Templates Parser

$Revision: 1.35 $
Date: 15 March 2003

Pascal Obry (p.obry@wanadoo.fr)

Table of Contents

1 Introduction

First of all this package is distributed under the GNAT modified GNU GPL.

The templates parser package has been designed to parse files and to replace some specific tags into these files by some specified values.

The main goal was to ease the development of Web servers. In CGI (*Common Gateway Interface*) mode you have to write the HTML page in the program (in Ada or whatever other languages) by using some specific libraries or by using only basic output functions like Ada `Put_Line` for example. This is of course not mandatory but by lack of a good library every Web development end up doing just that.

The main problems with this approach are:

- It is painful to have to recompile the program each time you have a slight change to do in the design (center an image, change the border width of a table...)
- You have the design and the program merged together. It means that to change the design you must know the Ada language. And to change the Ada program you need to understand what is going on with all these inline HTML command.
- You can't use the nice tools to generate your HTML.

With the templates parser package these problems are gone. The code and the design is **completely** separated. This is a very important point. PHP or JSP have tried this but most of the time you have the script embedded into the Web template. And worst you need to use another language just for your Web development.

- The HTML page is separated from the program code. Then you can change the design without changing the code. Moreover when you fix the code you don't have to handle all the specific HTML output. And you do not risk to break the design.
- It is easier to work on the design and the program at the same time using the right peoples for the job.
- It reduce the number of *edit/build/test* cycles. Writing HTML code from a program is error prone.
- It is possible to use standard tools to produce the HTML.
- You don't have to learn a new language.
- The script is Ada, so here you have the benefit of all the Ada power.

In fact, now the Ada program just compute some values, get some data from a database or whatever and then call the templates parser to output a page with the data displayed. To the templates parser you just pass the template filename and an associative table.

It is even more convenient to have different display with the same set of data. You just have to provides as many templates as you like.

2 What is a tag ?

A tag is a string found in the template page and surrounded by a specific set of characters. The default is `@_` at the start and `_@` at the end of the tag. This default can be changed using `Set_Tag_Separators` routine, see Chapter 6 [Templates_Parser Ada spec], page 20. Note that it must be changed as the first API call and should not be changed after that.

The tag will be replaced by a value specified in a translation table.

For example with the template file '`demo.tmplt`':

```

<P>Name @_NAME_@

```

Using the following code '`demo.adb`':

```
with Ada.Text_IO;
with Templates_Parser;

procedure Demo is

   Translations : Templates_Parser.Translate_Table :=
     (1 => Templates_Parser.Assoc ("NAME", "Ada"));

begin
   Ada.Text_IO.Put_Line
     (Templates_Parser.Parse ("demo.tmplt", Translations));
end Demo;
```

The program will print out :

```
<P>Name Ada
```

This is a very simple example, but you'll see that there is a lot of powerful construct that you can use into a template file.

3 Variable tags

3.1 Discrete, Boolean, Vector and Matrix

A variable tag is a specific string to be replaced by the template parser. There is four kinds of variable tags: **discrete**, **boolean**, **vector** and **matrix**. All variables tags are build using the Assoc constructors, see Chapter 6 [Templates_Parser Ada spec], page 20.

`discrete`
: We have already seen the discrete variable tag. This is a variable which has only one value. This value will replace the tag in the template file. Discrete variable constructors are provided for String, Unbounded_String and Integer (see Assoc routines).

`boolean`
: A boolean tag as a value of TRUE or FALSE. This value will replace the tag in the template file. These tags can also be used with the IF tag statement.

`vector`
: A vector tag is a variable which represent a set of values. These kind of variables will be used with the TABLE tag statement see Section 4.4 [TABLE tag statement], page 9. Outside a table statement it will be replaced by all values in the vector tag.

  There is many overloaded constructors to build vector tags (see "+" operators). The "+" operators are used to build a Vector_Tag item from standards types like String, Unbounded_String, Character, Integer and Boolean.

  To add items to a vector tag many overloaded operators are provided (see "&" operators). The "&" operators add one item at the end of the vector, it is possible to add directly String, Unbounded_String, Character, Integer and Boolean items using one of the overloaded operator.

  A Vector tag composed of only boolean value TRUE or FALSE is called a Boolean Vector tag. This tag is to be used with a IF tag statement inside a TABLE tag statement.

`matrix`
: A matrix tag is a variable which represent a set of vector tags. These kind of variables will be used inside a TABLE tag statement embedded into another TABLE tag statement (i.e. a TABLE tag statement of level 2).

  A matrix variable tag is built by concatenating vector variables.

