dialog` from which the programmer's dialog classes should inherit
- *Wd_template* (→ 200): Access to template definitions, including the possibility to instantiate templates and convert them to strings.
- *Wd_cycle* (→ 186): Processes a CGI request and computes the response, i.e. this is the entry point to request processing
- *Wd_transform* (→ 204): Here are the functions to load the UI file (it mainly contains the transformation engine, but this is hidden)
- *Wd_run_cgi* (→ 196): A sample main program for CGI scripts (can be used instead of calling `Wd_cycle` and `Wd_transform` directly).
- *Wd_run_jserv* (→ 197): A sample main program for JSERV servlets (can be used instead of calling `Wd_cycle` and `Wd_transform` directly).

Wd_application_dtd

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_application_dtd

77 WDialog API for Objective Caml: Wd_application_dtd

78 Module `Wd_application_dtd` : Contains the DTD of the UI language

```
val dtd_1 : string
```

The WDialog DTD version 1 as string

```
val dtd_2 : string
```

The WDialog DTD version 2 as string

Wd_cycle

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_cycle

79 WDialog API for Objective Caml: Wd_cycle

80 Module `Wd_cycle` : The scope of the module `Wd_cycle` is the whole CGI cycle, from the moment when the CGI request has just arrived until the moment just before the CGI response is delivered to the client.

```
val process_request :
  ?session_manager:Wd_types.session_manager_type ->
  ?self_url:string ->
  ?response_header:Wd_types.response_header ->
  Wd_types.universe_type -> Netcgi_types.cgi_activation -> unit
```

This is the main function processing requests coming from the browser. It expects a CGI environment, interprets the CGI variables and performs one request cycle:

- The old object is restored (deserialized)
- The variables are updated that are bound to interactors, i.e. the user modifications are propagated to these variables
- The current event is determined by checking which button or link has been clicked
- The method `handle` of the object is invoked
- The result of `handle` is interpreted
- The method `prepare_page` is invoked on the (probably next) object
- The HTML code for the selected page is generated The resulting HTML code is written to the output channel of the `cgi_activation` object passed to this function.

For more details of the request cycle, see the documents standard-cycle.txt and popup-cycle.txt.

```
val make_environment : Netcgi_types.cgi_activation -> Wd_types.environment
```

Creates an (otherwise empty) environment with the passed CGI activation object.

Wd_dialog

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_dialog

81 WDialog API for Objective Caml: Wd_dialog

82 Module `Wd_dialog` : This module defines the base class `dialog` from which all dialog classes should inherit

```
val dump_interactors : Format.formatter -> Wd_types.interactors -> unit
```

Prints a readable description of the contents of the `interactors` argument to the passed `formatter`. This function is a valid printer for the toplevel and the debugger.

```
class virtual dialog : Wd_types.universe_type -> string -> Wd_types.environment ->
  object
```

`inherit Wd_types.dialog_type` [104]

This class implements `dialog_type`

82.0.1 The following private methods are shortcuts for the `Wd_template`[98] module.

```
method private t_get : string -> Wd_template.template
```

Returns the template

```
method private t_apply :
  Wd_template.template -> (string * Wd_template.tree) list -> Wd_template.tree
```

Applies the template, and passes the parameters to it

```
method private t_apply_byname :
  string -> (string * Wd_template.tree) list -> Wd_template.tree
```

Applies the named template, and passes the parameters to it

```
method private t_apply_lazily :
  string -> (string * Wd_template.tree) list -> Wd_template.tree
```

Constructs the expression that applies the names template on demand.

```
method private t_concat :
  Wd_template.tree -> Wd_template.tree list -> Wd_template.tree
```

Concatenates the `tree list`, and put the separator `tree` between the parts

```
method private t_empty : Wd_template.tree
```

The empty XML tree

`method private t_text : string -> Wd_template.tree`

The XML tree containing this text

`method private t_html : string -> Wd_template.tree`

The XML tree containing this HTML material

`method private t_to_string : Wd_template.tree -> string`

Evaluates the XML tree in the current environment, and returns the corresponding output string (HTML code)

`method private put_tree : string -> Wd_template.tree -> unit`

Sets the `string` variable to the string representation of the `tree`

`end`

The class `dialog` is the implementation of `Wd_types.dialog_type`[104] that should be used as base definition for all dialog objects. This class defines all methods but `handle` and `prepare_page` - these must be defined in the subclasses provided by the application programmer. Furthermore, there are some extensions to `Wd_types.dialog_type`[104] realized as private methods to simplify access to templates. These additional methods begin with `t_`, but are otherwise named like the corresponding function in the module `Wd_template`[98].

The class takes three anonymous parameters:

- The universe for which the dialog is defined
- The name of the dialog in the UI definition
- The environment of the current CGI activation

Construction `new dialog universe name env`: Makes a new dialog called `name` which is bound to the `universe`

Normally, the class `dialog` is not used directly. It is intended to inherit from `dialog`, and to define the two methods `prepare_page` and `handle`; these are virtual in the base class:

```
class my_dialog universe name env =
   object (self)
     inherit dialog universe name env
   method prepare_page = ...
   method handle = ...
 end
```

After defining such a class, you can create a new `my_dialog` by `new my_dialog universe "my_dialog" env`. However, it is not recommended to do it this way. The module `Wd_universe`[108] includes a registry of constructors, and every dialog should put its constructor into it:

```
universe # register "my_dialog" (new my_dialog)
```

For example, you can place this statement where you call `Wd_run_cgi.run`[90]:

```
let universe = new Wd_universe.universe in
 universe # register ...;
 Wd_run_cgi.run ... ~universe
```

After the dialog has been registered, you can create new instances of the dialog by calling `universe#create`:

```
let dlg = universe # create env "my_dialog"
```

```
class instant_session_manager : unit -> Wd_types.session_manager_type
```

The "instant" session manager serializes the state of the dialog, and includes it literally into the HTML form. The serialized string is included in the generated HTML forms, and passed by CGI parameter from one activation to the next activation. The advantages are that you need not to set up any database to store the sessions, and that the session state is included in the history of the browser. The disadvantages are increased network traffic, and missing protection of the session data (the end user can decode the session state), i.e. the class is insecure, and should only be used in trusted environments.

`exception Invalid_session_checksum`

Raised when the session checksum as stored in the database is not the same that is transmitted by the browser. This normally means that a historic page was submitted ("Back" button).

`exception Session_not_found`

Raises when the current session was not found in the database.

`class database_session_manager : ?private_key:string -> ?enable_checksum:bool -> allocate:(unit -> int) ->`

The database session manager allows it to store the contents of sessions in databases. The database table should consist of rows with at least the four columns

- `id`: The row identifier chosen by the database system. This is an integer
- `skey`: The secure key consists of 32 hex digits, and is another identifier for the row that is chosen by WDialog
- `value`: The value is a large BASE64-encoded string. The length depends on the session size
- `checksum`: A checksum of `value`, again 32 hex digits

Note that sessions stored in databases behave differently with respect to the history function of the browser. It is not possible to "go back", or to select historic pages otherwise, and submit them again. This will be detected by the `database_session_manager`, and the exception `Invalid_session_checksum` will be raised. There is currently no way to turn this check off, nor to make the check more fine-grained. (Maybe an implementation with multiple versions of the same session is the way to go, I don't know yet.)

The `database_session_manager` does never delete sessions. It is recommended to remove unused sessions after a timeout period.

For a transaction-based DBMS, it is sufficient if the `update` function commits the current transaction.

Wd_dictionary

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_dictionary

83 WDialog API for Objective Caml: Wd_dictionary

84 Module `Wd_dictionary` : Dictionaries are maps from string to any type.

See the module `Map` of the standard library for details.

```
type +'a t
val empty : 'a t
val add : string -> 'a -> 'a t -> 'a t
val find : string -> 'a t -> 'a
val remove : string -> 'a t -> 'a t
val mem : string -> 'a t -> bool
val iter : (string -> 'a -> unit) -> 'a t -> unit
val map : ('a -> 'b) -> 'a t -> 'b t
val mapi : (string -> 'a -> 'b) -> 'a t -> 'b t
val fold : (string -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
val of_alist : (string * 'a) list -> 'a t
```

Convert the passed associative list to the corresponding dictionary

```
val to_alist : 'a t -> (string * 'a) list
```

Convert the dictionary to an associative list

Wd_encoding

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_encoding

85 WDialog API for Objective Caml: Wd_encoding

86 Module `Wd_encoding` : This module contains the output encodings.

The encodings are functions from `string` to `string`. They are usually entered into the application by default (see the method `output_encoding` of the `Wd_types.application_type`[104]), and accessible by `<ui:encoding>` and other elements of the UI language.

The functions defined here assume that an ASCII-compatible character set is used.

`val encode_as_html : string -> string`

Encodes strings as HTML:

- '<' becomes `<`
- '>' becomes `>`
- '"' becomes `"`
- '&' becomes `&`

All other characters remain unchanged.

From `<ui:encoding>`, this function is accessible under the name `html`.

`val encode_as_pre : string -> string`

Does the following encoding:

- ' ' becomes ` `
- '\\n' becomes `<br>`
- '\\t' is expanded to a sequence of ` `, tab width is 8

All other characters remain unchanged.

From `<ui:encoding>`, this function is accessible under the name `pre`.

`val encode_as_para : string -> string`

Multiple linefeeds are replaced by <p>.

From `<ui:encoding>`, this function is accessible under the name `para`.

`val encode_as_js_string : string -> string`

Encodes strings such that they can be placed between quotes in Javascript.

- '\\' becomes backslash backslash
- '"' becomes backslash double-quotes
- "'" becomes backslash single-quote

- '<' becomes \\x3c
- '%' becomes \\x25

The characters 0 to 31 and 127 become \\xHH All other characters remain unchanged.

