Templates Parser User's Guide

$Revision: 1.57 $
Date: 6 November 2004

Pascal Obry (pascal@obry.org)

http://www.obry.org

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 Appendix A [Templates_Parser API Reference], page 25. 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>Hello @_NAME_@
```

Using the following code 'demo.adb':

```
with Ada.Text_IO;
with Templates_Parser;

procedure Demo is

   Translations : constant 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>Hello 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 Composite

A variable tag is a specific string to be replaced by the template parser. There is three kinds of variable tags: **discrete**, **boolean**, **composite**. All variables tags are built using the `Assoc` constructors, see Appendix A [Templates_Parser API Reference], page 25.

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.

composite (Tag)
A composite tag is a variable which contains a set of values. These kind of variables will be used with the `TABLE` tag statement see Section 5.4 [TABLE tag statement],

page 12. Outside a table statement it will be replaced by all values concatenated with a specified separator. See `Set_Separator` routine. Such tag are variables declared on the Ada program a `Templates_Parser.Tag` type.

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

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

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

It is possible to build a composite tag having any number of nested level. A vector is a composite tag with only one level, a matrix is a composite tag with two level (a Tag with a set of vector tag).

Two aliases exists for composite tags with one or two nested level, they are named `Vector_Tag` and `Matrix_Tag`. In the suite of the document, we call *vector tag* a tag with a single nested level and *matrix tag* a tag with two nested level.

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.

Composite tags 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 or a discrete tag variable whose value is 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 or a discrete tag variable whose value is 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 or a discrete tag variable whose value is 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 or a discrete tag variable whose value is a number.

`ABS`
: Returns the absoute value.

`ADD_PARAM(NAME[=VALUE])`
: Add a parameter into an URL. This routine adds the '?' and '&' character if needed. *VALUE* can be a tag variable name.

`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.

`DEL_PARAM(NAME)`
: Delete parameter NAME from the URL. This routine removes the '?' and '&' character if needed. Returns the input string as-is if the parameter is not found.

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

`FORMAT_DATE(`**FORMAT**`)`
: Returns the date with the given format. The date must be in the ISO format (YYYY-MM-DD) eventually followed by a space and the time with the format HH:MM:SS. If the date is not given in the right format it returns the date as-is. The format is using the GNU/date description patterns as also implemented in `GNAT.Calendar.Time_IO`.

Characters:

%	a literal %
n	a newline
t	a horizontal tab

Time fields:

%H	hour (00..23)

%I hour (01..12)

%k hour (0..23)

%l hour (1..12)

%M minute (00..59)

%p locale's AM or PM

%r time, 12-hour (hh:mm:ss [AP]M)

%s seconds since 1970-01-01 00:00:00 UTC (a nonstandard extension)

%S second (00..59)

%T time, 24-hour (hh:mm:ss)

Date fields:

%a locale's abbreviated weekday name (Sun..Sat)

%A locale's full weekday name, variable length (Sunday..Saturday)

%b locale's abbreviated month name (Jan..Dec)

%B locale's full month name, variable length (January..December)

%c locale's date and time (Sat Nov 04 12:02:33 EST 1989)

%d day of month (01..31)

%D date (mm/dd/yy)

%h same as %b

%j day of year (001..366)

%m month (01..12)

%U week number of year with Sunday as first day of week (00..53)

%w day of week (0..6) with 0 corresponding to Sunday

%W week number of year with Monday as first day of week (00..53)

%x locale's date representation (mm/dd/yy)

%y last two digits of year (00..99)

%Y year (1970...)

By default, date pads numeric fields with zeroes. GNU date recognizes the following nonstandard numeric modifiers:

- (hyphen)
do not pad the field

_ (underscore)
pad the field with spaces

`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. The regular expression is using a format as found in '`gawk`', '`sed`' or '`grep`' tools.

`MAX(`*`N`*`)`

Returns the maximum value between the variable and the parameter.

`MIN(`*`N`*`)`

Returns the minimum value between the variable and the 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 or a discrete tag variable whose value is a number.

`NEG`

Change the sign of the value.

`NO_CONTEXT`

This is a special command filter which indicates that the tag must not be searched in the context. See see Section 6.1 [Context], page 24. `NO_CONTEXT` must be the first filter. This filter returns the value as-is.

`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. N must be a number or a discrete tag variable whose value is a number.

`REPLACE(`*`REGEXP`*`[/`*`STR`*`])`

This filter replaces **\n** (where *n* is a number) *STR*'s occurences by the corresponding match from *REGEXP*. The first match in *REGEXP* will replace **\1**, the second match **\2** and so on. Each match in *REGEXP* must be parenthesized. It replaces only the first match. *STR* is an optional parameter, its default value is **\1**. *STR* can be a tag variable name.

