PML Manual

PML provides an extension to HTML, embedding a simple imperative language with C-like syntax directly in web pages. This description of PML assumes prior knowledge of C, C++ or Java.

Format Specification

PML consists of three parts that are all closely connected:

1. PML expressions and assignments.
2. PML tags and attributes.
3. PML classes and named standard objects.

These three parts are described in the following sections.

Expressions and Assignments

All expressions consist of values combined with operators. Below, the different types of values are described, followed by the operators that can be applied.

Simple Values

PML uses three simple types: Strings, numbers and truth values. Although distinctions are made between these types, PML is a weakly typed language in the sense that variables are not assigned a specific type and all values are automatically converted to the type expected in the context.

The formats of the simple values are:

Strings Strings are noted with single quotes as in `'This is a nice string'`. If single quotes are needed within the string, two quotes are put next to each other: `'This string contains '' single quotes'`.

Numbers Numbers can be both integers and floats. The format of a number is:

$$[+|-]\mathit{integerpart}[.\mathit{decimalpart}][(e|E)[+|-]\mathit{exponent}]$$

Examples of numbers are: `42`, `-42.0`, `420e-1` and `+4.2E1`.

Truth values Truth values are either `true` or `false`. Since PML is case-insensitive `TRUE`, `False` and `fAlSe` are also legal truth values.

Simple values can be used in expressions or they can be printed — see under the out object or the `PML_VAL` tag.

Variables

PML variables need not be declared — instead they are created automatically when they are assigned a value. Variable names must begin with a letter or underscore (`'_'`), and can then subsequently consist of letters, numbers and underscores. PML is case-insensitive. Values are assigned to variables with the `PML_ASSIGN` tag or with the assignment operators described below. Furthermore, variables can be assigned values in `PML_FOR` tags, which are also described below. Variables are defined when they are assigned a value and stay defined throughout the rest of file.

Objects and Arrays

In addition to the simple values, an abstract type of objects exists. An object is essentially a collection of fields, each of which is a simple value or another object. Simple field values can, like other simple values, be used in expressions or be printed.

The fields of an object can be reached in three different ways, using these notation types:

1. objectname.fieldname
2. objectname[string]
3. objectname[index]

The first two are equivalent, but whereas the first requires a static field name, the second allows an arbitrary string which evaluates to a field name. For instance, these expressions will evaluate to the same (the `@` symbol is the string concatenation operator which will be described later):

- `product.type`
- `product['type']`
- `product['ty'@'pe']`

An advantage of this second notation type is that field names can be created dynamically at interpretation time. The third notation type gives the possibility of accessing all fields in an object by their index: All fields are given an index between 0 and the total number of fields minus 1. The order of the fields is arbitrary. So, if `type` is the first field in the object `product`, this field can also be referred to as `product[0]`. Similarly, field names are accessed with curly braces:

objectname{index}

In this case, the expression `product{0}` will return the field name `'type'`.

Arrays in PML are basically objects with unnamed fields which may only be accessed by index. Here, the {} operator will just return the index of a field (array{n}=n), which is not very useful. The number of fields in an object or a array is returned with the # operator:

#objectname

If the # operator is applied to simple values, it will always return 0.

Standard Objects

There are certain standard objects which are always defined:

cgi CGI variables transferred as parameters to this PML page.

date Used for various operations on dates, including printing them.

math Contains a number of standard mathematical methods.

out Used for printing within `PML_SCRIPT` tags (see below).

string Contains string manipulation methods.

These objects are automatically created and are directly available in the PML file.

Operators

The following operators are available:

Number operators	+ − * / div mod
String operators	@
Boolean operators	! & |
Conditional operator	? :
Object operators	. [ ] { } #
Comparison operators	= < <= > >= !=

These operators all have the same meaning and operator precedence as the corresponding C-operators with the following extensions:

- "@" is the string concatenator.
- Double operators such as "==" and "&&" are written "=" and "&" in PML.
- Simple values are automatically converted to the type expected by the operator.
- It is always possible to determine the type of an expression based on the operator that was last applied.

Expressions

Expressions have one of four forms, depending on the type of the expression: numerical expression (*n-expr*), string expression (*s-expr*), boolean expression (*b-expr*) or object expression (*o-expr*). In the description of the various PML tags below, *expr* will denote an arbitrary expression, while *n*-, *s*-, *b*- and *o*- signify that an expression must be of a particular type. This is particularly important in the case of object expressions, given that simple values can always be converted into each other.