From `<ui:encoding>`, this function is accessible under the name `js`.

```
val encode_as_js_longstring : enc:Pxp_types.rep_encoding -> string -> string
```

Similar to `encode_as_js_string`, but an additional rules prevents that long lines result. (Javascript interpreters often cannot cope with long lines.) The string is split up into pieces p1,...,pN and these are connected by p1" + \\n "p2" + \\n ... + \\n "pN

From `<ui:encoding>`, this function is accessible under the name `jslong`.

It is required to pass the character encoding of the strings as argument `enc`. In the case the encoding uses multi-byte character representations, it is avoided to break up these characters.

Note that you can combine the various encodings to get new effects. Reasonable combinations are:

- encode_as_html then encode_as_pre
- encode_as_html then encode_as_para
- encode_as_html then encode_as_para then encode_as_pre
- encode_as_html then encode_as_js_string
- encode_as_html then encode_as_js_longstring

... and more

Wd_interactor

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_interactor

87 WDialog API for Objective Caml: Wd_interactor

88 Module `Wd_interactor` : This module provides a data type for containers of interactor descriptions.

The background of the module is that interactors (such as input elements, hyperlinks, buttons) are referred to by two different naming schemes. On the one hand, the interactor elements in the ui file have a name and and optionally an index; e.g.

```
<ui:button label="Press here" name="my_button" index="1"/>
```

or

```
<ui:text variable="x" index="5"/>
```

Often the name/index pair is actually a variable/index pair because the interactor is bound to a certain instance variable of the uiobject. \- On the other hand, the interactors are transformed to HTML code, and the HTML/CGI name is different. For example the following HTML output might have been generated:

```
<input type="button" name="xbutton_25" value="Press here">
```

or

```
<input type="text" name="var_12" value="...">
```

Obviously, the HTML/CGI names are automatically chosen. Of course, there is a bijective mapping from the internal names to the HTML/CGI names, and the task of the Interactor.t type is to manage these mappings.

A value of type Interactor.t contains the following:

- A list of name/index pairs
- The corresponding HTML/CGI identifiers. Only the identifier after the prefix (such as "var_" or "xbutton_") is stored.
- The bijection which name/index pair corresponds to which identifier.
- For every name/index pair (or every identifier) there may be another arbitrary value which can store additional information related to the interactor

Example

```
let ia = create();;    (* : unit Interactor.t *)
 ...
 let id = add ia "my_button" "1" None ();;   (* = "25" *)
```

Note that the "25" is an automatically created identifier (just the next free number), and that the HTML/CGI name will be composed of a prefix indicating the type of the interactor and the identifier, e.g. "xbutton_25".

If we try to add the same name/index pair a second time, the add operation raises an exception:

```
add ia "my_button" "1" None ();;
```

==> raises `Element_exists "25"`

When the incoming CGI parameters are analyzed, it must be possible to look up the name/index pair for a given identifier. For example:

```
let (name,index,value) = lookup ia "25";; (* = ("my_button", "1", () *)
```

There is no reverse lookup operation. However, it is at least possible to test the existence of a given name/index pair:

`exists ia "my_button" "1"` yields `true`

As a last resort, it is possible to iterate over all members of the ia value:

```
let print id name index value =
   printf "ID=%s name=%s index=%s\n" id name index
 in
 iter print ia
```

Selecting the identifier manually

If the ui element contains the `cgi="keep"` attribute, and there is no index component, the CGI name retains the orignal name. For example:

```
<ui:button label="Press here" name="press" cgi="keep"/>
```

The generated HTML code:

```
<input type="button" name="button_press" value="Press here">
```

Such an interactor would be added as follows:

```
let id = add ia "press" "" (Some "press") ();;   (* = "press" *)
```

I.e. you can determine the identifier by passing it as fourth argument. Of course, in this case it is checked whether the identifier is still free; and `add` fails if the identifier was already previously used.

Representation

Note that the representation of `Interactor.t` is optimized for efficient serialization (using the Marshal module). Especially, if there are many name/index pairs differing only in the index component, the name component is stored only once.

Further information

See `Wd_types.interactors`[104]

`type 'a t`

The type of interactors

`type id = string`

Interactors are identified by strings

`exception Element_exists of id`

Raised by some functions if there is already a component with the ID

`val create : 'a -> 'a t`

`create x`: Creates a new interactor manager. The value `x` must only be passed because of typing restrictions.

`val clear : 'a t -> unit`

`clear ia`: Removes all entries from the interactor. The sequence generator for the automatic IDs is not reset, however.

```
val add : 'a t ->
  string -> string -> id option -> 'a -> id
```

`add ia name index opt_id value`: adds the triple `(name,index,value)` to `ia`; if `opt_id` is `None`, the triple gets an automatically selected numerical ID; if `opt_id` is `Some id`, the triple has the ID `id`. The function returns the ID.

Fails with `Element_exists` if the pair `(name,index)` is already contained. The argument of this exception is the ID.

```
val lookup : 'a t -> id -> string * string * 'a
```

`lookup ia id`: looks up the triple `(name,index,value)` for the ID `id` within `ia`.

Fails with "Interactor.lookup" if the id cannot be associated.

```
val exists : 'a t -> string -> string -> bool
```

`exists ia name index`: returns whether the pair `(name,index)` exists in `ia`.

```
val iter : (id -> string -> string -> 'a -> 'b) ->
  'a t -> unit
```

`iter f ia`: iterates over the elements of `ia` and invoke the function `f` for every element by `f id name index value`.

```
val serialize : ('a -> string) -> Buffer.t -> 'a t -> unit
```

`serialize f b ia`: Writes the serialization of `ia` to buffer `b`. Contained values are serialized by calling `f`.

```
val unserialize :
  (('a, 'b, 'c) Wd_serialize_types.poly_ds_buf -> 'd) ->
  ('a, 'b, 'c) Wd_serialize_types.poly_ds_buf -> 'd t
```

`unserialize f buf`: Reads the serialized string from `buf` and reconstructs the interactors. Values are read by calling `f`.

Wd_run_cgi

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_run_cgi

89 WDialog API for Objective Caml: Wd_run_cgi

90 Module `Wd_run_cgi` : This module contains a customizable main program for CGIs

```
val adjust_gc : unit -> unit
```

Adjust the garbage collector for short-living processes. This is recommended if used in a CGI environment.

```
val run :
  ?charset:Pxp_types.rep_encoding ->
  ?script:string ->
  ?self_url:string ->
  ?uifile:string ->
  ?session_manager:Wd_types.session_manager_type ->
  ?no_cache:bool ->
  ?error_page:(Netchannels.out_obj_channel -> exn -> unit) ->
  ?cgi:Netcgi_types.cgi_activation ->
  ?response_header:Wd_types.response_header ->
  ?reg:(Wd_types.universe_type -> unit) -> unit -> unit
```

A customizable "main program" which processes CGI requests and generates responses. It includes support for error handling.

The simplest way to make a working CGI program is to call `run ~reg ()`; the argument `~reg` registers the dialog classes. The other arguments have reasonable defaults for a normal CGI environment.

The function passed to `~reg` gets the universe as argument. Its task is to register the dialog classes by calling the method `register` of the universe.

`run` loads the UI file, initializes the CGI environment, processes the request (see `Wd_cycle`[80]), and generates the response. If an error happens, an error page is generated describing the exception. The optional arguments modify the standard behaviour:

Wd_run_jserv

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_run_jserv

91 WDialog API for Objective Caml: Wd_run_jserv

92 Module `Wd_run_jserv` : This module contains a customizable main program for application servers connected with the JSERV protocol

```
val create_request_handler :
  ?charset:Pxp_types.rep_encoding ->
  ?session_manager:Wd_types.session_manager_type ->
  ?no_cache:bool ->
  ?error_page:(Netchannels.out_obj_channel -> exn -> unit) ->
  ?response_header:Wd_types.response_header ->
  ?reg:(Wd_types.universe_type -> unit) ->
  uifile:string -> unit -> Netcgi_jserv_app.request_handler
```

This function creates a request handler for a JSERV-based application server. Use this function like in this example:

```
let req_hdl = create_request_handler ... () in
 let server = `Forking(20, [ "appname", req_hdl ]) in
 Netcgi_jserv.jvm_emu_main
   (Netcgi_jserv_app.run server `Ajp_1_2)
```

This main program creates a "forking server" that starts a new subprocess (up to 20) for every arriving request. This is currently the recommended mode. It sounds a bit like CGI, but is much faster because the subprocess is already initialized when it forks.

The application is accessible under the URL `http://yourserver/servlets/appname`. It is possible to bind several request handlers at the same time.

Call the function `create_request_handler` as follows:

```
let req_hdl = create_request_handler ~reg ~uifile ()
```

The argument `~reg` registers the dialog classes (like in Wd_run_cgi). The argument `~uifile` must be the absolute path of the UI definition. The suffix of this file must be ".ui" or ".ui.bin".

Wd_run_fcgi

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_run_fcgi

93 WDialog API for Objective Caml: Wd_run_fcgi

94 Module `Wd_run_fcgi` : This module contains a customizable main program for application servers connected with the fastcgi protocol

```
val create_request_handler :
  ?charset:Pxp_types.rep_encoding ->
  ?session_manager:Wd_types.session_manager_type ->
  ?no_cache:bool ->
  ?error_page:(Netchannels.out_obj_channel -> exn -> unit) ->
  ?response_header:Wd_types.response_header ->
  ?reg:(Wd_types.universe_type -> unit) ->
  uifile:string -> unit -> Netcgi_types.cgi_activation -> unit
```

This function creates a request handler for a fastcgi-based application server. Use this function like in this example:

Netcgi_fcgi.serv (Wd_run_fcgi.create_request_handler ~session_manager: (new memory_session_manager) ~charset:'Enc_utf8 ~reg ~uifile:"adder.ui" ()) buffered_transactional_optype

This main program creates a single threaded (single process) server which handles requests sequentially. Since fastcgi processes are managed by an application server, this is the only mode provided. The application server can then be configured to create process pools, cgi style servers, etc.. See the manual for your fastcgi application server for details.