3.2 Filters and Attributes

All kinds of variable tag can have one or more function-prefix or filter. The function prefix is applied to the variable value. The syntax is `@_[[FILTER_NAME[(parameter)]:]FILTER_NAME[(parameter)]:]SOME_VAR_@`. Filters are evaluated from right to left.

Vector and Matrix tag can also have attributes. Attributes are placed after the tag name and preceded with a simple quote. `@_SOME_VAR['ATTRIBUTE_NAME]_@`. It is possible to use filters and attributes together. In that case the attribute is first evaluated and the result is passed-through the filters.

3.2.1 Filters

The current supported filters are:

`"+"(`*N*`)` or `ADD(`*N*`)`
: Add N to variable and return the result. If the current variable value is not a number it returns the empty string. N must be a number.

`"-"(`*N*`)` or `SUB(`*N*`)`
: Subtract N to variable and return the result. If the current variable value is not a number it returns the empty string. N must be a number.

`"*"(`*N*`)` or `MULT(`*N*`)`
: Multiply N with variable and return the result. If the current variable value is not a number it returns the empty string. N must be a number.

`"/"(`*N*`)` or `DIV(`*N*`)`
: Divide variable by N and return the result. If the current variable value is not a number it returns the empty string. N must be a number.

`BR_2_LF`
: Replaces all occurrences of the `<br>` HTML tag by a LF (Line-Feed) character.

`CAPITALIZE`
: Put all characters in the variable in lower case except characters after a space or an underscore which are set in upper-case.

`CLEAN_TEXT`
: Keep only letters and digits all others characters are changed to spaces.

`COMA_2_POINT`
: Replaces all comas by points.

`CONTRACT`
: Converts any suite of spaces by a single space character.

`EXIST`
: Returns **True** if variable is set and has a value different that the null string and **False** otherwise.

`FORMAT_NUMBER`
: Returns the number with a space added between each 3 digits blocks. The decimal part is not transformed. If the data is not a number nothing is done.

`IS_EMPTY`
: Returns **True** if variable is the empty string and **False** otherwise.

`LF_2_BR`
: Replaces all occurrences of the character LF (Line-Feed) by a `<br>` HTML tag.

`LOWER`
: Put all characters in the variable in lower-case.

`MATCH(`*REGEXP*`)`
: Returns **True** if variable match the regular expression passed as filter's parameter.

`MOD(`*N*`)`
: Returns variable modulo N. If the current variable value is not a number it returns the empty string. N must be a number.

`NO_DIGIT`
: Replaces all digits by spaces.

`NO_LETTER`
: Replaces all letters by spaces.

`NO_SPACE`
: Removes all spaces in the variable.

`OUI_NON`
: If variable value is **True** it returns **Oui**, if **False** it returns **Non**, otherwise does nothing. It keeps the way **True/False** is capitalized (all upper, all lower or first letter capital).

`POINT_2_COMA`
: Replaces all comas by points.

`REPEAT(`*N*`)`
: Returns N times the variable, N being passed as filter's parameter.

`REVERSE`
: Reverse the string.

`SIZE`
: Returns the size (number of characters) of the string value.

`TRIM`
: Removes leading and trailing spaces.

`UPPER`
: Put all characters in the variable in upper-case.

`WEB_ESCAPE`
: Replaces characters '<', '>', '"' and '&' by corresponding HTML sequences: &lt; &gt; &quot; and &amp;

`WEB_NBSP`
: Replaces all spaces by an HTML non breaking space.

`YES_NO`
: If variable value is **True** it returns **Yes**, if **False** it returns **No**, otherwise does nothing. It keeps the way **True/False** is capitalized (all upper, all lower or first letter capital).

For example:

```
If VAR is set to "vector_tag" then:


@_VAR_@                       ->  vector_tag
@_UPPER:VAR_@                 ->  VECTOR_TAG
@_CAPITALIZE:VAR_@            ->  Vector_Tag
@_EXIST:VAR_@                 ->  TRUE
@_UPPER:REVERSE:VAR_@         ->  GAT_ROTCEV
@_MATCH(VEC.*):UPPER:VAR_     ->  TRUE
```

3.2.2 Attributes

Current supported attributes are:

`V'length`
: Returns the number of item in the vector.

`M'Line`
: Returns the number of line in the matrix.

`M'Min_Column`
: Returns the size of smallest vector in the matrix.

`M'Max_Column`
: Returns the size of largest vector in the matrix.

For example:

```
If VEC is set to "<1 , 2>" and MAT to "<a, b, c> ; <2, 3, 5, 7>" then:


@_VEC'Length_@              ->  2
@_ADD(3):VEC'Length_@       ->  5
@_MAT'Line_@                ->  2
@_MAT'Min_Column_@          ->  3
@_MAT'Max_Column_@          ->  4
```

3.3 Other variables tags

There is some specific variables tags that can be used in any templates. Here is a description of them:

`YEAR`
: Current year number using 4 digits.

