Templates Parser User's Guide

Document revision level $Revision$
Date: 19 August 2008

AdaCore

http://libre.act-europe.fr/aws

Table of Contents

1 Introduction 1

2 Tags 1

2.1 Tags in template files 1
2.2 Translations 2
2.3 Discrete, Boolean and Composite values 3
2.4 Filters 4
2.4.1 Predefined filters 5
2.4.2 User defined filters 9
2.5 Attributes 11
2.6 Predefined tags 11
2.7 Dynamic tags 12
2.7.1 Lazy_Tag 12
2.7.2 Cursor_Tag 13

3 Template statements 13

3.1 Comments 13
3.2 INCLUDE statement 13
3.3 IF statement 14
3.4 TABLE statement 15
3.5 SET statement 21
3.6 INLINE statement 22

4 Other services 23

4.1 Tag utils 23
4.2 XML representation 23
4.3 Templates2Ada 23
4.3.1 Running templates2ada 25
4.3.2 Customizing templates2ada 26
4.4 Templatespp 26
4.5 Debug 27

Appendix A Templates_Parser API Reference 27

A.1 Templates_Parser 28
A.2 Templates_Parser.Debug 29
A.3 Templates_Parser.Utils 30
A.4 Templates_Parser.XML 31

Index 31

1 Introduction

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 people for the job.
- It reduces 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, the Ada program now simply computes some values, gets some data from a database or whatever and then calls the templates parser to output a page with the data displayed. To the templates parser you just pass the template file name and an associative table.

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

2 Tags

2.1 Tags in template files

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 27. 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 the Ada code. In this context, the role of the Ada code is therefore to prepare what is known as a translation, and then pass it to the templates parser, along with the name of the template file to parse. This results in an expanded

version of the templates file, where all tags have been replaced by the value given in the Ada code.

Let's start with a simple example. Here is the contents of the file '`demo.tmplt`', which is a very basic template file:

```
|
```

On its own, this template has little interest. However, it is used from some Ada code similar to the following '`demo.adb`' file:

```
|
```

Compile this program, link with the templates parser, and when you run it, the output will be:

```
<P>Hello Ada
```

As you can imagine, this is a bare bone example. More complex structures are of course possible. One thing to note, though, is that the template file requires no Ada knowledge for editing, and is strongly related to your application domain. One of the main usage for such templates is to generate web pages. This can be done by a designer that knows nothing of how your Ada code works. But you can use templates in other domains, including to generate Ada code.

2.2 Translations

In your Ada code, you can associate one or more values with a name, and then reference that name in the template file as we just saw above.

Associating the value(s) with the name is done through one of the `Assoc` constructors, see Appendix A [Templates_Parser API Reference], page 27. Ada's overloading resolution mechanism will take care of calling the appropriate constructor automatically.

These associations are then grouped into a dictionary. This dictionary is passed along with the name of the template file to the `Parse` routine itself, which generates the final expanded representation of the template. In fact, you will almost never have to manipulate an association directly, since as soon as it is created you store it in the dictionary.

There are two types of dictionaries in the templates parser:

`Translate_Table`
: This is an array of associations. If you know the exact number of associations when you write your code, this will generally provide a very readable code. The array can be initialized as soon as it is declared.

```
declare
     T : constant Translate_Table :=
       (1 => Assoc ("NAME1", Value1),
        2 => Assoc ("NAME2", Value2));
  begin
     Put_Line (Parse ("demo.tmplt", T));
  end;
```

`Translate_Set`
: If, on the other hand, you do not know statically the number of associations, it is generally a lot more flexible to use another type of dictionary, which isn't limited in size. It is also better to use this type of dictionary if you need extra code to compute the values.

```
declare
     T : Translate_Set
  begin
     Insert (T, Assoc ("NAME1", Value1));
     Insert (T, Assoc ("NAME2", Value2));
  end;
```

Internally, the templates parser will always convert all dictionaries to a `Translate_Set`, which is much more efficient when we need to look values up.