The application is accessible under the URL which you have configured for it in your web server.

Call the function `create_request_handler` as follows:

```
let req_hdl = create_request_handler ~reg ~uifile ()
```

The argument `~reg` registers the dialog classes (like in Wd_run_cgi). The argument `~uifile` must be the absolute path of the UI definition. The suffix of this file must be ".ui" or ".ui.bin".

Wd_stdlib

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_stdlib

95 WDialog API for Objective Caml: Wd_stdlib

96 Module `Wd_stdlib` : An internal module containing the template stdlib

```
val stdlib_iso88591_1 : string
```

The contents of stdlib.xml as Pxp_marshal'ed string (ISO-8859-1)

```
val stdlib_utf8_1 : string
```

The contents of stdlib.xml as Pxp_marshal'ed string (UTF8)

Wd_template

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_template

97 WDialog API for Objective Caml: Wd_template

98 Module `Wd_template` : User access to the template definitions

```
exception Template_not_found of string
```

Raised when a template cannot be found. The argument is the name of the missing template

```
type template
```

The type of template definitions

```
type tree
```

The type of instantiated templates ("XML trees")

Many of the following functions either have an `Wd_types.application_type`[104] object as first argument or a `Wd_types.dialog_type`[104] object. These objects contain "background information" that are necessary to do their job.

Note that the base dialog class `Wd_dialog.dialog`[82] defines a number of private methods that correspond to the following functions, but that know the first argument implicitly. For example, you can call

```
self#t_to_string tree
```

instead of

```
Wd_template.to_string self tree
```

.

```
val get : Wd_types.application_type -> string -> template
```

`get app n`: looks up the template with name `n`, or raise `Template_not_found`.

```
val apply :
  Wd_types.dialog_type ->
  template -> (string * tree) list -> tree
```

`apply dlg t params`: instantiates the template `t` with parameters `params` and returns the result. If more parameters are supplied than actually referenced in the template, these are silently ignored. If parameters are referenced that are missing in `params`, an `Wd_types.Instantiation_error`[104] happens.

```
val apply_byname :
  ?localized:bool ->
  Wd_types.dialog_type ->
  string -> (string * tree) list -> tree
```

`apply_byname dlg n params = apply (get n) params`: instantiates the template with name `n` with parameters `params` and returns the result. See `apply` for the possible exceptions.

`~localized`: If `true` (default), the localized template is returned, if available. If `false`, exactly the template is returned that has been requested.

```
val apply_lazily :
  Wd_types.dialog_type ->
  string -> (string * tree) list -> tree
```

`apply_lazily dlg n params`: creates a tree node which applies the template `n` by instantiating the parameters `params`. The tree node is simply a "`<ui:use template=...>...</ui:use>`" node, and it is expanded only when needed.

```
val concat : Wd_types.application_type ->
  tree -> tree list -> tree
```

`concat app sep l`: if `l = [n1;n2;...;nN]`, the concatenation `n1 . sep . n2 . sep ... sep . nN` is formed.

```
val empty : Wd_types.application_type -> tree
```

A tree node which expands to the empty string.

```
val text : Wd_types.application_type -> string -> tree
```

`text app s`: Forms a tree node which expands to the text `s`. This means if `s` contains meta characters (esp. <, >, &) these are converted to their corresponding entities (< > & etc)

```
val html : Wd_types.application_type -> string -> tree
```

`html app s`: Forms a tree node which expands exactly to `s`. This means that if `s` contains HTML meta characters these are left as they are. Because of this it is possible to include HTML elements. The string `s` is not parsed.

NOTE: It is **not** possible to create ui:* elements with this function that will be interpreted. If you want to create such elements you must instantiate templates containing them.

Internally, this function creates a `ui:special` node.

```
val to_string : Wd_types.dialog_type -> tree -> string
```

`to_string dlg t`: converts the tree representation `t` to its string representation. You need this function to put a tree into a string variable. You can then insert this variable into your document by using `<ui:dynamic variable="name-of-variable" special="yes"/>`.

For convenience, dialogs have a method `put_tree` putting a tree into string variable.

NOTE: The expansion process assumes that the generated HTML code will be inserted into a main page and **not** a popup page.

Wd_templrep

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_templrep

99 WDialog API for Objective Caml: Wd_templrep

100 Module `Wd_templrep` : Template representation

```
type 'a Pxp_document.node #Pxp_document.extension as 'a t
```

The type of prepared templates

```
type 'a Pxp_document.node #Pxp_document.extension as 'a param = {
  param_tree : 'a Pxp_document.node ;
```

parameter as XML tree

```
  param_text : string Lazy.t ;
```

parameter as string

```
}
```

The type of parameters to instantiate. In `param_tree`, the instantiation text must be passed as XML tree; in `param_text`, the same must be passed as string. The first component is used if a parameter is found in element context; the second component is used if a parameter is found in an attribute value. `param_text` is a lazy-evaluated value because it is unlikely that the string transcription is actually used.

```
type expr =
  | Expr_var of string
```

a variable in an expr

```
  | Expr_strconst of string
```

a string constant

```
  | Expr_apply of (string * expr list)
```

a function call

```
  | Expr_param of (string * string list)
```

a template parameter

Expressions inside $...

```
val prepare_tree_with_parameters :
  mk_uiencode:(unit ->
               ('a Pxp_document.node #Pxp_document.extension as 'a)
               Pxp_document.node) ->
  string ->
  Wd_types.application_type -> 'a Pxp_document.node list -> 'a t
```

`let pt = prepare_tree_with_parameters ~mk_uiencode name app nodes`:

Prepares the node list `nodes` for the instantiation of parameters. `nodes` remains unmodified; but references to inner nodes of `nodes` are stored in `pt` (this should be transparent). `app` is the application. `name` is the name of the template; it is only used to generate error messages.

The preparation procedure scans all data nodes and all attribute values for '$' parameters. `pt` contains a restructured copy of `nodes` in which the parameters are specially marked up such that the instantiation procedure can find them quickly.

Recognized forms:

- `$name`: Replaced by the parameter `name` here
- `${name}`: Same
- `${name/enc1/enc2/...}`: Apply the encodings `enc1`,`enc2`,... in turn before replacing
- `$[name]`: Replaced by the value of the dialog variable `name`.
- `$[name/enc1/enc2/...]`: Apply the encodings `enc1`,`enc2`,... in turn before replacing

`~mk_uiencode`: This function must return an empty `<ui:encode/>` node. Such nodes are generated for `${name/enc1/enc2/...}` in element context.

```
val get_parameters :
  ('a Pxp_document.node #Pxp_document.extension as 'a) t ->
  unit Wd_types.dict
```

Returns the parameters that have been found in the template as dictionary

```
val instantiate :
  ?eval_expr:(expr -> string) ->
  ('a Pxp_document.node #Pxp_document.extension as 'a) t ->
  'a param Wd_types.dict list -> 'a Pxp_document.node -> unit
```

`instantiate ~eval_expr pt params container`:

Instantiates the already prepared tree `pt`, and sets the sub nodes of `container` to the new instance (the instance is a node list). The parameters are searched in `params`, strictly from left to right.

An `Instantiation_error` is raised if something goes wrong.

`~eval_expr`: Evaluates an expression found inside brackets `$[expr]` and returns the result as string. The expression never contains `Expr_param` nodes, because these have already been replaced by `Expr_strconst` nodes.

Wd_transform

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_transform

101 WDialog API for Objective Caml: Wd_transform

102 Module `Wd_transform` : This is the processor transforming the ui file to HTML output.

```
class syntax_tree : Wd_types.syntax_tree_type
```

The realization of the syntax tree

```
val parse_uiapplication :
  ?charset:Pxp_types.rep_encoding -> string -> Wd_types.application_type
```

Parses the file whose name is passed to the function, and returns the contents of the ui file as application declaration.

E.g. `parse_uiapplication "index.ui"`

`~charset`: This argument determines the _internal_ encoding of the characters. The internal encoding may be different from the encoding found in the parsed files; if necessary, the characters are recoded. This argument determines also the charset of the returned application, and thus indirectly:

- The charset of dialog variables and other state data
- The charset of the generated HTML pages

```
val load_uiapplication :
  ?charset:Pxp_types.rep_encoding -> string -> Wd_types.application_type
```

Loads the ui definition contained in the binary file, and returns the contents as application declaration.

E.g. `load_uiapplication "index.ui.bin"`

`~charset`: See also `parse_uiapplication`. The charset MUST be the same as the charset used in the compiled binary. There is no check whether this is actually true.

```
val compile :
  ?charset:Pxp_types.rep_encoding -> string -> Pervasives.out_channel -> unit
```

Compiles the file and writes it to the `out_channel`. The compiled file can later be loaded by `load_uiapplication` which is much faster than `parse_uiapplication`.

`~charset`: See also `parse_uiapplication`. This charset determines the charset used in the compiled binary. It must be the same as the charset used to `load_uiapplication`

Wd_types

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_types

103 WDialog API for Objective Caml: Wd_types

104 Module `Wd_types` : This module defines the fundamental types for WDialog

```
type 'a dict = 'a Wd_dictionary.t
```

The type `'a dict` is just an abbreviation for dictionaries (mappings from strings to `'a`).

```
type event =
  | Button of string
  | Image_button of (string * int * int)
  | Indexed_button of (string * string)
  | Indexed_image_button of (string * string * int * int)
  | No_event
  | Popup_request of string
```

Events are generated when the user presses a button, clicks at a hyperlink, or opens a server-generated popup window.