`MONTH`
: Current month number using 2 digits.

`DAY`
: Current day number using 2 digits.

`HOUR`
: Current hour using range 0 to 23 using 2 digits.

`MINUTE`
: Current minute using 2 digits.

`SECOND`
: Current seconds using 2 digits.

`MONTH_NAME`
: Current full month name (January .. December).

`DAY_NAME`
: Current full day name (Monday .. Sunday).

4 Tag statements

There is three different tag statements. A tag statement is surrounded by `@@`. The tag statements are:

4.1 Comments tag statement

Every line starting with `@@`– are comments and are completly ignored by the parser. The resulting page will have the exact same format and number of lines with or without the comments.

```
@@-- This template is used to display the client's data
@@-- It uses the following tags:
@@--
@@--    @_CID_@       Client ID
@@--    @_ITEMS_V_@   List of items (vector tag)

<P>Client @_CID_@

...
```

4.2 INCLUDE tag statement

This tag is used to include another template file. This is useful if you have the same header and/or footer in all your HTML pages. For example:

```
@@INCLUDE@@ header.tmplt

<P>This is by Web page

@@INCLUDE@@ footer.tmplt
```

It is also possible to pass arguments to the include file. These parameters are put after the include filename. It is possible to reference these parameters into the included file with the special variable names `@_$<n>_@`, where *n* is the include's parameter indice (0 is the include filename, 1 the first parameter and so on).

```
@@INCLUDE@@ another.tmplt @_VAR_@ azerty
```

In file '`another.tmplt`'

`@_$0_@`	is another.tmplt
`@_$1_@`	is the variable `@_VAR_@`
`@_$2_@`	is the string "azerty"

If an include variable references a non existing include parameter the tag is kept as-is.

4.3 IF tag statement

This is the conditional tag statement. The complete form is:

```
@@IF@@ <expression1>
   part1
@@ELSIF@@ <expression2>
   part2
@@ELSE@@
   part3
@@END_IF@@
```

The part1 one will be parsed if expression1 evaluate to "TRUE", part2 will be parsed if expression2 evaluate to "TRUE" and the part3 will be parse in any other case. The ELSIF and ELSE part are optional.

The expression here is composed of boolean variable (or conditional variable) and/or boolean expression. Recognized operators are:

`A = B`	Returns TRUE if A equal B
`A /= B`	Returns TRUE if A is not equal B
`A > B`	Returns TRUE if A greater than B. If A and B are numbers it returns the the number comparison (5 > 003 = TRUE) otherwise it returns the string comparison (“5” > “003” = FALSE).
`A >= B`	Returns TRUE if A greater than or equal to B. See above for rule about numbers.
`A < B`	Returns TRUE if A lesser than B. See above for rule about numbers.
`A <= B`	Returns TRUE if A lesser than or equal to B. See above for rule about numbers.
`A and B`	Returns TRUE if A and B is TRUE and FALSE otherwise.
`A or B`	Returns TRUE if A or B is TRUE and FALSE otherwise.
`A xor B`	Returns TRUE if either A or B (but not both) is TRUE and FALSE otherwise.
`not A`	Returns TRUE if either A is FALSE and FALSE otherwise.

The default evaluation order is done from left to right, all operators having the same precedence. To build an expression it is possible to use the parentheses to change the evaluation order. A value with spaces must be quoted as a string. So valid expressions could be:

```
@@IF@@ (@_VAR1_@ > 3) or (@_COND1_@ and @_COND2_@)

@@IF@@ not (@_VAR1_@ > 3) or (@_COND1_@ and @_COND2_@)

@@IF@@ (@_VAR1_@ > 3) and not @_COND1_@

@@IF@@ @_VAR1_@ = "a value"
```

Note also that variables and values can be surrounded by quotes if needed. Quotes are needed if a value contain spaces.