`REPLACE_ALL(`*REGEXP*`[/`*STR*`])`
: Idem as above but replaces all occurences.

`REPLACE_PARAM(`*NAME[=VALUE]*`)`
: This is filter is equivalent to ADD_PARAM(*NAME[=VALUE]*):DEL_PARAM(*NAME*). *VALUE* can be a tag variable name.

`REVERSE`
: Reverse the string.

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

`SLICE(`*x*` .. `*y*`)`
: Returns the sub-string starting from position x and ending to position y. Note that the string to slice always start from position 1.

`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: < > " and &

`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", ONE to "1" and TWO to "2" 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
@_SLICE(1..6):VAR_@                        ->  vector
@_REPLACE(([^_]+)):VAR_@                   ->  vector
@_REPLACE(([a-z]+)_([a-z]+)/\2_\1):VAR_@   ->  tag_vector
@_"+"(TWO):ONE_@                           ->  3
@_"-"(TWO):ONE_@                           ->  -1
```

3.2.2 Attributes

Current supported attributes are:

`V'length`
: Returns the number of item in the composite tag (can be applied only to a composite tag having a single nested level - a vector).

`M'Line`

Returns the number of line in the composite tag. This is identical to 'Length but can be applied only to a composite tag having two nested level - a matrix).

`M'Min_Column`

Returns the size of smallest composite tag in M composite tag. This attribute can be applied only to a composite tag having two nested level - a matrix.

`M'Max_Column`

Returns the size of largest composite tag in M composite tag. This attribute can be applied only to a composite tag having two nested level - a 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 variable tags

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

`NOW`

Current date and time with format "YYYY-MM-DD HH:MM:SS".

`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 Translations

Associations between variable tags and the template tag names are created with one of the `Assoc` routines. This set of associations are used by the parser (`Parse` routine). There is two kinds of associations set:

Translate_Table

an array of associations, this is easy to use when the number of associations is known at the creation time.

Translate_Set

a set of associations, it is possible to insert as many associations as needed into this object.

Note that this difference is only for users, the Templates_Parser engine uses only `Translate_Set` objects internally as it is much more efficient.

5 Tag statements

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

5.1 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_@

...
```

5.2 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.

Note that it is possible to pass the include parameters using names, a set of positional parameters can be pass first, so all following include command are identical:

```
@@INCLUDE@@ another.tmplt one two three four "a text"
@@INCLUDE@@ another.tmplt (one, two, 3 => three, 4 => four, 5 => "a text")
@@INCLUDE@@ another.tmplt (one, 5 => "a text", 3 => three, 2 => two, 4 => four)
```

5.3 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 : constant 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 : constant 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.
```

5.4 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 tables. Using a composite tag variable in a @@TABLE@@ tag statement it is possible to build very complex Web pages.

Syntax:

```
@@TABLE@@ [@@TERMINATE_SECTIONS@@]
      ...
   [@@BEGIN@@]
      ...
   [@@SECTION@@]
      ...
   [@@END@@]
      ...
@@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 : constant 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 separator. The default is a comma and a space ", ".

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

```
function Assoc (Variable  : in String;
                Value     : in Tag;
                Separator : in String    := Default_Separator;
                return Association;
--  Build an Association (Variable = Value) to be added to Translate_Table.
--  This is a tag association. Separator will be used when outputting the
--  a flat representation of the Tag (outside a table statement).
```

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 when you want to alternate the line background color for a better readability 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=#FF0000>
   <TD>@_DEVICES_@
   <TD>@_PRICES_@
@@SECTION@@
   <TR BGCOLOR=#00000F>
   <TD>@_DEVICES_@
   <TD>@_PRICES_@
@@END_TABLE@@
</TABLE>

<TABLE>
@@TABLE@@ @@TERMINATE_SECTIONS@@
   <TR>
   <TD BGCOLOR=#00000F WIDTH=10>
   <TD WIDTH=150>@_DEVICES_@
@@SECTION@@
   <TD WIDTH=150>@_DEVICES_@
@@SECTION@@
   <TD WIDTH=150>@_DEVICES_@
   <TD BGCOLOR=#00000F 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 : constant 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=#FF0000>
   <TD>Screen
   <TD>$500
   <TR BGCOLOR=#00000F>
   <TD>Keyboard
   <TD>$20
   <TR BGCOLOR=#FF0000>
   <TD>Mouse
   <TD>$15
   <TR BGCOLOR=#00000F>
   <TD>Hard Drive
   <TD>$140
</TABLE>