- `Button n`: The button with name `n` has been pressed (`<ui:button name="n" ...>`)
- `Image_button (n,x,y)`: The image button with name `n` has been pressed at the coordinates `(x,y)`. (`<ui:imagebutton name="n" ...>`)
- `Indexed_button (n,i)`: The button with name `n` and index `i` has been pressed (`<ui:button name="n" index="i" ...>`)
- `Indexed_image_button (n,i,x,y)`: The image button with name `n` and index `i` has been pressed at the coordinates `(x,y)`. (`<ui:imagebutton name="n" index="i" ...>`)
- `No_event`: This value indicates that there was no event (or no recognized event)
- `Popup_request s`: A server popup window has just popped up, and the contents for the window have been requested. The argument is the second argument of the "open" Javascript function that has been generated by the ui:server-popup element.

```
type interactors = {
  mutable ui_buttons : string option Wd_interactor.t ;
  mutable ui_imagebuttons : string option Wd_interactor.t ;
  mutable ui_anchors : string option Wd_interactor.t ;
  mutable ui_indexed_buttons : string option Wd_interactor.t ;
  mutable ui_indexed_imagebuttons : string option Wd_interactor.t ;
  mutable ui_indexed_anchors : string option Wd_interactor.t ;
  mutable ui_vars : unit Wd_interactor.t ;
  mutable ui_enumvars : (string * string option * string) list ;
  mutable ui_uploads : unit Wd_interactor.t ;
}
```

Values of this type store the mapping from the (name,index) pairs of interactor elements to the real CGI parameter names together with auxiliary components.

What's the problem? Interactor elements like `ui:a` allow the programmer to identify these either by a single name (which is an arbitrary string) or by a pair of a "name" and an "index" (i.e. two arbitrary strings). In contrast to this, CGI parameter names are much more restricted. First, these names are only strings; there is no built-in representation for pairs of strings. Second, you cannot use arbitrary characters within these names because of limitations of the transport protocol and because of bugs in browsers. (As the "multipart/form-data" representation used in the transport protocol bases on the RFC 822 mail format, it is neither possible to pass 8 bit values, nor to pass control characters. A known bug in Netscape browsers is that the double-quote and backslash characters are incorrectly represented.)

The solution is to generate the CGI parameter names using only unproblematic characters, and to keep the mapping of the original `string` or `string * string` to the generated CGI name. This record stores the mappings for the various namespaces. The mapping is encapsulated in the `Wd_interactor`[88] module, and the type `'a Wd_interactor.t` (where `'a` is arbitrary) represents such a mapping. The CGI names are simply enumerated, and the `Wd_interactor.t` value stores only the numbers (IDs) of the CGI parameters. The complete CGI name is formed using a prefix and the number stored in `Wd_interactor.t`. For example, ui:buttons without index have the prefix "button_"; and if the ui_button record component contains entries for the IDs 0, 1, and 2, the complete CGI names are "button_0", "button_1", and "button_2", respectively.

The type parameter `'a` of `'a Wd_interactor.t` is the type of the auxiliary component stored with every entry. See below for the meanings in every case.

The `Wd_interactor.t` structure always maps from pairs `string * string` to numeric IDs. Because of this, there are usually two mappings, one for the (name,index) pairs, and one for the simple names. The record components listed below that have a name with "indexed" are responsible for the pairs, the corresponding component without "indexed" stores the mapping for the simple case.

The simple case is represented as `Wd_interactor.t` by using always the empty string `""` as index.

THE COMPONENTS ARE:

1. **BUTTON-TYPE INTERACTORS:**
   - `ui_buttons`: Enumerates the `ui:button` interactors that have only a name, not an index. CGI prefix: "button_".
   - `ui_indexed_buttons`: Enumerates the `ui:button` interactors that have both a name and an index. CGI prefix: "xbutton_".
   - `ui_imagebuttons`: Enumerates the `ui:imagebutton` interactors that have only a name, not an index. CGI prefix: "imagebutton_".
   - `ui_indexed_imagebuttons`: Enumerates the `ui:imagebutton` interactors that have both a name and an index. CGI prefix: "ximagebutton_".
   - `ui_anchors`: Enumerates the `ui:a` interactors that have only a name, not an index. CGI prefix: "anchor_".
   - `ui_indexed_anchors`: Enumerates the `ui:a` interactors that have both a name and an index. CGI prefix: "xanchor_".

   These components use the auxiliary component to store the "goto" attribute of the interactor, if present.
2. **BOX-LIKE INTERACTORS:**
   - `ui_vars`: Enumerates the interactors that are bound to an object variable (i.e. they have a `variable` attribute, like `ui:text`). As variables allow either only non-indexed names or only indexed names, this component contains both interactors identified by a simple string name and interactors identified by (name,index) pairs; it is not possible that there are conflicts between the two naming methods. CGI prefix: "var_". This component does not have an auxiliary component.
   - `ui_enumvars`: This list enumerates the simple names `(name, None, pg)` or pairs `(name, Some index, pg)` that occur in checkbox, radiobutton and select list interactors. `pg` is the name of the page where the interactor occurs (knowing the page is necessary for popup dialogues).

- `ui_uploads`: Enumerates the `ui:file` interactors. These may only have a simple name. CGI prefix: "upload_". This component does not have an auxiliary component.

```
type 'a poly_var_value =
  | String_value of string
  | Enum_value of string list
  | Dialog_value of 'a option
  | Dyn_enum_value of (string * string) list
  | Alist_value of (string * 'a poly_var_value) list
```

`poly_var_value` is the type of instance variables of dialogs. There are six possibilities for `poly_var_value`s:

- `String_value s`: The instance variable contains the string `s`
- `Enum_value [x1;x2;...]`: The instance variable contains an enumerator value with the internal items `x1`, `x2`, etc. (As variables are declared it is known which items are possible, and whether there are corresponding external values.)
- `Dialog_value None`: The instance variable does not contain a dialog.
- `Dialog_value (Some dlg)`: The variable contains the dialog dlg.
- `Dyn_enum_value [(x1,y1);(x2,y2);...]`: The variable contains the enumerator with internal items `x1`, `x2`,... and the corresponding external values `y1`, `y2`, ...
- `Alist_value [(i1,v1); (i2,v2); ...]`: The variable contains the associative list where the index `i1` is mapped to the value `v1`, `i2` is mapped to `v2` etc.

See also `Wd_types.var_value`[104] below.

```
type enum_decl = {
  enum_name : string ;
  mutable enum_definition : (string * string) list ;
}
```

The type of an enumeration declaration (i.e. `ui:enumeration`). The component `enum_name` is the name of the enumeration. The component `enum_definition` is the list of the pairs of internal and external names, in the right order.

```
type var_type_name =
  | String_type
  | Enum_type of enum_decl
  | Dialog_type
  | Dyn_enum_type
```

The different variable types

```
type 'a poly_var_decl = {
  var_name : string ;
  var_type : var_type_name ;
  var_default : 'a poly_var_value option ;
  var_temporary : bool ;
  var_associative : bool ;
  var_protected : bool ;
}
```

This record describes variable declarations. `var_name` contains the name of the variable. `var_type` is the declared type according to the `type` attribute. `var_default` contains either `None`, in which case no special initial value is declared, or `Some v`, where `v` is the initial value. `var_temporary` corresponds to the `temporary` attribute, `var_associative` to the `associative` attribute, and finally `var_protected` to the `protected` attribute.

```
type response_header = {
  mutable rh_status : Netcgi_types.status ;
  mutable rh_content_type : string ;
  mutable rh_cache : Netcgi_types.cache_control ;
  mutable rh_filename : string option ;
  mutable rh_language : string option ;
  mutable rh_script_type : string option ;
  mutable rh_style_type : string option ;
  mutable rh_set_cookie : Netcgi_types.cgi_cookie list ;
  mutable rh_fields : (string * string list) list ;
}
```

This record contains the CGI header of the response. It is initialized quite early and can be modified while executing the request. The fields correspond to the arguments of the method `set_header` of the class type `cgi_activation`, defined in `Netcgi_types`.

```
type debug_mode_style = [ `Fully_encoded | `Partially_encoded ]
```

Whether the generated HTML comments do escape HTML meta characters always (`` `Fully_encoded ``), or only partially (`` `Partially_encoded ``).

```
type environment = {
  debug_mode : bool ;
  debug_mode_style : debug_mode_style ;
  prototype_mode : bool ;
  server_popup_mode : bool ;
  self_url : string ;
  response_header : response_header ;
  cgi : Netcgi_types.cgi_activation ;
}
```

This record contains data that may be different for every CGI request. The `debug_mode` and `prototype_mode` components are true iff the corresponding mode is switched on. The `server_popup_mode` is true iff the current request is a popup request. `self_url` is the URL that invokes the CGI recursively. `cgi` contains the full CGI request. The `request_header` is the designated header of the HTTP response.

```
type trans_vars = {
  mutable within_popup : bool ;
  mutable current_page : string ;
  mutable popup_env_initialized : bool ;
  mutable condition_code : bool ;
  mutable serialize_session : unit -> string ;
}
```

This record is private for the transformation engine.

```
type ('a, 'b, 'c) poly_ds_buf = ('a, 'b, 'c) Wd_serialize_types.poly_ds_buf = {
  ds_str : string ;
  mutable ds_pos : int ;
  ds_end : int ;
  ds_universe : 'a ;
  ds_environment : 'c ;
  ds_dialogs : (int, 'b) Hashtbl.t ;
}
```

The deserialization buffer

```
class type dialog_decl_type =
  object
```

`method name : string`

The name of the dialog

`method enumeration : string -> Wd_types.enum_decl`

Returns the declared enumeration

`method variable_names : string list`

Returns the names of all declared variables

`method variable : string -> Wd_types.dialog_type Wd_types.poly_var_decl`

Returns a single variable declaration

`method page_names : string list`

Returns the names of all declared pages

`method page : string -> Wd_types.syntax_tree_type`

Returns the XML tree of the demanded page (i.e. the `ui:page` node)

`method page_is_declared_as_popup : string -> bool`

Returns whether the page is declared as popup page (attribute `popup`

`method start_page : string`

Returns the name of the start page

`method default_context : Wd_types.syntax_tree_type Wd_types.dict`

Returns the default context of the dialog, i.e. the context parameters that are declared by the `ui:context` element occurring directly in the `ui:dialog` element, and not in a certain page.