2.3 Discrete, Boolean and Composite values

As we just saw, the values by which a tag is replaced must be provided by the Ada code. Such values can be provided in different formats, depending on the intended use.

The three kinds of tags are **discrete**, **Boolean** and **composite** tags. These are all ways to associate one or more value to a name, which is the name used in the template file.

discrete values

This represents a single value associated with a name. The types of value currently supported are String, Unbounded_String and Integer.

```
Insert (T, Assoc ("NAME", 2));
 Insert (T, Assoc ("NAME", "VALUE"));
```

Boolean values

These are similar to discrete values. However, they are more convenient to manipulate within `@@IF@@` statements in the template. When outside an `IF` statement, such values are represented as TRUE or FALSE.

```
Insert (T, Assoc ("NAME", True));
```

composite values

A composite tag is a variable which contains a set of values. In terms of programming languages, these would generally be called vectors. Since each value within that can itself be a composite tag, you can therefore build multi-dimensional arrays.

These kind of variables will be used with the `TABLE` tag statement see Section 3.4 [TABLE statement], page 15. Outside a table statement, the tag is replaced by all values concatenated with a specified separator. See `Set_Separator` routine. Such tag are variables declared in the Ada program a `Templates_Parser.Tag` type.

There are 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 directly add 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.

```
--  Building a composite tag
   --  Then add it into a translate set
   declare
     V : Tag;
     T : Translate_Set;
   begin
     for Index in 1 .. 10 loop
        V := V & I;
     end loop;
     Insert (T, Assoc ("VECTOR", V));
   end;
```

2.4 Filters

Within the template file, functions can be applied to tags. Such functions are called `filters`. These filters might require one or more parameters, see the documentation for each filter.

The syntax is:

```
@_[[FILTER1_NAME[(parameter)]:]FILTER2_NAME[(parameter)]:]SOME_VAR_@.
```

When multiple filters are associated to a tag, they are evaluated from right to left. In the example above, `FILTER1_NAME` is applied to the result of applying `FILTER2_NAME` to `SOME_VAR`.

Remember that one of the goals in using templates is to remove as much hard-coded information from the actual Ada source, and move it into easily editable external files. Using filters is a convenient way to give the template designer the power to specify the exact output he wants, even without changing the Ada code. For instance, imagine that one suddenly decides that some names should be capitalized in a template. There are two solutions to such a change in design:

- Modify the Ada code to capitalize strings before storing them in a tag variable. What if, in the template, we need the name once capitalized, and once with its original casing ? This means the Ada code would have to create two tags.
- Modify the template itself, and use a filter. A single tag is required on the Ada side, which doesn't even have to be changed in fact. The template would for instance become:

  ```
  @_CAPITALIZE:VAR_@ : constant String := "@_VAR_@";
  ```

The templates parser comes with a number of predefined filters, that can be used in various situations. Some of these are highly specialized, but most of them are fairly general. You can also define your own filters, adapted to specific needs you might have.

Here are some more examples using the predefined filters:

```
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
```

2.4.1 Predefined filters

Here is the complete list of predefined filters that come with the templates parser.

`"+"(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 absolute 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_EOL(EOL)`
: Replaces all occurrences of the `<br>` HTML tag by a line terminator determined by EOL. EOL must be either CR (Carriage-Return), LF (Line-Feed), LFCR (Line-Feed followed by Carriage-Return) or CRLF (Carriage-Return followed by Line-Feed).

`BR_2_LF`
: Shortcut for BR_2_EOL(LF).

`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.

`FILE_EXISTS`

Returns **True** if variable is set and has a value that corresponds to a file name present on the file system 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 with four digits (1970...)

- *Padding*:

  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_DYNAMIC`
: This is a special command filter which indicates that the tag must not be searched in the dynamic tags. See see Section 2.7.1 [Lazy_Tag], page 12. `NO_DYNAMIC` 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.