To generate a conditional variable tag it is possible to use the following **Templates_Parser** function:

```
function Assoc (Variable  : in String;
                Value     : in Boolean;
                return Association;
-- build an Association (Variable = Value) to be added to a
-- Translate_Table. It set the variable to TRUE or FALSE depending on
-- Value.
```

Let’s see an example using an IF tag statement. With the following template:

```
@@IF@@ @_USER_@
   <P>As a user you have a restricted access to this server.
@@ELSE@@
   <P>As an administrator you have full access to this server.
@@END_IF@@
```

The following program:

```
with Ada.Text_IO;
with Templates_Parser;

procedure User1 is

   Translations : Templates_Parser.Translate_Table :=
     (1 => Templates_Parser.Assoc ("USER", True));

begin
   Ada.Text_IO.Put_Line
     (Templates_Parser.Parse ("user.tmplt", Translations));
end User1;
```

Will display:

```
   <P>As a user you have a restricted access to this server.
```

But the following program:

```
with Ada.Text_IO;
with Templates_Parser;

procedure User2 is

   Translations : Templates_Parser.Translate_Table :=
     (1 => Templates_Parser.Assoc ("USER", False));

begin
   Ada.Text_IO.Put_Line
     (Templates_Parser.Parse ("user.tmplt", Translations));
end User2;
```

Will display:

```
   <P>As an administrator you have full access to this server.
```

4.4 TABLE tag statement

A table tag is useful to generate HTML table for example. Basically the code between the `@@TABLE@@` and `@@END_TABLE@@` will be repeated as many time as the vector tag will have of values. If many vector tags are specified in a table statement, the code between the table will be repeated a number of time equal to the maximum length of all vector tags in the table tag statement.

A table tag statement is a kind of implicit iterator. This is a very important concept to build HTML table. Using `Vector_Tag` or `Matrix_Tag` variable in a `@@TABLE@@` tag statement it is possible to build very complex Web pages.

Syntax:

```
@@TABLE@@ [@@TERMINATE_SECTIONS@@]
   ...
[@@SECTION@@]
   ...
@@END_TABLE@@
```

Let's have an example. With the following template:

```


<P>Here is the ages of some peoples:

<TABLE>
@@TABLE@@
   <TR>
   <TD>@_NAME_@
   <TD>@_AGE_@
@@END_TABLE@@
</TABLE>

```

And the following program:

```
with Ada.Text_IO;
with Templates_Parser;

procedure Table is

   use type Templates_Parser.Vector_Tag;

   Names : constant Templates_Parser.Vector_Tag
     := +"Bob" & "Bill" & "Toto";
   Ages  : constant Templates_Parser.Vector_Tag
     := +"10" & "30" & "5";

   Translations : Templates_Parser.Translate_Table :=
     (1 => Templates_Parser.Assoc ("NAME", Names),
      2 => Templates_Parser.Assoc ("AGE", Ages));

begin
   Ada.Text_IO.Put_Line
     (Templates_Parser.Parse ("table.tmplt", Translations));
end Table;
```

The following output will be generated:

```
<P>Here is the ages of some peoples:

<TABLE>
   <TR>
   <TD>Bob
   <TD>10
   <TR>
   <TD>Bill
   <TD>30
   <TR>
   <TD>Toto
   <TD>5
</TABLE>
```

Note that we use vector tag variables here. A discrete variable tag in a table will be replaced by the same (the only one) value for each row. A vector tag outside a table will be displayed as a list of values, each value being separated by a specified string. The default is a comma and a space ", ".

The complete prototype for the `Vector_Tag` Assoc function is:

```
   function Assoc (Variable  : in String;
                   Value     : in Vector_Tag;
                   Separator : in String    := Default_Separator;
                   return Association;
–  build an Association (Variable = Value) to be added to a
–  Translate_Table. This is a vector tag association, value is a
–  Vector_Tag. If the vector tag is found outside a table tag statement
–  it is returned as a single string, each value being separated by the
–  specified separator.
```

A table can contain many sections. The section to use will be selected depending on the current line. For example, a table with two sections will use different data on even and odd lines. This is useful, for example, when you want to alternate the line background color for a better lisibility when working on HTML pages.

A table with sections can have the modifier `@@TERMINATE_SECTIONS@@`. This ensure that the table output will end with the last section. If the number of data in the vector variable tag is not a multiple of the number of sections then the remaining section will be complete with empty tag value.

```
<P>Here are some available computer devices:

<TABLE>
@@TABLE@@
   <TR BGCOLOR=#F00>
   <TD>@_DEVICES_@
   <TD>@_PRICES_@
@@SECTION@@
   <TR BGCOLOR=#00F>
   <TD>@_DEVICES_@
   <TD>@_PRICES_@
@@END_TABLE@@
</TABLE>

<TABLE>
@@TABLE@@ @@TERMINATE_SECTIONS@@
   <TR>
   <TD BGCOLOR=#00F WIDTH=10>
   <TD WIDTH=150>@_DEVICES_@
@@SECTION@@
   <TD WIDTH=150>@_DEVICES_@
@@SECTION@@
   <TD WIDTH=150>@_DEVICES_@
   <TD BGCOLOR=#00F WIDTH=10>
@@END_TABLE@@
</TABLE>
```