`method language_variable : string option`

Returns `Some v` if a language variable called `v` is declared, or `None` if no such variable exists.

```
  end
```

This class type contains the dialog declaration

```
class type virtual dialog_type =
  object
```

`method copy : Wd_types.dialog_type`

return a copy of the dialog
The copy is not attached to a session, even if the original object is.

`method name : string`

return the name of the dialog

`method page_name : string`

return the name of the current page

`method page_names : string list`

returns the names of all defined pages of this dialog

`method variable : string -> Wd_types.dialog_type Wd_types.poly_var_value`

`variable n`: Get the variable with name `n`, or raise `No_such_variable`. The name `n` cannot only refer to variables declared in this dialog, but also to variables of subdialogs using the dot notation: "name1.name2...nameN". Here, `name1` must be a dialog variable, and `name2` a variable of this subdialog, and so on, until the final variable `nameN` is reached, whose value is returned. - The dot notation is accepted by the following methods, too.

```
method variable_decl : string -> Wd_types.dialog_type Wd_types.poly_var_decl
method string_variable : string -> string
```

`string_variable n`: Get the contents of the string variable `n`. Raise `No_such_variable` if the variable does not exist. `Runtime_error` if the variable is not a string.

`method enum_variable : string -> string list`

`enum_variable n`: Get the contents of the enumerator variable `n`. The enumerator must have been declared. The returned list contains only the internal values. Raise `No_such_variable` if the variable does not exist. `Runtime_error` if the variable is not a declared enumerator.

`method dyn_enum_variable : string -> (string * string) list`

`dyn_enum_variable n`: Get the contents of the enumerator variable `n`. The enumerator may be dynamic or may be declared. The returned list contains both the internal and the external values. Raise `No_such_variable` if the variable does not exist. `Runtime_error` if the variable is not an enumerator.

`method dialog_variable : string -> Wd_types.dialog_type option`

`dialog_variable n`: Get the contents of the dialog variable `n`. Raise `No_such_variable` if the variable does not exist. `Runtime_error` if the variable is not a dialog.

```
method alist_variable :
  string -> (string * Wd_types.dialog_type Wd_types.poly_var_value) list
```

`alist_variable n`: Get the contents of the associative variable `n`. Raise `No_such_variable` if the variable does not exist. `Runtime_error` if the variable is not an alist.

```
method lookup_string_variable : string -> string -> string
method lookup_enum_variable : string -> string -> string list
method lookup_dyn_enum_variable : string -> string -> (string * string) list
method lookup_dialog_variable :
  string -> string -> Wd_types.dialog_type option
```

`lookup_*_variable n x`: Get the contents of the associative variable `n` at index `x`. `No_such_variable` if the variable does not exist. `Not_found` if the variable is undefined at `x`. `Runtime_error` if the variable has the wrong type.

```
method set_variable :
  string -> Wd_types.dialog_type Wd_types.poly_var_value -> unit
```

`set_variable n v`: Sets the variable `n` to the value `v`. The value `v` must be compatible to the declared type of the variable. `No_such_variable` if the variable does not exist. `Runtime_error` if the variable is not compatible.

```
method unset_variable : string -> unit
```

`unset_variable n`: Sets the variable `n` to the declared default value. If the default is not declared, the following default values apply: For strings: the default is the empty string. For enumerators: the default is the empty list. For dialogs: the default is that the value does not exist.

```
method lookup_uploaded_file : string -> Netcgi_types.cgi_argument option
```

`lookup_uploaded_file name`: Checks whether the file upload box `name` was used. If so, `Some arg`, where `arg` is the transporting CGI argument is returned. If the box was not used, but the box exists, `None` is returned. Raises a `Runtime_error` if the named box does not exist.

Important note: Uploaded files are not persistent. This means that they are only existent in the `handle` phase, not during initialization nor the `prepare_page` phase. You get a failure "Upload.get: not initialized" if you try to access uploaded files in the wrong moment.

```
method dump : Format.formatter -> unit
```

`dump f` output a textual description of the current state into formatter `f`. (A debugging aid.)

```
method next_page : string
```

Returns the name of the designated next page.

```
method set_next_page : string -> unit
```

Sets the name of the designated next page

```
method event : Wd_types.event
```

returns the event that just has happened

```
method is_server_popup_request : bool
```

Returns whether someone invoked `set_server_popup_request` before. This is usually done if an dialog is restored for a server-driven popup window (tag `ui:server-popup`).

```
method serialize : (Wd_types.dialog_type, unit) Hashtbl.t -> Buffer.t -> unit
```

Writes the state of this dialog into the buffer. The hashtable is used in the recursive descent into the object hierarchy to remember which dialogs have already been serialized. Call the method initially with an empty hash table.

```
method unserialize :
  (Wd_types.universe_type, Wd_types.dialog_type, Wd_types.environment)
  Wd_types.poly_ds_buf -> unit
```

Sets the state of this dialog from the deserialization buffer which must contain previously serialized dialog.

```
method enter_session_scope : Wd_types.session_type -> unit
```

Tells the dialog the current session

```
method leave_session_scope : unit -> unit
```

Tells the dialog that it is no longer under control of this session

```
method session : Wd_types.session_type
```

Returns the current session. Fails when there is no session

`method environment : Wd_types.environment`

Return the environment of the current CGI activation

`method declaration : Wd_types.dialog_decl_type`

Return the declaration of this dialog

`method application : Wd_types.application_type`

Return the application

`method universe : Wd_types.universe_type`

Return the universe

104.0.2 The following methods must be defined in subclasses:

`method virtual prepare_page : unit -> unit`

This method is invoked just before a new output page is generated.

Preconditions:

The method `page_name` returns the name of this page. The method `event` still returns the action last happened, but the name of the page where this action happened is lost. It may be interesting whether the last event was `No_event` because this indicates that the page is the initial page of the dialog.

This method should set any variables which are necessary to generate the new page (mostly variable containing HTML fragments). Furthermore, any state that needs to be saved should be put into variables, too.

`method virtual handle : unit -> unit`

This method is invoked just after the user triggered an event (e.g. pressed a button).

Preconditions: The method `page_name` returns the name of the page that was visible while the event was triggered. The method `event` returns the description of the event.

Postconditions: The method may set the next page to display by invoking `set_next_page`; if it does not then the default will be used. The default is either specified by the XML element describing the interactor that triggered the event, or the default is otherwise this page again.

This method should modify the variables according to the event that happened, and optionally set the next page to display.

There is also an alternate way to go to another page: raising the exception Change_page.

By raising the exception Change_dialog this method may force to go to another dialog.

```
  end
```

This class contains the dialog instance

```
class type application_type =
  object
```

`method start_dialog_name : string`

Returns the name of the start dialog

`method dialog_names : string list`

Returns the names of the declared dialogs

`method dialog_declaration : string -> Wd_types.dialog_decl_type`

Returns the declaration for the passed dialog

`method template_names : string list`

Returns the names of the declared templates (including the templates of the core and the standard libraries)

`method template : string -> Wd_types.template_type`

Returns the definition of the passed template

`method study : unit -> unit`

Studies all defined templates

`method output_encoding : string -> string -> string`

Returns the output encoding as function `string -> string`

`method add_output_encoding : string -> (string -> string) -> unit`

Adds the output encoding function (second argument) under the passed name (first argument) to the application. It is not possible to redefine existing functions.

```
method var_function :
  string ->
  Wd_types.dialog_type ->
  Wd_types.dialog_type Wd_types.poly_var_value list ->
  Wd_types.dialog_type Wd_types.poly_var_value
```

Returns the variable function as O'Caml function. If the function has lazy arguments, this property will be lost when it is returned by this method. Use `lazy_var_function` instead.

```
method lazy_var_function :
  string ->
  Wd_types.dialog_type ->
  Wd_types.dialog_type Wd_types.poly_var_value Lazy.t list ->
  Wd_types.dialog_type Wd_types.poly_var_value
```

Returns the variable function as O'Caml function

`method dtd : Pxp_dtd.dtd`

Returns the DTD of WDialog

`method charset : Pxp_types.rep_encoding`

Returns the character set used for the internal representation, and for the generated HTML pages.

`method debug_mode : bool`

Returns whether there is a processing instruction `<?wd-debug-mode?>`. This value is used to initialize the environment.