`REPEAT(`*STR*`)`

This is the second `REPEAT` form. In this case *STR* is repeated a number of time corresponding to the variable value which must be a number.

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

This filter replaces **\n** (where *n* is a number) *STR*'s occurrences 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**. It is possible to space characters in *STR* to avoid parsing confusions. This is required if you need to have `@_` or `_@` or a parenthesis in *STR* for example. *STR* can be a tag variable name. *STR* can contain the following escaped characters : **\n** Carriage Return, **\r** Line Feed and **\t** for Horizontal Tabulation.

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

Idem as above but replaces all occurrences.

`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. If x or y are negative, they are counted from the end of the string, so that 0 matches the last character of the string, -1 matches the character just before,...

`TRIM`

Removes leading and trailing spaces.

`UPPER`

Put all characters in the variable in upper-case.

`WEB_ENCODE`

As WEB_ESCAPE and also encodes all non 7-bit characters and non printable characters using **&#xxx;** HTML encoding.

`WEB_ESCAPE`

Replaces characters '<', '>', '"' and '&' by corresponding HTML sequences: < > " and &

`WEB_NBSP`

Replaces all spaces by an HTML non breaking space.

`WRAP(N)`

Wraps lines having more N characters.

`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).

2.4.2 User defined filters

It is also possible to define a new filter by registering a callback routine associated with the filter name.

You can define three kinds of filters: filters that take no argument, and are therefore simply used as in `@_`FILTER:TAG`_@`, filters that take one or more arguments, used as in `@_`FILTER(param1,param2):TAG`_@`, and filters that are implemented as tagged objects, and take the same form as the filters with arguments described above.

The latter form of filters (using tagged types) provides slightly more flexibility, as you can store your own user data in the filter when it is registered. Among other things, this makes it possible to share filters between various applications, when the filter needs to access some application-specific variable as well.

The templates parser will not try to interpret the parameters for you, and will simply return the string representation of the list of parameters, for instance `"param1,param2"` in the example above. This provides enhanced flexibility, since you are free to use any parameter-separator you want, and to interpret parameters as integer, strings, references to other tags,. . .

The templates parser doesn't support tag substitution within the parameter list, but this is trivial to implement in your own code. For instance, if the user has used `@_`FILTER(REFTAG):TAG`_@`, you are free to either take `REFTAG` as a constant string, or as a reference to another tag, to be looked up in a translation table. You should of course properly document the behavior of your filter.

Here is the templates parser API for defining your own custom filters:

```
type Filter_Context is record
      Translations : Translate_Set;
      Lazy_Tag     : Dynamic.Lazy_Tag_Access;
   end record;

   type Callback is access function
     (Value      : in String;
      Parameters : in String;
      Context    : in Filter_Context) return String;
   --  User's filter callback

   type Callback_No_Param is access function
     (Value   : in String;
      Context : in Filter_Context) return String;
   --  User's filter callback

   procedure Register_Filter
     (Name    : in String;
      Handler : in Callback);
   --  Register user's filter Name using the specified Handler

   procedure Register_Filter
     (Name    : in String;
      Handler : in Callback_No_Param);
   --  Register user's filter Name using the specified Handler
```

In the above calls, Value is the value of the tag on which the filter applies. In the examples above, that would be the value of `TAG` as looked up in the translation table. Context contains the the translation table and the current lazy tag object you can use if you need to look up other tags.