Assignments

PML allows a short notation for assignments in the initialisation and incrementation parts of *for*-loops, as well as inside `PML_SCRIPT` tags:

variablename := expression

The fields of an object can be assigned values in a similar way. In addition to the := assignment operator, a number of increment assignment operators exist. These operators all work as would be expected. The unary increment operators only exist in the postfix form.

Assignment operators	:= @= ++ −−

PML Attributes

Special PML attributes can be used within all HTML tags — these PML attributes will be evaluated and replaced by corresponding HTML attributes. The form of the PML attributes is

$$\text{PML\_}\textit{html-tagname} = \text{“}\textit{expression}\text{”}$$

which will be replaced by

$$\textit{html-tagname} = \text{“}\textit{value}\text{”}$$

where *value* is the value of the evaluated *expression*.

PML attributes where the expression evaluates to a truth value are treated as a special case. In this case the value `true` results in the HTML attribute being written as a stand alone attribute, while `false` means that the attribute is removed. As an example:

```
<INPUT TYPE=button PML_NAME="'number'@i">
```

will for $i = 42$ be replaced by

```
<INPUT TYPE=button NAME="number42">
```

and

```
<INPUT TYPE=checkbox PML_CHECKED="id=42">
```

will for $id = 42$ be replaced by

```
<INPUT TYPE=checkbox CHECKED>
```

and otherwise

```
<INPUT TYPE=checkbox>.
```

PML Tags

All PML tags are described below:

PML_ASSIGN

```
<PML_ASSIGN NAME="variablename" VALUE="expr" [COND="b-expr"]>
```

Assigns the value of `expr` to `variablename`. If the `COND` attribute is set, the value is only assigned if the conditions in `COND` are satisfied.

PML_VAL

```
<PML_VAL [PRECISION="number"] [ENCODING="html|url"]
 [WIDTH="[-]fieldwidth"] [COND="b-expr"]>expr</PML_VAL>

<PML_VAL ITEM="o-expr" [PRECISION="number"] [ENCODING="html|url"]
 [WIDTH="[-]fieldwidth"] [COND="b-expr"]>fieldname</PML_VAL>
```

Prints a PML value in the HTML text. When values of the fields of an object are printed using `PML_VAL`, the object is given in the `ITEM`-attribute. The `PRECISION`-attribute indicates the number of decimals in a floating point number. This number must be non-negative. `ENCODING=html` encodes and prints the value as HTML (e.g. æ=&aelig;), whereas `ENCODING=url` encodes and prints the value as a URL (e.g. æ=%E6). `WIDTH` denotes a minimum width for the value, which is padded with blank spaces if necessary. A minus before the width denotes that the value will be flushed left (i.e. the padding spaces will be added on the right). If the `COND` attribute is set, the value is only printed if the conditions in `COND` are satisfied.

PML_FOR, PML_HEADER, PML_FOOTER and PML_BREAK

```
 <PML_FOR [NAME="string"] [INIT="assign"] COND="b-expr" [INCR="assign"]>
 [<PML_HEADER> PML </PML_HEADER>]
 PML
 [<PML_BREAK [NAME="string"]>]
 PML
 [<PML_SEPARATOR> PML </PML_SEPARATOR>]
 [<PML_FOOTER> PML </PML_FOOTER>]
</PML_FOR>
```

This is a loop structure which repeats the HTML (or PML) present between the start and the end tag. The loop runs until the condition given by `COND` evaluates to false. An initial value can be assigned to a variable in the `INIT` attribute and `INCR` can perform an incrementation at the end of each loop.

The optional tags `PML_HEADER`, `PML_SEPARATOR` and `PML_FOOTER` contain HTML which should only be used in the first, inner and last loop, reprectively. For instance, these can be used for table headers (`<TH>`) and other HTML which should only be applied if the loop is executed at least once.