And the following program:

```
with Ada.Text_IO;
with Templates_Parser;

procedure Table_Section is

   use type Templates_Parser.Vector_Tag;

   Devices : constant Templates_Parser.Vector_Tag
     := +"Screen" & "Keyboard" & "Mouse" & "Hard Drive";
   Prices  : constant Templates_Parser.Vector_Tag
     := +"$500" & "$20" & "$15" & "$140";

   Translations : Templates_Parser.Translate_Table :=
     (1 => Templates_Parser.Assoc ("DEVICES", Devices),
      2 => Templates_Parser.Assoc ("PRICES", Prices));

begin
   Ada.Text_IO.Put_Line
     (Templates_Parser.Parse ("table_section.tmplt", Translations));
end Table_Section;
```

The following output will be generated:

```
<P>Here are some available computer devices:

<TABLE>
   <TR BGCOLOR=#F00>
   <TD>Screen
   <TD>$500
   <TR BGCOLOR=#00F>
   <TD>Keyboard
   <TD>$20
   <TR BGCOLOR=#F00>
   <TD>Mouse
   <TD>$15
   <TR BGCOLOR=#00F>
   <TD>Hard Drive
   <TD>$140
</TABLE>

<TABLE>
   <TR>
   <TD BGCOLOR=#00F WIDTH=10>
   <TD WIDTH=150>Screen
   <TD WIDTH=150>Keyboard
   <TD WIDTH=150>Mouse
   <TD BGCOLOR=#00F WIDTH=10>
   <TR>
   <TD BGCOLOR=#00F WIDTH=10>
   <TD WIDTH=150>Hard Drive
   <TD WIDTH=150>
   <TD WIDTH=150>
   <TD BGCOLOR=#00F WIDTH=10>
</TABLE>
```

Into a table construct there are some additional variable tags available:

`@_UP_TABLE_LINE_@`
: This tag will be replaced by the table line number of the upper table statement. It will be set to 0 outside a table statement or inside a single table statement.

`@_TABLE_LINE_@`
: This tag will be replaced by the current table line number. It will be replaced by 0 outside a table statement.

`@_NUMBER_LINE_@`
: This is the number of line displayed in the table. It will be replaced by 0 outside a table statement.

`@_TABLE_LEVEL_@`
: This is the table level number. A table construct declared in a table has a level value of 2. It will be replaced by 0 outside a table statement.

Let’s have a look at a more complex example with mixed IF and TABLE tag statement.

Here is the template:

```
Hello here are a list of devices:

<table>
<tr>
<th>Device Name
<th>Price
<th>Order

@@TABLE@@
<tr>
<td>@_DEVICES_@
<td>@_PRICES_@

<td>
@@IF@@ @_AVAILABLE_@
<a href="/order?DEVICE=@_DEVICES_@">Order
@@ELSE@@
Sorry, not available
@@END_IF@@

@@END_TABLE@@
```

And the following program:

```
with Ada.Text_IO;
with Templates_Parser;

procedure Table_If is

   use type Templates_Parser.Vector_Tag;

   function In_Stock (Device : in String) return Boolean;
   --  Complex function. Does a SQL access to the right database to know if
   --  the Device is available and thus can be ordered.

   procedure Add (Device, Price : in String);
   --  Add the device into the list to be displayed.

   Devices   : Templates_Parser.Vector_Tag;
   Prices    : Templates_Parser.Vector_Tag;
   Available : Templates_Parser.Vector_Tag;

   ---------
   -- Add –
   ---------

   procedure Add (Device, Price : in String) is
   begin
      Devices := Devices & Device;
      Prices  := Prices & Price;
      Available := Available & In_Stock (Device);
   end Add;

   --------------
   -- In_Stock –
   --------------

   function In_Stock (Device : in String) return Boolean is
   begin
      if Device = "Keyboard" then
         return True;
      else
         return False;
      end if;
   end In_Stock;

   Translations : Templates_Parser.Translate_Table (1 .. 3);

begin
   Add ("Screen", "$500");
   Add ("Keyboard", "$15");
   Add ("Mouse", "$15");
   Add ("Hard Drive", "$140");

   Translations := (Templates_Parser.Assoc ("DEVICES", Devices),
                    Templates_Parser.Assoc ("PRICES", Prices),
                    Templates_Parser.Assoc ("AVAILABLE", Available));

   Ada.Text_IO.Put_Line
     (Templates_Parser.Parse ("table_if.tmplt", Translations));
end Table_If;
```