<TABLE>
   <TR>
   <TD BGCOLOR=#00000F WIDTH=10>
   <TD WIDTH=150>Screen
   <TD WIDTH=150>Keyboard
   <TD WIDTH=150>Mouse
   <TD BGCOLOR=#00000F WIDTH=10>
   <TR>
   <TD BGCOLOR=#00000F WIDTH=10>
   <TD WIDTH=150>Hard Drive
   <TD WIDTH=150>
   <TD WIDTH=150>
   <TD BGCOLOR=#00000F WIDTH=10>
</TABLE>
```

It is important to note that is is possible to avoid code duplication by using the `@@BEGIN@@` and `@@END@@` block statements. In this case only the code inside the block is part of the section, code outside is common text to all sections. Here is an example to generate an HTML table with different colors for each line:

The template file above can be written this way:

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

<TABLE>
@@TABLE@@
   <TR BGCOLOR=
   @@BEGIN@@
      "#FF0000"
   @@SECTION@@
      "#000000F"
   @@END@@
   >
   <TD>@_DEVICES_@
   <TD>@_PRICES_@
@@END_TABLE@@
</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.Tag;
   Prices    : Templates_Parser.Tag;
   Available : Templates_Parser.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 statements can also be used with matrix tag or more nested tag variables. In this case, for a tag variable with N nested level, the Nth closest enclosing `TABLE` tag statement will be used for the corresponding indices. If there is not enough indices, the last axis are just streamed as a single text value.

Let's see what happen 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` iterates 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` iterates 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 using the specified separator (see Assoc constructor for Tag or `Set_Separator` routine).

3. Outside a table statement.

   In this case the matrix is converted to a string. Each line represents a vector converted as a string using the supplied separator (see point 2 above), and each vector is separated by an ASCII.LF character. The separators to use for each level can be specified using `Set_Separator`.

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
```

6 Other services

6.1 Context

The context object can be used to dynamically handle tags. Such context can be passed to the `Parse` routines. If a template's tag is not found in the `Translate_Table` the context callback method is called by the parser. The default callback method does nothing, it is up to the user to define it. The callback procedure is defined as follow:

```
procedure Callback
  (Context  : access Templates_Parser.Context;
   Variable : in     String;
   Result   :    out Unbounded_String;
   Found    :    out Boolean);
```

`Result` must be set with the value for the tag `Variable` and `Found` set to `True`, otherwise `Found` can be set to `False`.

6.2 Tag utils

The child package `Utils`, see Section A.3 [Templates_Parser.Utils], page 32 contains a routine to encode a Tag variable into a string and the inverse routine that build a Tag given it's string representation. This is usefuln for example, in the context of AWS to store a Tag into a session variable. See AWS project.

6.3 XML representation

The child package `XML`, see Section A.4 [Templates_Parser.XML], page 33 contains routines to save a `Translation_Set` into an XML document or to create a `Translation_Set` by loading an XML document. The XML document must conforms to a specific `DTD` (see Ada spec file).

6.4 Debug

A set of routine to help debug the `Templates_Parser` engine, see Section A.2 [Templates_Parser.Debug], page 31. For example, `Debug.Print_Tree` will display, to the standard output, a representation of the internal semantic tree for a template file.

7 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 an `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://www.obry.org/`.