`method debug_mode_style : Wd_types.debug_mode_style`

Returns the style of the debug mode

`method prototype_mode : bool`

Returns whether there is a processing instruction `<?wd-prototype-mode?>`. This value is used to initialize the environment.

```
method onstartup_call_handle : bool
```

Returns whether there is a processing instruction `<?wd-onstartup-call-handle?>`.

```
method add_var_function :
  string ->
  (Wd_types.dialog_type ->
   Wd_types.dialog_type Wd_types.poly_var_value list ->
   Wd_types.dialog_type Wd_types.poly_var_value) ->
  unit
```

Adds a variable function with eager evaluation

```
method add_lazy_var_function :
  string ->
  (Wd_types.dialog_type ->
   Wd_types.dialog_type Wd_types.poly_var_value Lazy.t list ->
   Wd_types.dialog_type Wd_types.poly_var_value) ->
  unit
```

Adds a variable function with lazy evaluation of its arguments

```
end
```

This class represents the whole application

```
class type template_type =
  object
```

```
method study : Wd_types.application_type -> unit
```

Studies the template for the scope of the passed application

```
method instantiate :
  ?context:Wd_types.syntax_tree_type Wd_types.dict ->
  ?vars:Wd_types.trans_vars ->
  ?params:Wd_types.syntax_tree_type Wd_types.dict ->
  Wd_types.dialog_type -> Wd_types.syntax_tree_type
```

Instantiates the template. The parameter `~context` may contain the context parameters (defaults to the empty set). `~vars` may contain the transformation variables (defaults to: not available). `~params` may contains the lexical parameters (defaults to the empty set). In every case, the current dialog instance must be passed.

The result of the instantiation is the XML tree where all dollar notations have been replaced by the passed actual parameter values. It is not necessary that the template is completely expanded, however, i.e. it may contain `ui:use` elements.

The exception `Instantiation_error` will be raised if something goes wrong.

```
end
```

This class represents a template definition

```
class type syntax_tree_type =
  object
```

```
inherit Pxp_document.extension
```

The syntax tree is an extension of the PXP model for XML trees

`inherit Wd_types.template_type` [104]

The syntax tree is always a template definition. This has formal reasons.

```
method scan_application : Wd_types.application_type -> unit
```

Scanner methods

The side-effect of the scanners is to put the result into the passed argument.

```
method scan_dialog :
  Wd_types.application_type -> Wd_types.dialog_decl_type -> unit
method scan_enumeration : Wd_types.enum_decl -> unit
method scan_literal : unit -> Wd_types.dialog_type Wd_types.poly_var_value
```

(But this one is functional)

```
method escaped_data : string
```

Returns the string contained in the data node after HTML-escaping has been applied. This string is cached.

```
method to_html :
  ?context:Wd_types.syntax_tree_type Wd_types.dict ->
  ?vars:Wd_types.trans_vars ->
  Wd_types.dialog_type -> Netchannels.out_obj_channel -> unit
```

This method writes the current node and all its children to the passed output channel. This method is for HTML context, i.e. character data are to be HTML-escaped, and all elements have to be interpreted.

```
method to_text :
  ?context:Wd_types.syntax_tree_type Wd_types.dict ->
  ?vars:Wd_types.trans_vars ->
  Wd_types.dialog_type -> Netchannels.out_obj_channel -> unit
```

This method writes the current node and all its children to the passed output channel. This method is for attribute context, i.e. character data remain as they are without escaping them, and only the elements are interpreted that are defined for attribute context.

```
  end
```

This class represents the XML syntax tree

```
class type universe_type =
  object

    method application : Wd_types.application_type
```

Return the application this universe is made for

```
    method register :
      string ->
      (Wd_types.universe_type ->
       string -> Wd_types.environment -> Wd_types.dialog_type) ->
      unit
```

Registers that a certain O'Caml class implements the callbacks of the dialog declared in the UI definition. More precisely,

```
u # register name f_new
```

registers that the dialog `name` is realized by the objects returned by the evaluation of the constructing function `f_new`. The arguments passed to `f_new` are:

```
f_new u name env
```

where `u` and `name` are the universe and the dialog name, and `env` is the environment of the current CGI activation. It is legal to store `env` in the created objects because they are created for every activation anew.

```
method create : Wd_types.environment -> string -> Wd_types.dialog_type
```

Creates a new object for the passed environment and dialog name. This method effectively calls the function `f_new` that has previously been registered.

```
end
```

The universe is the registry of classes and dialogs

```
class type session_manager_type =
  object

    method create : Wd_types.dialog_type -> Wd_types.session_type
```

Creates a new session for this dialog

```
    method unserialize :
      Wd_types.universe_type ->
      Wd_types.environment -> string -> Wd_types.session_type
```

Restores an old session for the passed universe and the passed environment. The string is the session in serialized form as returned by the `serialize` method.

```
  end
```

The session manager creates new sessions, and looks sessions up in the (possibly fictive) session database

```
class type session_type =
  object

    method session_id : string
```

Returns the ID of the session. Fails if the session does not have an ID (e.g. it is not stored in the database)

```
    method dialog_name : string
```

Returns the name of the dialog this session encapsulates

```
    method dialog : Wd_types.dialog_type
```

Returns the dialog

```
    method commit_changes : unit -> unit
```

Causes that `serialize` returns the current state of the dialog. This means that `commit_changes` extracts the state from the dialog, and prepares the string that will be returned by `serialize`.

`method serialize : string`

Returns the state of the dialog at the time of the last `commit_change` or the state of the initial dialog

`method change_dialog : Wd_types.dialog_type -> unit`

Continue with a new dialog. The methods `dialog_name` and `dialog` will immediately reflect the change. However, you have to call `commit_changes` to make `serialize` return the state of the new dialog.

```
  end
```

This is a single session

```
type var_value = dialog_type poly_var_value
```

This type is the concrete version of `Wd_types.poly_var_value`[104] that is actually used

```
type var_decl = dialog_type poly_var_decl
```

This type is the concrete version of `Wd_types.poly_var_decl`[104] that is actually used

```
type ds_buf = (universe_type, dialog_type, environment)
  poly_ds_buf
```

This type is the concrete version of `Wd_serialize.poly_ds_buf` that is actually used

`exception Change_dialog of dialog_type`

The implementation of the `handle` method of a dialog may raise `Change_dialog` to drop the current dialog and continue with another dialog.

`exception Change_page of string`

The implementation of the `handle` method of a dialog may raise `Change_page` to set the next page to display for the current dialog.

`exception Formal_user_error of string`

A formal error that happens independently of the current runtime state. The string argument explains the error.

`exception Runtime_error of string`

A certain operation cannot be performed because the current state does not fulfill the necessary preconditions. The string argument explains the error.

`exception No_such_variable of string`

It has been tried to access a non-declared variable. The string is the name of the variable.

`exception Instantiation_error of string`

An error generated by the `instantiate` method. The caller should catch this exception and report the error from its own view

Wd_serialize_types

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_serialize_types

105 WDialog API for Objective Caml: Wd_serialize_types

106 Module `Wd_serialize_types`

```
type ('a, 'b, 'c) poly_ds_buf = {
  ds_str : string ;
  mutable ds_pos : int ;
  ds_end : int ;
  ds_universe : 'a ;
  ds_environment : 'c ;
  ds_dialogs : (int, 'b) Hashtbl.t ;
}
```

Wd_universe

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_universe

107 WDialog API for Objective Caml: Wd_universe

108 Module `Wd_universe` : Contains the implementation of `Wd_types.universe_type`[104]

```
class universe : Wd_types.application_type -> Wd_types.universe_type
```

Implements the universe. A universe is always bound to a certain application, so you must pass this application on creation

Wd_upload

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_upload

109 WDialog API for Objective Caml: Wd_upload

110 Module `Wd_upload` : This module manages file upload parameters.

`type upload_manager`

Manages the file upload parameters of a certain request

`val get : upload_manager -> string -> Netcgi_types.cgi_argument`

Returns the CGI argument containing the file upload information for the file upload box with the passed name. This name is what is specified in the "name" attribute of ui:file.

See netcgi.mli for accessor functions for `cgi_argument` values.

This function raises `Not_found` if there is no ui:file box with the passed name. This function returns a pseudo argument with empty value, empty filename, and empty MIME type if the browser did not send the corresponding CGI argument to the server. (Note: At least Netscape browsers always send CGI arguments even if the user did not specify files to upload, and these arguments have empty value, empty filename, and empty MIME type. It is a good idea to check for an empty filename in order to find out whether the upload box was used or not.)

`val init : Wd_types.environment -> Wd_types.interactors -> upload_manager`

Scans the available CGI parameters for file uploads, and initializes this module.

- It is required that the CGI module is already initialized (in `environment`)
- The argument of this function is the interactor definition of the page that has been submitted.

Wd_var_functions

Web Path: WDialog / Reference / WDialog API (O'Caml) / Modules / Wd_var_functions

111 WDialog API for Objective Caml: Wd_var_functions

112 Module `Wd_var_functions` : This module defines the "variable functions" that can be called in `$[...]` expressions.

All these functions take formally a list of `var_value` arguments and return a `var_value`. Actually, they are often only defined for a certain number of arguments, and for arguments of certain type.

The functions raise a `Failure` exception if something is wrong.

```
val id :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns the first argument

```
val length :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns the length of the single string argument in characters.

```
val card :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns the cardinality of the single argument. The cardinality are the number of iterations <ui:iterate> or <ui:enumerate> would do for a variable with the given value:

- For a string: the number of words of the string
- For an enumerator: the number of elements
- For an association: the number of elements

```
val size :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns the size of the single argument, which may be a `String_value`, `Enum_value`, `Dyn_enum_value`, or an `Alist_value`. The returned size is encoded as `String_value`.

`size` **is deprecated! Use either** `length` **or** `card` **instead!**

```
val add :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Adds the numbers encoded as `String_value`, and returns the sum as `String_value`.

```
val sub :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Subtracts the following numbers from the first number. All numbers must be encoded as `String_value`. Returns the result as `String_value`.

```
val mul :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Multiplies the numbers encoded as `String_value`, and returns the product as `String_value`.