Here is a simple example of a custom filter, which can be used to generate HTML forms. In such a form, it is common to have some `<input>` tags that need a `selected='selected'` attribute if the toggle button should be selected. This can be done without the use of a filter, of course, using a simple `@@IF@@` statement, but that makes the template less readable. The custom filter below behaves as such: it takes one argument, and compares the value of the tag on which the filter is applied to that argument. If they are equal, the string `selected='selected'` will be substituted. As a special case, if the argument to the filter starts with a `'@'` character, the argument is interpreted as the name of a tag to look up first.

```
function Custom_Select_Filter
     (Value      : in String;
      Parameters : in String;
      Context    : in Filter_Context) return String is
   begin
      if Parameters /= "" and then Parameters (Parameters'First) = '@' then
         if Get (Get (Context.Translations,
                      Parameters (Parameters'First + 1 .. Parameters'Last))) =
           Value
         then
            return "selected='selected'";
         end if;

      elsif Value = Parameters then
         return "selected='selected'";
      end if;

      return "";
   end Custom_Select_Filter;

   Register_Filter ("SELECTED", Custom_Select_Filter'Access);
```

and a template would look like:

```
<option value="foo" @_SELECTED(@SELECTED_STATUS):STATUS_@ />
```

2.5 Attributes

In addition to filters, you can also apply attributes to composite tags. 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.

You cannot define your own 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).

`V'Up_Level(n)`
: Use index from the table command **n** level(s) upper so this attribute must be used in a nested table command tag. `'Up_Level` is equivalent to `'Up_Level(1)` (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
```

2.6 Predefined tags

There are some specific tags that can be used in any templates. Here is an exhaustive list:

`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).

2.7 Dynamic tags

Dynamic tags are associations that are not created when `Parse` is called, but only later on when they are actually needed.

Dynamic tags are handled through abstract interfaces and give the opportunity to create tags dynamically while the template is being parsed.

2.7.1 Lazy_Tag

The `Lazy_Tag` object can be used to dynamically handle tags. Such object can be passed to the `Parse` routines. If a template's tag is not found in the translation dictionary, the `Lazy_Tag`'s Value 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 Value
  (Lazy_Tag     : access Dynamic.Lazy_Tag;
   Var_Name     : in     String;
   Translations : in out Translate_Set) is abstract;
--  Value is called by the Parse routines below if a tag variable was not
--  found in the set of translations. This routine must then add the
--  association for variable Name. It is possible to add other
--  associations in the translation table but a check is done to see if
--  the variable Name as been set or not. The default implementation does
--  nothing.
```

One common usage is to handle tag variables that can be shared by many templates and are not always used (because a conditional is False for example). If computing the corresponding value (or values for a ...) is somewhat expensive it is better to delay building such tag at the point it is needed. Using a `Lazy_Tag` object, it is possible to do so. The `Value` procedure will be called if the tag value is needed. At this point, one can just add the corresponding association into the `Translate_Set`. Note that it is possible to add more than one association. If the association for `Var_Name` is not given, this tag has no value.

`Value` will be called only once per template and per tag. This is so that if the value for the tag is expensive to compute, you only pay the price once, and the value is then cached for the remaining of the template. If the value should be recomputed every time, you should consider using a `Cursor_Tag` instead (see Section 2.7.2 [Cursor_Tag], page 13).

2.7.2 Cursor_Tag

In some cases, data structure on the Ada side can be so complex that it is difficult to map it into a variable tag. The `Cursor_Tag` object has been designed to work around such problem. Using a `Cursor_Tag` it is possible to create an iterator through a data structure without mapping it into a variable tag. The data stays on the Ada side. To create a `Cursor_Tag` it is necessary to implement the following abstract routines:

```
function Dimension
  (Cursor_Tag : access Dynamic.Cursor_Tag;
   Var_Name   : in     String) return Natural is abstract;
--  Must return the number of dimensions for the given variable name. For
--  a matrix this routine should return 2 for example.

type Path is array (Positive range <>) of Natural;
--  A Path gives the full position of a given element in the cursor tag

function Length
  (Cursor_Tag : access Dynamic.Cursor_Tag;
   Var_Name   : in     String;
   Path       : in     Dynamic.Path) return Natural is abstract;
--  Must return the number of item for the given path. The first
--  dimension is given by the Path (1), for the second column the Path is
--  (1, 2). Note that each dimension can have a different length. For
--  example a Matrix is not necessary square.

function Value
  (Cursor_Tag : access Dynamic.Cursor_Tag;
   Var_Name   : in     String;
   Path       : in     Dynamic.Path) return String is abstract;
--  Must return the value for the variable at the given Path. Note that
--  this routine will be called only for valid items as given by the
--  dimension and Length above.
```