Appendix A Templates_Parser API Reference

A.1 Templates_Parser

```
------------------------------------------------------------------------------
--                              Templates Parser                            --
--                                                                          --
--                           Copyright (C) 1999 - 2004                      --
--                                 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 1.43 2004/11/03 16:23:07 obry Exp $

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

with Strings_Maps;

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.

   -----------------
   -- Generic Tag --
   -----------------

   type Tag is private;
   --  A tag is using a by reference semantic

   function "+" (Value : in String)           return Tag;
   function "+" (Value : in Character)        return Tag;
```

```
function "+" (Value : in Boolean)          return Tag;
function "+" (Value : in Unbounded_String) return Tag;
function "+" (Value : in Integer)          return Tag;
function "+" (Value : in Tag)              return Tag;
--  Tag constructors

function "&" (T : in Tag; Value : in String)           return Tag;
function "&" (T : in Tag; Value : in Character)        return Tag;
function "&" (T : in Tag; Value : in Boolean)          return Tag;
function "&" (T : in Tag; Value : in Unbounded_String) return Tag;
function "&" (T : in Tag; Value : in Integer)          return Tag;
function "&" (T : in Tag; Value : in Tag)              return Tag;
--  Add Value at the end of the tag

function "&" (Value : in String;           T : in Tag) return Tag;
function "&" (Value : in Character;        T : in Tag) return Tag;
function "&" (Value : in Boolean;          T : in Tag) return Tag;
function "&" (Value : in Unbounded_String; T : in Tag) return Tag;
function "&" (Value : in Integer;          T : in Tag) return Tag;
--  Add Value at the front of the tag

procedure Set_Separator (T : in out Tag; Separator : in String);
--  Set separator to be used when building a flat representation of
--  a composite tag.

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

function Size (T : in Tag) return Natural;
--  Returns the number of value into T

function Item (T : in Tag; N : in Positive) return String;
--  Returns the Nth Tag's item. Raises Constraint_Error if there is
--  no such Item in T (i.e. T length < N).

function Composite (T : in Tag; N : in Positive) return Tag;
--  Returns the Nth Tag's item. Raises Constraint_Error if there is
--  no such Item in T (i.e. T length < N).

subtype Vector_Tag is Tag;
subtype Matrix_Tag is Tag;

------------------
-- Associations --
------------------

type Association is private;

Null_Association : constant Association;

type Association_Kind is (Std, Composite);
--  The kind of association which is either Std (a simple value), a vector
--  tag or a Matrix tag.

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 Tag;
   Separator : in String := Default_Separator)
   return Association;
--  Build an Association (Variable = Value) to be added to Translate_Table.
--  This is a tag association. Separator will be used when outputting the
--  a flat representation of the Tag (outside a table statement).

function Get (Assoc : in Association) return Tag;
--  Returns the Tag in Assoc, raise Constraint_Error if Assoc is not
--  containing a Tag (Association_Kind is Std).

---------------------------
-- Association table/set --
---------------------------

type Translate_Table is array (Positive range <>) of Association;
--  A table with a set of associations, note that it is better to use
--  Translate_Set below as it is more efficient.

No_Translation : constant Translate_Table;

type Translate_Set is private;
--  This is a set of association like Translate_Table but it is possible to
--  insert item into this set more easily, furthermore there is no need to
--  know the number of item before hand. This is the object used internally
--  by the templates engine as it is far more efficient to retrieve a
--  specific item from it.

procedure Insert (Set : in out Translate_Set; Item : in Association);
--  Add Item into the translate set. If an association for this variable
--  already exists it just replaces it by the new item.

procedure Insert (Set : in out Translate_Set; Items : in Translate_Set);
--  Add Items into the translate set. If an association for variables in
--  Items already exists it just replaces it by the new one.

procedure Remove (Set : in out Translate_Set; Name : in String);
--  Removes association named Name from the Set. Does nothing if there is
--  not such association in the set.

function Get (Set : in Translate_Set; Name : in String) return Association;
```

```
--  Returns the association named Name in the Set. Returns Null_Association
--  is no such association if found in Set.

function Exists
  (Set      : in Translate_Set;
   Variable : in String) return Boolean;
--  Returns True if an association for Variable exists into the Set

generic
   with procedure Action
     (Item : in     Association;
      Quit : in out Boolean);
procedure For_Every_Association (Set : in Translate_Set);
--  Iterates through all associations in the set, call Action for each one.
--  Set Quite to True to stop the iteration.

function To_Set (Table : in Translate_Table) return Translate_Set;
--  Convert a Translate_Table into a Translate_Set

---------------
-- Callbacks --
---------------

type Context is tagged private;
type Context_Access is access all Context'Class;

procedure Callback
  (Context  : access Templates_Parser.Context;
   Variable : in     String;
   Result   :    out Unbounded_String;
   Found    :    out Boolean);
--  Callback is called by the Parse routines below if a tag variable was not
--  found in the set of translations. This routine must then set Result with
--  the value to use for Variable (name of the variable tag) and in this
--  case Found must be set to True. If Variable is not handled in this
--  callback, Found must be set to False. This default implementation will
--  always return with Found set to False.

Null_Context : constant Context_Access;

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

function Parse
  (Filename          : in String;
   Translations      : in Translate_Table := No_Translation;
   Cached            : in Boolean         := False;
   Keep_Unknown_Tags : in Boolean         := False;
   Context           : in Context_Access  := Null_Context)
   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;
   Context           : in Context_Access  := Null_Context)
```

```
      return Unbounded_String;
   --  Idem but returns an Unbounded_String

   function Parse
     (Filename          : in String;
      Translations      : in Translate_Set;
      Cached            : in Boolean        := False;
      Keep_Unknown_Tags : in Boolean        := False;
      Context           : in Context_Access := Null_Context)
      return String;
   --  Idem with a Translation_Set

   function Parse
     (Filename          : in String;
      Translations      : in Translate_Set;
      Cached            : in Boolean        := False;
      Keep_Unknown_Tags : in Boolean        := False;
      Context           : in Context_Access := Null_Context)
      return Unbounded_String;
   --  Idem with a Translation_Set

   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 composite tags are replaced by the empty string.

   function Translate
     (Template     : in String;
      Translations : in Translate_Set)
      return String;
   --  Idem with a Translation_Set

private
   --  implementation removed
end Templates_Parser;
```