```
val div :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Divides the first number through the following numbers. All numbers must be encoded as `String_value`. Returns the result as `String_value`.

```
val modulo :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

The first number module the second number. Both numbers must be encoded as `String_value`. Returns the result as `String_value`.

```
val assoc :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Looks the second argument up in the first argument, which must be an `Alist_value`. The index, i.e. second argument, must be an `String_value`.

```
val nth :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns the nth element of the first argument which must be an `Alist_value`. The second argument is the position, a number encoded as `String_value`.

```
val contains :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns 1 if the first argument contains the second argument, and 0 otherwise. The second argument must be a string value. The function is defined as follows:

- If the first argument is a string, it is split up into a list of words, and it is tested whether the second string occurs in the list of words.
- If the first argument is a declared enumerator, it is tested whether the value includes the string as item.
- If the first argument is a dynamic enumerator, it is tested whether the value includes the string as internal item.
- If the first argument is an associative list, it is tested whether the value includes the string as index.

```
val mentions :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns 1 if the first argument contains the second argument, and 0 otherwise. The second argument must be a string value. The function is defined as follows:

- If the first argument is a dynamic enumerator, it is tested whether the value includes the string as external item.
- If the first argument is an associative list, it is tested whether the value includes the string as mapped value.

```
val translate :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Like ui:translate maps internal values of enumerators to external values. The function takes two arguments:

- `translate [dyn; int]`: The first argument is a dynamic enumerator, and the second argument is the internal value to translate to the external value
- `translate [type; int]`: The first argument denotes a declared enumerator (as a string), and the second argument is the internal value to translate to the external value

```
val rev_translate :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Maps external values to internal values:

- `rev_translate [dyn; int]`: The first argument is a dynamic enumerator, and the second argument is the external value to translate to the internal value
- `translate [type; int]`: The first argument denotes a declared enumerator (as a string), and the second argument is the external value to translate to the internal value

```
val eq :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Compares two strings, and returns 1 when equal and 0 when not equal

```
val ne :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Compares two strings, and returns 0 when equal and 1 when not equal

```
val match_ :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Checks whether the first string matches the regular expression in the second string, and returns 1 on match and 0 otherwise

```
val nomatch :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Checks whether the first string matches the regular expression in the second string, and returns 0 on match and 1 otherwise

```
val substring :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns a substring of the first argument, a `String_value`. The following forms are accepted:

- `substring [s; pos; length]`: The second argument is the position, and the third argument is the length. The length can be negative.
- `substring [s; pos]`: Only the position is passed. Returns the substring beginning at this position until the end of the string.

If the start position or end position of the substring is outside the possible range, only the part of the substring is returned that is inside the possible range.

```
val concat :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Concatenates all the `String_value` arguments, and returns the result as `String_value`.

```
val int_eq :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
val int_ne :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
val int_lt :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
val int_le :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
val int_gt :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
val int_ge :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

These functions compare two strings as integers, and return 1 if the condition is fulfilled, and 0 if not. The conditions are:

- int_eq: The integers are equal
- int_ne: The integers are not equal
- int_lt: The first integer is less than the second integer
- int_le: The first integer is less or equal than the second integer
- int_gt: The first integer is greater than the second integer
- int_ge: The first integer is greater or equal than the second integer

```
val int_min :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
val int_max :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

These functions compute the minimum/maximum of all integer arguments

```
val int_abs :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
val int_sign :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

These functions compute the absolute value/the sign of the first integer argument

```
val card_eq :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
val card_ne :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
val card_lt :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
val card_le :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
val card_gt :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
val card_ge :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

These functions compare the cardinality of the first argument with a string taken as integer, and return 1 if the condition is fulfilled, and 0 if not. The cardinality is defined as for the `card` function. The conditions are:

- int_eq: The cardinality is equal to the integer
- int_ne: The cardinality is not equal to the integer
- int_lt: The cardinality is less than the integer
- int_le: The cardinality is less or equal than the integer
- int_gt: The cardinality is greater than the integer
- int_ge: The cardinality is greater or equal than the integer

```
val height :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns the height of the `String_value` argument. The height is the number of lines, i.e. the number of LF/CR/CRLF occurrences plus 1.

```
val width :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns the width of the `String_value` argument. The width is the length of the longest line (not counting the line separators).

```
val dialog_exists :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

When called with only one argument: The function returns 1 if the argument is an existing dialog (i.e. `Dialog_value(Some _)`) and 0 if the dialog does not exist (i.e. `Dialog_value None`).

When called with two arguments (legacy mode): The first argument must be a dialog value as before, and the second argument must be either the string "yes" or the string "no". The function returns 1 if the existence of the dialog is equivalent to the boolean value of the string.

```
val and_ :
  Wd_types.dialog_type -> Wd_types.var_value Lazy.t list -> Wd_types.var_value
```

Calculates the logical AND operation for all its arguments: Returns 1 if all arguments are non-zero integers, and 0 otherwise.

```
val or_ :
  Wd_types.dialog_type -> Wd_types.var_value Lazy.t list -> Wd_types.var_value
```

Calculates the logical OR operation for all its arguments: Returns 1 if there is a non-zero integer as argument, and 0 otherwise.

```
val not_ :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Calculates the logical NOT operation of its single argument: Returns 1 if the argument is 0, and 0 if the argument is non-zero.

```
val true_ :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns the constant 1

```
val false_ :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns the constant 0

```
val if_ :
  Wd_types.dialog_type -> Wd_types.var_value Lazy.t list -> Wd_types.var_value
```

Evaluates the first argument (which must be an integer). If the integer is non-zero, the result is the second argument. If the integer is 0, the result is the third argument.

```
val var :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns the contents of the variable whose name is passed as first argument, a `String_value`.

```
val dialog :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns the name of the current dialog (no arguments)

```
val self :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns the current dialog (no arguments)

```
val page :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns the name of the current page (no arguments)

```
val language :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns the current language, or "" if none selected (no arguments)

```
val self_base_url :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns the URL pointing to the current script without session state

```
val session_id :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Returns the session ID without checksum

```
val create_anchor_event :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Add an anchor event and return the identifier

```
val create_xanchor_event :
  Wd_types.dialog_type -> Wd_types.var_value list -> Wd_types.var_value
```

Add an indexed anchor event and return the identifier

Runtime models

Web Path: WDialog / Reference / Runtime models

113 Runtime models

There are currently three ways of connecting a WDialog application to the outer world, especially to the Web: CGI, FastCGI, and JSERV. Note that the Perl bindings currently only support CGI.

Besides choosing from three acronyms, the runtime model determines how the resources of the operating system can be used. The question is whether two activations of the application run in the same process, or run in different processes, and how parallel accesses to the same runtime entities are resolved. Although we are focusing on the connection to the web (server), the runtime model also determines possible solutions to other connections, for example to database systems.

113.1 CGI

The CGI interface is well-known and available for almost all web servers. Furthermore, CGI defines a set of possible interactions between the web server and the application, and serves as a reference for what one can expect. Because of these reasons, CGI is the basic interface for WDialog.

CGI starts a new process for every request. This has the advantage that (1) the requests can be processed separately such that they do not interfer with each other, and that (2) it is ensured that the application gives all allocated resources (open files etc) back to the operating system. These two points are the reasons why CGI is still used today for critical applications, although there is a performance bottleneck as every process must be initialized anew.

There are some wrong legends about CGI. For example, some people think that the fact that CGI uses the `fork` and `exec` system calls makes it slow from the very beginning, especially when the binary to start has a size of several megabytes. This is not the problem. Modern Unix-based operating system are heavily optimized regarding `fork` and `exec`, and an experiment showed that my old 400 Mhz system can start 30 CGI processes per second without causing high CPU load; the process image was bigger than one megabyte. Actually, the problem with CGI is that loading the process image is not all of the initialization work. WDialog must parse the XML file containing the UI definition, and it must prepare the XML tree for the transformation. These actions may take more than a second for big applications.

Nevertheless, there is a way to reduce the initialization time significantly, and this makes CGI interesting again. The idea is to avoid parsing the XML file by loading preformatted binary data instead. You can create the binary representation by calling the program `wd-xmlcompile` which is part of the WDialog distribution:

```
wd-xmlcompile sample.ui
```

- this would create `sample.ui.bin`, and the loader of WDialog automatically finds this file and loads it instead of `sample.ui`. This trick often reduces the load time to less than 0.5 seconds.

In order to run the application as CGI, call the function `Wd_run_cgi.run` from your main program - it does all the rest.

113.2 FastCGI

The FastCGI protocol is an extension of the CGI model which allows multiple requests to be processed by the same process. The CGI application either runs in an application server environment provided by the web server (the most

common method) or as a stand alone daemon listening for FastCGI connection. The details of the fastcgi protocol, including instructions on how to set it up for various web servers can be found at *The FastCGI Project*[9] web page.

Using fastcgi in WDialog is accomplished by calling the function `Wd_run_fastcgi.serv`. This function implements a run loop which processes connections sequentially.

113.2.1 Concurrency

Many different forms of concurrency are possible with fastcgi, and in most cases very little needs to be done to make the WDialog application aware of it. This is especially true in the web server managed environment, where the web server generally implements a process pool model in which it runs N copies of the application at startup time. Requests are then routed to each process by the web server in an implementation dependant way. As long as session state is in some sort of shared store the application need not even be aware that it is operating in a concurrent environment. Threads are also possible in two cases. You may either start multiple threads, and have each one call `Wd_run_fastcgi.serv`, or you may use threads to perform background tasks which do NOT talk on the fastcgi output channels. No multiplexing of output is possible over a single connection to the web server. The first thread model is very similar to the process pool model, except that a shared session manager need not be used. For an external application, one which does not use the web server as a process manager, concurrency is left completely up to the application.

113.3 JSERV

The JSERV protocol was developed by the *Java Apache Project*[10], and is still be used by *Jakarta* [11]. Although these projects base on the Java language, the protocol as such is language-independent, and it turns out that it is very simple to connect a JSERV-enabled web server with a servlet engine that is not written in Java.

The Java Apache Project is dead, no further development takes place, as all subprojects have moved to Jakarta. Nevertheless, I currently recommend to use mod_jserv, the JSERV extension for Apache 1.3 from the Java Apache Project, because it is much simpler to extract it from the whole software project. However, mod_jk works, too.

The architecture behind JSERV is quite simple. The web server is extended with the JSERV protocol, and every request opens a new connection to the servlet process. This process is a permanently running daemon. The web server forwards the page request over this connection to the servlet process, and the latter processes it and sends the answer back to the web server. Effectively, the servlet process behaves like a second web server behind the first, but it does not support the full HTTP protocol but the simpler and less general JSERV protocol.

In the original Java environment, the servlet process is a JVM (Java virtual machine), and it executes the code of the application. There is also a part handling the JSERV protocol, but this is simply a library that can be loaded like any other library. - The Java background explains why the servlet process is permanently running: CGI is not a choice for Java, because of the long startup time of the JVM. Furthermore, Java's excellent multi-threading capabilities makes it possible to handle concurrency inside the JVM.

That the servlet process is permanently running is the important advantage for the O'Caml port, too. The servlet process is simply an O'Caml program that uses the library for JSERV (which is included in the *Ocamlnet*[12] package). However, there are differences to the Java original:

- The servlets are not dynamically loaded. A normal, pre-linked program is used. This means that you must shutdown the servlet process before you can exchange a servlet by a newer version, or add a servlet.
- Instead of multi-threading, a range of execution models is supported. One reason is that multi-threading is not always adequate, another reason is that the multi-threading support in O'Caml is not as good as in Java. The models are:

[9] (URL: http://www.fastcgi.com)
[10] (URL: http://java.apache.org/)
[11] (URL: http://jakarta.apache.org/)
[12] (URL: link-ocamlnet;)

- `'Sequential`: Serial execution in a single process
- `'Forking`: Every request spawns a new process
- `'Process_pool`: Requests are forwarded to a process pool
- `'Thread_pool`: Requests are processed by a thread pool

The latter model is not yet implemented!

The various models are discussed in detail below.

WDialog provides the module `Wd_run_jserv` that defines request handlers for the various execution models. A sample main program for a servlet process would be:

```
let req_hdl = Wd_run_jserv.create_request_handler ... () in
let server = 'Forking(20, [ "appname", req_hdl ]) in
Netcgi_jserv.jvm_emu_main
  (Netcgi_jserv_app.run server 'Ajp_1_2)]
```

The real main program is `Netcgi_jserv.jvm_emu_main` which accepts command-line arguments that are compatible (enough) with the arguments of the Java JVM. (Useful, because the JSERV web server extension usually starts the servlet process, and the web server assumes that it starts a JVM.)

The function `Netcgi_jserv_app.run` is the main entry point for the JSERV protocol handler. It gets as argument the `server` definition, here of `'Forking` type. The list defines that the servlet `appname` is handled by `req_hdl`, the WDialog-specific request handler.

In order to get the servlet server running, you also need the `jserv.properties` file containing the configurations that are needed by both the web server and the servlet server. Furthermore, `httpd.conf`, the configuration file of Apache, must be extended with some `mod_jserv`-specific definitions. You can find more information in the Java Apache distribution.

113.3.1 The JSERV execution model 'Sequential

Sequential execution means that a single process gets all arriving requests which are processed one after another. This works very well unless it takes too long to process a request. The big advantage of this execution model is that there is almost no management overhead to handle concurrent accesses, because these do not happen. However, if the computations for a request last very long, the server will block until this time-consuming request is done, and any other requests happening at the same time must wait.

There is another advantage: It is quite simple to cache frequently accessed data, because these can be stored in global variables, again without any additional overhead.

The sequential model is very attractive for web applications that have a limited number of concurrent users, and that run on single-CPU systems. However, some care must be taken to avoid that individual requests block the whole application. For example, one possibility would be to set the alarm clock (`Unix.alarm` or `Unix.setitimer`) and to raise an exception after the maximum period of time has expired.

113.3.2 The JSERV execution model 'Forking

In this model, every incoming request causes that the main process spawns a subprocess. The subprocess performs all computations that are necessary to reply, while the main process continues immediately accepting new connections.

This model sounds like CGI, but it is actually different in one important aspect. When the subprocess is spawned, all necessary initializations have already happened, and the subprocess can immediately begin to analyze the request, and to

do all the other work related to the request. In contrast to this, the CGI subprocess must first initialize itself, for example read the XML file containing the UI definition.

Effectively, the setup time is longer than for `'Sequential` execution, but still rather short. The concurrently running activations are isolated from each other like in CGI, and the operating system takes care to deallocate the resources when the activation is over. There is no simple way to let the activations share data or other resources.

113.3.3 The JSERV execution model 'Process_pool

This model combines the advantages of `'Forking` and `'Sequential`, and is probably the most attractive model for highly loaded servers. At startup time, a fixed number of processes are spawned (after initialization), and every process of this pool accepts sequentially the incoming requests. When a new page request arrives it is likely that some of the processes of the pool are currently busy and that the rest is idle. One of the free processes will get the request, and will be busy until the request is processed.

It may happen that all processes are busy. The newly arrived request must wait until one of the processes is free again. (Note: The length of this queue can be specified by the backlog parameter.)

This model can process requests in parallel; however, parallelism is restricted to the fixed number that must be known at startup time. A good choice is the number of CPUs times a small factor, but there should be enough memory such that no process is swapped out.

Furthermore, this model avoids the cost of forking for every request, because the processes run sequentially once started. This results in very low overhead and quick responses.

However, this model also combines the disadvantages of `'Forking` and `'Sequential`, because the activations for the requests are neither isolated from each other nor they are not isolated, you simply do not know.

113.3.4 The JSERV execution model 'Thread_pool

You may wonder why I do not simply follow the Java original and only implement thread pools. There are a number of arguments against multi-threading, some critising this technique in general, some only applying to the O'Caml implementation.

- Multi-threading requires a lot of programming discipline. In general, the whole code must be reentrant, and special means like mutexes, condition variables etc. must be used to ensure that never two threads interfer with each other in an uncontrolled manner. Unfortunately, there are no tools (like type checkers) that enforce these rules, the programmer must do it himself. Furthermore, it is very difficult to find the errors by testing the programs, because the problems often have the character of race conditions that only happen in rare cases. But if you have a lot of seldom occurring races, the stability of the whole program certainly decreases significantly.
- The O'Caml implementation of multi-threading had some serious bugs in the past, although it was programmed by an outstanding expert. You may take this as a proof of the previous thesis, but it also means that O'Caml has not been used very often for multi-threaded programming (otherwise these errors would have been found earlier), and that the stability cannot (yet) be trusted for production applications.
- Last but not least, the O'Caml implementation has the fundamental restriction that it cannot take advantage from several CPUs, even if the underlying multi-threading library of the operating system supports this.

I hope this explains why the multi-threaded execution model does not rank as number 1 in the priority list. However, there are benefits from such a model.

Most important, this is the only model that can combine parallelism with the ability to easily access shared data structures. The other models (`'Forking` and `'Process_pool`) can share data only by special means of the operating system (e.g. by sharing files, or by shared memory that is now available in the `bigarray` library).

Furthermore, it becomes possible to program servers that respond to multiple protocols. For example, such a server could combine a web frontend with RPC services. (Like EJB, but I do not see a strict necessity to do that. Both aspects can be separated.)

No summary yet, as the model is not yet implemented.

113.4 Which model is the right one for me?

Obviously, there is no simple answer. I have tried to enumerate the pros and cons for all the models, and it depends on your application which arguments count. Maybe the following simplifications point you to the right direction for your evaluation.

- *Development:* In this phase of a project the CGI protocol is the best choice. You need not to restart servers to test a new version of your program.
- *Best compromise:* If stability and speed both count, I can only recommend JSERV with `'Forking` processes. The processes are isolated from each other, and are properly cleaned up, so you do not have to care about these issues. It is still fast enough for the majority of applications.
- *Maximum performance:* That's very simple, the `'Process_pool` is your friend. It allows parallelism almost without performance costs. For *maximum* performance, I would additionally recommend to install the web server and the JSERV engine on different systems. For *MAXIMUM* performance, I would further recommend to install several instances of the JSERV engine on several systems, and to use JSERV's load-balancing feature to drive them. The architecture is scalable, isn't it.
- *Flexibility:* Once implemented, `'Thread_pool` is probably the most flexible solution. In the meantime, you may consider to use `'Sequential`, and to start threads for background activitities. (Unfortunately, multithreading is not possible for forked processes because of limitations of O'Caml implementation, so you cannot start threads in the worker processes of `'Process_pool`.)

113.5 Secondary network connections

It is often necessary to open network connections to further services in order to process a request. For example, accessing database systems is nowadays done in this way. You have several choices for that:

- You can open a new connection for every activation, and close it afterwards. This isolates the accesses best, but this may cause performance problems.
- An alternative for `'Process_pool` is to open only one connection per process, and to use it for all requests performed by the process. For database systems with transactions, a reasonable degree of isolation can be achieved by closing the current transaction between requests. Note that the configuration parameter of `Netcgi_jserv_app.run` provides the two hooks `js_init_process` and `js_fini_process` that are called for every process to initialize and for every process to finalise, respectively. So these functions can open and close the database connection.