3 Template statements

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

3.1 Comments

Every line starting with `@@–` are comments and are completely 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_@

...
```

3.2 INCLUDE statement

This statement 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 given after the include file name. It is possible to reference these parameters into the included file with the special variable names `@_$<n>_@`, where n is the include's parameter index (0 is the include file name, 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 commands 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)
```

3.3 IF statement

This is the conditional statement. The complete form is:

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

<expression> is TRUE if it evaluates to one of "TRUE", "T" or "1" and FALSE otherwise. Note that the test is not case sensitive.

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` parts are optional.

The expression here is composed of Boolean variables 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 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.

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

```

```

The following program:

```

```

Will display:

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


```

But the following program:

```

```

Will display:

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


```

3.4 TABLE statement

Table tags are useful to generate `HTML` tables for example. Basically the code between the `@@TABLE@@` and `@@END_TABLE@@` will be repeated as many times as the vector tag has values. If many vector tags are specified in a table statement, the code between the table will be repeated a number of times 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]['REVERSE]@@
      ...
   [@@BEGIN@@]
      ...
   [@@SECTION@@]
      ...
   [@@END@@]
      ...
@@END_TABLE@@
```

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

```
|
```

And the following program:

```
|
```

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 attributes:

`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.

`REVERSE`
: The items will be displayed in the reverse order.

```
|
```

And the following program:

```
|
```

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 it 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, the code outside is common 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:

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 statements.

Here is the template:

And the following program:

The following output will be generated:

```
Hello here is 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 levels, the Nth closest enclosing `TABLE` tag statement will be used for the corresponding index. If there are not enough indexes, the last axis are just streamed as a single text value.

Let's see what happens for a matrix tag:

1. Inside a table of level 2 (a TABLE statement inside a TABLE 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 the second matrix's vector and so on. And the second `TABLE` will be used 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 the 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 to 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:

Using the program:

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
```

3.5 SET statement

The `SET` command tag can be used to define a constant or an alias for an include file parameter. This is especially important in the context of reusable template files. For example, instead of having many references to the **red** color in an HTML document, it is better to define a constant *COLOR* with the value **red** and use *COLOR* everywhere. It is then easier to change the color afterward.

The first form, to define a simple constant that can be used as any other variable in a template file, is:

```
@@SET@@ <name> = <value>
```

The second form, to define an alias for a template file parameter, is:

```
@@SET@@ <name> = $n [| <default_value>]
```

In this case <name> is an alias for the Nth include parameter. In this form it is also possible to define a default value that would be used if the Nth include parameter is not specified.

Some examples:

```
@@SET@@ COLOR = red

@@SET@@ SIZE = $1

@@SET@@ COLOR = $4 | green
```

It is important to note that a variable is set global to a template file. It means that constants set into an include file are visible into the parent template. This is an important feature to be able to have a "theme" like include template file for example.

3.6 INLINE statement

The INLINE statement can be used to better control the result's layout. For example it is not possible to have the results of a vector tag on the same line, also it is not possible to have a conditional output in the middle of a line. The INLINE block tag statement can be used to achieve that.

Elements in an inlined block are separated by a single space by default. It is possible to specify any string as the separator. The text layout on an INLINE block has no meaning (the lines are trimmed on both side). As part of the inline command it is possible to specify texts to output before and after the block.

Syntax:

```
@@INLINE[(<before>)(<separator>)(<after>)]@@
      ...
@@END_INLINE@@
```

There are three supported uses:

`@@INLINE@@`
: In this case there is no text before and after the block and the separator is a single space.

`@@INLINE(<separator>)@@`
: In this case there is no text before and after the block and the separator is the string given as parameter <*separator*>.

`@@INLINE(<before>)(<separator>)(<after>)@@`
: In this case all three values are explicitly given.

<*before*>, <*separator*> and <*after*> may contain control characters:

`\n` To insert a new-line (CR+LF or LF depending on the Operation System)

`\r` To insert a line-feed

`\\` To insert a single backslash

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

```
|
```

Using the program:

```
|
```

We get the following result:

```
colors="Red, Green, Blue"
```

Another example with an `IF` tag statement:

```
|
```

Using the program:

```
|
```

We get the following result:

```
A big car.
```

4 Other services

4.1 Tag utils

The child package `Utils`, see Section A.3 [Templates_Parser.Utils], page 30 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 useful for example, in the context of AWS to store a Tag into a session variable. See the AWS project.

4.2 XML representation

The child package `XML`, see Section A.4 [Templates_Parser.XML], page 31 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 conform to a specific `DTD` (see the Ada spec file).

4.3 Templates2Ada

`templates2ada` is a tool that will generate a set of Ada packages from a templates file. These Ada packages can then be used in your application to avoid hard-coded strings, and help maintain the templates and the code synchronized.

One of its goal is to ensure that you are only setting tags that actually exist in the template (and thus prevent, as much as possibly, typos in the name of tags); also, when combined with other tools, to help ensure that all tags needed by the template are properly set.

Templates2ada also has special knowledge about HTTP constructs and will generate Ada constants for the HTTP parameters you might receive in return. Once more the goal is to help avoid typos in the Ada code.

For instance, we will consider a simple template file, found in a local file '`resources/block1.thtml`'. This template contains the following simple html code:

```
<form>
  <input name="PARAM1" value="@_TAG1_@" />
  <input name="PARAM2" value="@_TAG2_@" />
</form>
```

When you run '`templates2ada`' (as described in the following subsection), the following Ada package will be generated. Note that this is only the default output of '`templates2ada`', which can be fully tailored to your needs.

```
package Templates.Block1 is
  pragma Style_Checks (Off);
  Template : constant string := "resources/block1.thtml";
  Tag1 : constant String := "TAG1";
  Tag2 : constant String := "TAG2";
  package Http is
    Param1 : constant String := "PARAM1";
    Param2 : constant String := "PARAM2";
  end Http;
end Templates.Block1;
```

`templates2ada` knows about special constructs in the template file. Such templates are generally associated with html pages. It is possible to specify within the template itself what the url associated with the template is, so that it provides a convenient link between the two. Likewise, you can also define explicitly what the possible HTTP parameters are when loading that page. This is mostly useful when those parameters do not correspond to some form fields within the page itself. The syntax for these two is the following:

```
@--  HTTP_URL(the_url): any comment you want
@--  HTTP_GET(param1_name): description of the parameter
@--  HTTP_GET(param2_name): description of the parameter
```

and that results in the following constants in the generated Ada package:

```
package Templates.Block1 is
  URL : constant String := "the_url";
  package Http is
     Param1_Name : constant String := "param1_name";
     Param2_Name : constant String := "param2_name";
  end Http;
end Templates.Block1;
```

The templates parser API lets you define your own custom filters. It is often useful for those filters to take parameters, just like the predefined filters do. However, it is also useful for these parameters to be able to check the value of other tags. One convention for doing this is to start the name of the parameter with "`:`. See for example the example in see Section 2.4.2 [User defined filters], page 9. As a reminder, the template would look like

```
<option value="foo" @_SELECTED(@SELECTED_STATUS):STATUS_@ />
```

The `templates2ada` tool knows about this special convention, and would generate the following Ada package from this example:

```
package Templates.Block1 is
   Selected_Status : constant String := "SELECTED_STATUS";
   Status : constant String := "STATUS";
end Templates.Block1;
```

4.3.1 Running templates2ada

This tool parses all the template files found in a directory, and then generate an output file from these, based on a template file (a default example of which is provided as '`templates.tads`'). The latter contains in fact two examples, depending on whether one Ada package should be generated per template, or whether a single package should be build. In the former case, if you are using the GNAT compiler, you should run `gnatchop` on the resulting file. Here is an example to run this tool for the example we described above.