The following output will be generated:

```


Hello here are a list of devices:

<table>
<tr>
<th>Device Name
<th>Price
<th>Order

<tr>
<td>Screen
<td>$500

<td>
Sorry, not available

<tr>
<td>Keyboard
<td>$15

<td>
<a href="/order?DEVICE=Keyboard">Order

<tr>
<td>Mouse
<td>$15

<td>
Sorry, not available

<tr>
<td>Hard Drive
<td>$140

<td>
Sorry, not available


```

Table tag statement can also be used with matrix tag. There is 3 possibles placements for a matrix tag:

1. Inside a table of level 2 (a TABLE tag statement inside a TABLE tag statement).

   In this case the first TABLE iterate through the matrix lines. First iteration will use the first matrix's vector, second iteration will use second matrix's vector and so on. And the second TABLE will be use to iterate through the vector's values.

2. Inside a table of level 1.

   In this case the TABLE iterate through the matrix lines. First iteration will use the first matrix's vector, second iteration will use second matrix's vector and so on. Each vector is then converted to a string by concatenating all values and using the supplied separator (see Assoc constructor for Matrix_Tag).

3. Outside a table statement.

In this case the matrix is converted to a string. Each line represent a vector converted as a string using the supplied separator (see point 2 above), and each vector is separated by an ASCII.LF character.

Let's look at an example, with the following template:

```
A matrix inside a table of level 2:

@@TABLE@@
<tr>
@@TABLE@@
<td>
@_MAT_@
</td>
@@END_TABLE@@
</tr>

@@END_TABLE@@

The same matrix inside a single table:

@@TABLE@@
<tr>
<td>
@_MAT_@
</tr>

@@END_TABLE@@

The same matrix outside a table:

@_MAT_@
```

Using the program:

```
with Ada.Text_IO;
with Templates_Parser;

procedure Matrix is

   package TP renames Templates_Parser;

   use type TP.Vector_Tag;
   use type TP.Matrix_Tag;

   V1 : constant TP.Vector_Tag := +"A1.1" & "A1.2";
   V2 : constant TP.Vector_Tag := +"A2.1" & "A2.2";
   V3 : constant TP.Vector_Tag := +"A3.1" & "A3.2";

   M  : constant TP.Matrix_Tag := +V1 & V2 & V3;

begin
   Ada.Text_IO.Put_Line
     (TP.Parse ("matrix.tmplt",
                TP.Translate_Table'(1 => TP.Assoc ("MAT", M))));
end Matrix;
```

We get the following result:

```
A matrix inside a table of level 2:

<tr>
<td>
A1.1
</td>
<td>
A1.2
</td>
</tr>

<tr>
<td>
A2.1
</td>
<td>
A2.2
</td>
</tr>

<tr>
<td>
A3.1
</td>
<td>
A3.2
</td>
</tr>


The same matrix inside a single table:

<tr>
<td>
A1.1, A1.2
</tr>

<tr>
<td>
A2.1, A2.2
</tr>

<tr>
<td>
A3.1, A3.2
</tr>


The same matrix outside a table:

A1.1, A1.2
A2.1, A2.2
A3.1, A3.2
```

5 Last notes

The templates parser has be written to parse HTML page but it is usable with any king of files. There is nothing hard coded for HTML, it is then possible to use it with plain text files, XML files, SGML files or whatever as long as it is not a binary file.

All tag statements can be mixed together. A `table` tag statement can be put in a `if` tag statement. An `if` tag statement can be put in a `table` tag statement. Idem for the `include` tag statement.

Download page is `http://perso.wanadoo.fr/pascal.obry/`.