A.2 Templates_Parser.Debug

```
------------------------------------------------------------------------------
--                              Templates Parser                            --
--                                                                          --
--                           Copyright (C) 2004                             --
--                              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-debug.ads,v 1.1 2004/05/09 08:18:48 obry Exp $

package Templates_Parser.Debug is

   procedure Print (T : in Tag);
   --  Print tag representation

   procedure Print_Tree (Filename : in String);
   --  Print tree for template Filename

end Templates_Parser.Debug;
```

A.3 Templates_Parser.Utils

```
------------------------------------------------------------------------------
--                              Templates Parser                            --
--                                                                          --
--                           Copyright (C) 2004                             --
--                              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-utils.ads,v 1.1 2004/05/18 19:45:59 obry Exp $

package Templates_Parser.Utils is

   function Image (T : in Tag) return String;
   --  Returns a string representation for this tag

   function Value (T : in String) return Tag;
   --  Give a string representation of a tag (as encoded with Image above),
   --  build the corresponding Tag object. Raises Constraint_Error if T is
   --  not a valid tag representation.

end Templates_Parser.Utils;
```

A.4 Templates_Parser.XML

```
------------------------------------------------------------------------------
--                              Templates Parser                            --
--                                                                          --
--                           Copyright (C) 2004                             --
--                              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-xml.ads,v 1.4 2004/05/19 19:34:22 obry Exp $

--  This API provides a way to save a Translate_Set as an XML document.
--  There is special rules to know about composite tags.
--
--  Composite tags :
--
--     If a tag named TAG exists, then the name TAG_DESCRIPTION is used as a
--     description for this specific tag.
--
--  Composite tags (more than one nested level)
--
--     If a tag named TAG exists, then the names TAG_DIM[n]_LABELS is used as
--     a set of labels for the tag's nth axis. In this case TAG_DIM[n]_LABELS
--     must be a vector tag, each entry corresponds to a label on this
--     axis. Also TAG_DIM[n]_DESCRIPTION is used as a description for this
--     axis.
--
--  Here is the DTD :
--
--  <?xml version="1.0" encoding="UTF-8"?>
--  <!--Description of a tag or dimension (ex: year)-->
--  <!ELEMENT Description (#PCDATA)>
--  <!--a dimension-->
--  <!ELEMENT Dim (Description, Labels)>
--  <!ATTLIST Dim
--      n CDATA #REQUIRED
--  >
--  <!--entry of a CompositeTag-->
--  <!ELEMENT Entry (ind+, V)>
--  <!--label of an indice of a dimension (ex: 2000)-->
--  <!ELEMENT Label (#PCDATA)>
--  <!ATTLIST Label
```

```
--      ind CDATA #REQUIRED
--  >
--  <!--list of labels of one dimension (ex: 1999, 2000, 2001)-->
--  <!ELEMENT Labels (Label+)>
--  <!--alias and information-->
--  <!ELEMENT Tag (Name, Description)>
--  <!--tagged data to be published in templates-->
--  <!ELEMENT Tagged (SimpleTag*, CompositeTag*)>
--  <!--simple variable value-->
--  <!ELEMENT V (#PCDATA)>
--  <!ELEMENT ind (#PCDATA)>
--  <!ATTLIST ind
--      n CDATA #REQUIRED
--  >
--  <!--identification name for this tag-->
--  <!ELEMENT Name (#PCDATA)>
--  <!--Tag with no dimension (simple variable)-->
--  <!ELEMENT SimpleTag (Tag, V)>
--  <!--Tag with one or more dimensions-->
--  <!ELEMENT CompositeTag (Tag, Dim+, Entry)>

package Templates_Parser.XML is

   function Image (Translations : in Translate_Set) return Unbounded_String;
   --  Returns a string representation encoded in XML for this
   --  translate table.

   function Load (Filename : in String) return Translate_Set;
   --  Read XML document Filename and create the corresponding Translate_set

   procedure Save (Filename : in String; Translations : in Translate_Set);
   --  Write the translate table into filename

end Templates_Parser.XML;
```

Index