```
$ rm -f src/templates/*.ads
$ templates2ada -d resources/ -o src/templates/generated -r
$ cd src/templates; gnatchop -w -q generated
$ rm -f src/templates/generated
```

If, in you Ada code, you no longer use hard-coded strings but only the constants found in the output packages, this will ensure that you are not trying to set tags that are never used in the template.

The other check that impacts the quality of your code is to ensure that all tags that are used by the templates are properly set. This cannot be ensured by the compiler only, but using an external tool it is relatively to do.

For instance, if you are using GNAT, we recommend the following additional targets in your '`Makefile`':

```
unset_tags:
gnat xref -u main.adb | fgrep templates-
```

This checks for all unused entities in files called '`templates-*`', which are the files generated by '`templates2ada`'.

'`templates2ada`' can be used in other situations as well. For instance, one possible use is to generate, as output, a new template file that itself contains a series of `@@SET@@` commands. This generated file can then be `@@INCLUDE@@`d in your own templates. We have used it with some success when implementation a web server: it is often the case that hyper links refer to other pages in the same server. We have avoided hard-coding the URLs and the names of their HTTP GET parameters, by fetching these names from the generated file we were talking above.

The templates parser comes with an example file, called '`all_urls.thtml`', which can be used with the `-t` switch to `templates2ada`, and will generated a template file as output. You would use it as:

```
@@INCLUDE@@ all_urls.html
<a href="@_URI_BLOCK1_@?@_HTTP_BLOCK1__PARAM1_@=12" />
```

and this ensures the link is valid.

'`templates2ada`' supports a number of command line switches:

- -d <dir>

  This switch specifies the directory in which the templates file are searched for.

- -o <file>

  This switch specifies the output file name

- -e <ext>

  This file specifies the file name extension for template files. All files in the directory that have this extension will be processed by templates2ada.

- -t <tmplt>

  This file specifies the template file to be used for the output file. The templates parser comes with an example for such a file, called '`templates.tads`', that you can adapt to your own needs.

- -r

  Sub directories of the one specified by `-d` will also be searched.

- -v

  Activate the verbose mode. This will output a warning when an http parameter has a name made only of template parser tags, since no matching entry can then be created for it in the output file.

4.3.2 Customizing templates2ada

As was mentioned before, the output of templates2ada is a single file that results from parsing a template file. An example of such a file is provided in the templates2ada distribution, as '`templates.tads`'.

You are strongly encouraged to modify this file to adapt it to your needs, and then use the `-t` switch to '`templates2ada`' to make use of your modified file.

This file contains extensive comments on how to make use, and customize, it. This documentation is not duplicated here

4.4 Templatespp

`templatespp` is a pre-processor based on the template parser. It is generally used from scripts to process files and generate other files. One of the possible uses, for instance, is to write the CSS (style-sheet) of a web site as a template file (for instance '`mycss.tcss`'), and use template parser structures in there. This is a good way to share colors for instance, or to name constants, as is often done in Ada code.

Here is a small example of such a CSS:

```
@@SET@@ COLOR1=blue
@@SET@@ COLOR2=red
@@SET@@ LENGTH1=10

body {background:@_COLOR1_@}
div  {background:@_COLOR2_@}
ul.class {background:@_COLOR1_@}  /* same color as body */

ul   {width:@_ADD(3):LENGTH1_@px} /* ul 3 pixels wider than li */
li   {width:@_LENGTH1_@px}
```

Such a file would be processed with the following command line:

```
templatespp -o mycss.css mycss.tcss
```

4.5 Debug

A set of routines to help to debug the `Templates_Parser` engine, see Section A.2 [Templates_Parser.Debug], page 29. For example, `Debug.Print_Tree` will display, to the standard output, a representation of the internal semantic tree for a template file.

Appendix A Templates_Parser API Reference

A.1 Templates_Parser

A.2 Templates_Parser.Debug

A.3 Templates_Parser.Utils

A.4 Templates_Parser.XML

Index

@

@_DAY_@ ... 12
@_DAY_NAME_@ ... 12
@_HOUR_@ ... 12
@_MINUTE_@ ... 12
@_MONTH_@ ... 11
@_MONTH_NAME_@ ... 12
@_NOW_@ ... 11
@_NUMBER_LINE_@ ... 18
@_SECOND_@ ... 12
@_TABLE_LEVEL_@ ... 18
@_TABLE_LINE_@ ... 18
@_UP_TABLE_LINE_@ ... 18
@_YEAR_@ ... 11

A

Attribute, 'Length ... 11
Attribute, 'Line ... 11
Attribute, 'Max_Column ... 11
Attribute, 'Min_Column ... 11
Attribute, 'Up_Level ... 11

C

Command, @@– ... 13
Command, comments ... 13
Command, IF ... 14
Command, IF expression ... 14
Command, INCLUDE ... 13
Command, INLINE ... 22
Command, REVERSE ... 16
Command, SET ... 21
Command, TABLE ... 15
Command, TERMINATE_SECTIONS ... 16
Cursor_Tag ... 13

D

Debug ... 27
Dynamic tags ... 12

F

Filter, "*" ... 5
Filter, "+" ... 5
Filter, "-" ... 5
Filter, "/" ... 5
Filter, ABS ... 5
Filter, ADD_PARAM ... 5
Filter, BR_2_EOL ... 5
Filter, BR_2_LF ... 5
Filter, CAPITALIZE ... 5
Filter, CLEAN_TEXT ... 5
Filter, COMA_2_POINT ... 5
Filter, CONTRACT ... 6
Filter, DEL_PARAM ... 6
Filter, EXIST ... 6
Filter, FILE_EXISTS ... 6
Filter, FORMAT_DATE ... 6
Filter, FORMAT_NUMBER ... 7
Filter, IS_EMPTY ... 7
Filter, LF_2_BR ... 7
Filter, LOWER ... 7
Filter, MATCH ... 7
Filter, MAX ... 7
Filter, MIN ... 7
Filter, MOD ... 7
Filter, NEG ... 7
Filter, NO_DIGIT ... 7
Filter, NO_DYNAMIC ... 7
Filter, NO_LETTER ... 7
Filter, NO_SPACE ... 7
Filter, OUI_NON ... 8
Filter, POINT_2_COMA ... 8
Filter, REPEAT ... 8
Filter, REPLACE ... 8
Filter, REPLACE_ALL ... 8
Filter, REPLACE_PARAM ... 8
Filter, REVERSE ... 8
Filter, SIZE ... 8
Filter, SLICE ... 8
Filter, TRIM ... 8
Filter, UPPER ... 8
Filter, WEB_ENCODE ... 8
Filter, WEB_ESCAPE ... 8
Filter, WEB_NBSP ... 9
Filter, WRAP ... 9
Filter, YES_NO ... 9
Filters ... 4

L

Lazy_Tag ... 12

T

Tag utils ... 23
Tag, Boolean ... 3
Tag, composite ... 3
Tag, discrete ... 3
Templates_Parser ... 28
Templates_Parser.Debug ... 29
Templates_Parser.Utils ... 30
Templates_Parser.XML ... 31
templates2ada ... 23
templatespp ... 26
Translate_Set ... 2
Translate_Table ... 2

X

XML ... 23