6 Templates_Parser Ada spec

```
------------------------------------------------------------------------------
--                            Templates Parser                              --
--                                                                          --
--                         Copyright (C) 1999 - 2001                        --
--                               Pascal Obry                                --
--                                                                          --
--  This library is free software; you can redistribute it and/or modify    --
--  it under the terms of the GNU General Public License as published by    --
--  the Free Software Foundation; either version 2 of the License, or (at   --
--  your option) any later version.                                         --
--                                                                          --
--  This library is distributed in the hope that it will be useful, but     --
--  WITHOUT ANY WARRANTY; without even the implied warranty of              --
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU       --

--  General Public License for more details.                                --
--                                                                          --
--  You should have received a copy of the GNU General Public License       --
--  along with this library; if not, write to the Free Software Foundation, --
--  Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.          --
--                                                                          --
--  As a special exception, if other files instantiate generics from this   --
--  unit, or you link this unit with other files to produce an executable,  --
--  this  unit  does not  by itself cause  the resulting executable to be   --
--  covered by the GNU General Public License. This exception does not      --
--  however invalidate any other reasons why the executable file  might be  --
--  covered by the  GNU Public License.                                     --
------------------------------------------------------------------------------

--  $Id: templates_parser.ads,v 1.29 2002/12/19 18:12:19 obry Exp $

with Ada.Finalization;
with Ada.Strings.Unbounded;

package Templates_Parser is

   use Ada.Strings.Unbounded;

   Template_Error : exception;
```

```
Default_Begin_Tag : constant String := "@_";
Default_End_Tag   : constant String := "_@";

Default_Separator : constant String := ", ";

procedure Set_Tag_Separators
  (Start_With : in String := Default_Begin_Tag;
   Stop_With  : in String := Default_End_Tag);
--  Set the tag separators for the whole session. This should be changed as
--  the very first API call and should not be changed after.

----------------
-- Vector Tag --
----------------

type Vector_Tag is private;
--  A vector tag is a set of strings. Note that this object is using a
--  by-reference semantic. A reference counter is associated to it and
--  the memory is realeased when there is no more reference to it.

function "+" (Value : in String) return Vector_Tag;
--  Vector_Tag constructor.

function "+" (Value : in Character) return Vector_Tag;
--  Vector_Tag constructor.

function "+" (Value : in Boolean) return Vector_Tag;
--  Vector_Tag constructor.

function "+" (Value : in Unbounded_String) return Vector_Tag;
--  Vector_Tag constructor.

function "+" (Value : in Integer) return Vector_Tag;
--  Vector_Tag constructor.

function "&"
  (Vect  : in Vector_Tag;
   Value : in String)
   return Vector_Tag;
--  Add Value at the end of the vector tag set.

function "&"
  (Vect  : in Vector_Tag;
   Value : in Character)
   return Vector_Tag;
--  Add Value at the end of the vector tag set.

function "&"
  (Vect  : in Vector_Tag;
   Value : in Boolean)
   return Vector_Tag;
--  Add Value (either string TRUE or FALSE) at the end of the vector tag
--  set.

function "&"
```

```
  (Vect  : in Vector_Tag;
   Value : in Unbounded_String)
   return Vector_Tag;
--  Add Value at the end of the vector tag set.

function "&"
  (Vect  : in Vector_Tag;
   Value : in Integer)
   return Vector_Tag;
--  Add Value (converted to a String) at the end of the vector tag set.

procedure Clear (Vect : in out Vector_Tag);
--  Removes all values in the vector tag. Current Vect is not released but
--  the returned object is separated (not using the same reference) from
--  the original one.

function Size (Vect : in Vector_Tag) return Natural;
--  Returns the number of value into Vect.

function Item (Vect : in Vector_Tag; N : in Positive) return String;
--  Returns the Nth Vector Tag's item. Raises Constraint_Error if there is
--  no such Item in the vector (i.e. vector length < N).

----------------
-- Matrix Tag –
----------------

type Matrix_Tag is private;
--  A matrix tag is a set of vectors. Note that this object is using a
--  by-reference semantic. A reference counter is associated to it and
--  the memory is realeased when there is no more reference to it.

function "+" (Vect : in Vector_Tag) return Matrix_Tag;
--  Matrix_Tag constructor. It returns a matrix with a single row whose
--  value is Vect.

function "&"
  (Matrix : in Matrix_Tag;
   Vect   : in Vector_Tag)
   return Matrix_Tag;
--  Returns Matrix with Vect added to the end.

function Size (Matrix : in Matrix_Tag) return Natural;
--  Returns the number of Vector_Tag (rows) inside the Matrix.

function Vector (Matrix : in Matrix_Tag; N : in Positive) return Vector_Tag;
--  Returns Nth Vector_Tag in the Matrix.

-----------------------
-- Association table –
-----------------------

type Association is private;

type Translate_Table is array (Positive range <>) of Association;
```

```
No_Translation : constant Translate_Table;

function Assoc
  (Variable  : in String;
   Value     : in String)
   return Association;
--  Build an Association (Variable = Value) to be added to a
--  Translate_Table. This is a standard association, value is a string.

function Assoc
  (Variable  : in String;
   Value     : in Unbounded_String)
   return Association;
--  Build an Association (Variable = Value) to be added to a
--  Translate_Table. This is a standard association, value is an
--  Unbounded_String.

function Assoc
  (Variable  : in String;
   Value     : in Integer)
   return Association;
--  Build an Association (Variable = Value) to be added to a
--  Translate_Table. This is a standard association, value is an Integer.
--  It will be displayed without leading space if positive.

function Assoc
  (Variable  : in String;
   Value     : in Boolean)
   return Association;
--  Build an Association (Variable = Value) to be added to a
--  Translate_Table. It set the variable to TRUE or FALSE depending on
--  value.

function Assoc
  (Variable  : in String;
   Value     : in Vector_Tag;
   Separator : in String     := Default_Separator)
   return Association;
--  Build an Association (Variable = Value) to be added to a
--  Translate_Table. This is a vector tag association, value is a
--  Vector_Tag. If the vector tag is found outside a table tag statement
--  it is returned as a single string, each value being separated by the
--  specified separator.

function Assoc
  (Variable  : in String;
   Value     : in Matrix_Tag;
   Separator : in String     := Default_Separator)
   return Association;
--  Build an Association (Variable = Value) to be added to a
--  Translate_Table. This is a matrix tag association, value is a
--  Matrix_Tag. If the matrix tag is found outside of a 2nd level table tag
--  statement, Separator is used to build string representation of the
--  matrix tag's vectors.

-----------------------------
```

```
   -- Parsing and Translating -
   -----------------------------

   function Parse
     (Filename          : in String;
      Translations      : in Translate_Table := No_Translation;
      Cached            : in Boolean         := False;
      Keep_Unknown_Tags : in Boolean         := False)
      return String;
   --  Parse the Template_File replacing variables' occurrences by the
   --  corresponding values. If Cached is set to True, Filename tree will be
   --  recorded into a cache for quick retrieval. If Keep_Unknown_Tags is set
   --  to True then tags that are not in the translate table are kept
   --  as-is if it is part of the template data. If this tags is part of a
   --  condition (in an IF statement tag), the condition will evaluate to
   --  False.

   function Parse
     (Filename          : in String;
      Translations      : in Translate_Table := No_Translation;
      Cached            : in Boolean         := False;
      Keep_Unknown_Tags : in Boolean         := False)
      return Unbounded_String;
   --  Idem as above but returns an Unbounded_String.

   function Translate
     (Template     : in String;
      Translations : in Translate_Table := No_Translation)
      return String;
   --  Just translate the discrete variables in the Template string using the
   --  Translations table. This function does not parse the command tag
   --  (TABLE, IF, INCLUDE). All Vector and Matrix tag are replaced by the
   --  empty string.

   procedure Print_Tree (Filename : in String);
   --  Use for debugging purpose only, it will output the internal tree
   --  representation.

private

   ------------------
   --  Vector Tags -
   ------------------

   type Vector_Tag_Node;
   type Vector_Tag_Node_Access is access Vector_Tag_Node;

   type Vector_Tag_Node is record
      Value : Unbounded_String;
      Next  : Vector_Tag_Node_Access;
   end record;

   type Integer_Access is access Integer;

   type Vector_Tag is new Ada.Finalization.Controlled with record
      Ref_Count : Integer_Access;
      Count     : Natural;
      Head      : Vector_Tag_Node_Access;
      Last      : Vector_Tag_Node_Access;
   end record;

   type Vector_Tag_Access is access Vector_Tag;

   procedure Initialize (V : in out Vector_Tag);
   procedure Finalize   (V : in out Vector_Tag);
   procedure Adjust     (V : in out Vector_Tag);

   ------------------
   --  Matrix Tags –
   ------------------

   type Matrix_Tag_Node;

   type Matrix_Tag_Node_Access is access Matrix_Tag_Node;

   type Matrix_Tag_Node is record
      Vect : Vector_Tag;
      Next : Matrix_Tag_Node_Access;
   end record;

   type Matrix_Tag_Int is new Ada.Finalization.Controlled with record
      Ref_Count : Integer_Access;
      Count     : Natural; -- Number of vector
      Min, Max  : Natural; -- Min/Max vector's sizes
      Head      : Matrix_Tag_Node_Access;
      Last      : Matrix_Tag_Node_Access;
   end record;

   type Matrix_Tag is record
      M : Matrix_Tag_Int;
   end record;

   procedure Initialize (M : in out Matrix_Tag_Int);
   procedure Finalize   (M : in out Matrix_Tag_Int);
   procedure Adjust     (M : in out Matrix_Tag_Int);

   ------------------
   --  Association –
   ------------------

   type Var_Kind is (Std, Vect, Matrix);

   type Association (Kind : Var_Kind := Std) is record
      Variable  : Unbounded_String;

      case Kind is
         when Std =>
            Value : Unbounded_String;

         when Vect =>
            Vect_Value : Vector_Tag;
            Separator  : Unbounded_String;
         when Matrix =>
            Mat_Value        : Matrix_Tag;
            Column_Separator : Unbounded_String;
      end case;
   end record;

   No_Translation : constant Translate_Table
     := (2 .. 1 => Association'(Std,
                                Null_Unbounded_String,
                                Null_Unbounded_String));

end Templates_Parser;
```