The optional `PML_BREAK` tag is used to leave the loop before `COND` evaluates to false. If there are several `PML_FOR` loops nested in each other, they can be named with the `NAME` attribute in the `PML_FOR` tag and the corresponding `NAME` attribute of the `PML_BREAK` tag can be used to determine which loop should be stopped.

```
<PML_FOR COUNTER=var ITEM=o-expr>...</PML_FOR>
```

Above is an alternative version of `PML_FOR` for traversing arrays and objects. `var` is a variable and `o-expr` is an expression which evaluates to an object. This correponds to writing

```
<PML_FOR INIT="var:=0" COND="o-expr[var]!=null" INCR="var:=var+1">
```

PML_SWITCH, PML_CASE and PML_DEFAULT

```
<PML_SWITCH EXPR="expr">
 {<PML_CASE VALUE="expr"> PML [</PML_CASE>]}
 [<PML_DEFAULT> PML [</PML_DEFAULT>]]
</PML_SWITCH>
```

This is a multi-way selection structure. Within the `PML_SWITCH` tag a number of `PML_CASE` tags can be declared along with one optional `PML_DEFAULT` tag. The `EXPR` attribute of the `PML_SWITCH` tag is compared with the `VALUE` attribute of each `PML_CASE` tag. Only the HTML contained in a `PML_CASE` tag where the `VALUE` matches the switch expression is used. If no `PML_CASE` tags match, `PML_DEFAULT` is used — if no `PML_DEFAULT` exists, no HTML is applied.

Please note that, contrary to the switch structure in C, no break statement should be used within a `PML_CASE` tag.

PML_IF, PML_ELSIF and PML_ELSE

```
<PML_IF COND="b-expr">PML</PML_IF>
{<PML_ELSIF COND="b-expr">PML</PML_ELSIF>}
[<PML_ELSE>PML</PML_ELSE>]
```

This is the classic selection structure, where the HTML inside the `PML_IF` tag is only used if the condition in `COND` is satisfied. Immediately after a `PML_IF` tag, one or more `PML_ELSIF` tags and/or a `PML_ELSE` tag can be used. If `COND` evaluates to false in the first `PML_IF`, the `COND` attributes of the following (if any) `PML_ELSIF` tags is tested, and the HTML contained in the first one which evaluates to true is used. If none match, the HTML inside the `PML_ELSE` tag is used.

PML_META

```
<PML_META name=value ...>
```

The `PML_META` tag can be used to control settings of the pml-interpreter, and other special tasks. `name` can be one of the following:

debug If debug is set to true, parse errors and null values are printed as HTML comments.

verbose If verbose is set to true, an HTML comment is printed for each interpreted PML tag.

nospace If nospace is set to true, spaces and newlines are removed from all pml output (except in debug mode) until a `<PML_META nospace=false>` is met.

redirect Redirects the browser to a given URL, where the URL is supplied as a PML string.

PML_COMMENT

```
<PML_COMMENT></PML_COMMENT>
```

Comments which will not appear in the HTML page.

PML_SCRIPT

```
<PML_SCRIPT>statement</PML_SCRIPT>
```

A `PML_SCRIPT` tag can contain larger blocks of pure PML in a C-like syntax described in the following BNF:

```
statement : $
          | statement ;
          | statement expr ;
          | statement for ( expr ; expr ; expr )  statement
          | statement foreach ( expr ; expr )  statement
          | statement break ;
          | statement switch ( expr )  cases
          | statement if ( expr )  statement  elsif else
```

```
cases : $
      | cases case ( expr )  statement
      | cases default  statement
elsif : $
      | elsif elsif ( expr )  statement
else : $
     | else  statement
```

In addition to these statements, C++ style comments (// and /**/) can be used.

All `expr` are regular pml expressions. However, the initialisation and incrementation parts of for loops must be assignments of the type `var := value`. Single `expr` can also use assignments as well as regular pml expressions. These expressions are evaluated and the result is discarded, making method call with side effects possible (e.g. all methods on the out object):

Example

```
<PML_SCRIPT>
a := 42;
out.put(a);
</PML_SCRIPT>
```

There are two types of for loops: A regular `for` loop correponding to the `PML_FOR` tag with `init`, `cond` and `incr` parts, and a `foreach` loop corresponding to `PML_FOR` with `counter` and `item`.

Example

```
<PML_SCRIPT>
foreach(i; cgi) {
 out.put(cgi[i]);
}
</PML_SCRIPT>
```

Standard Objects

Out

The out object contains methods for printing values in pml-script. This object also has access to the the methods and attributes available to the `PML_META` tag.

Out object attributes

`out.verbose` Read/write boolean which indicates whether pml parses in verbose mode (see `PML_META`).

`out.debug` Read/write boolean which indicates whether pml parses in debug mode (see `PML_META`).

`out.nospace` Read/write boolean which indicates whether pml parses in nospace mode (see `PML_META`).

`out.width` Read/write integer which indicates the width of the of the print at the next call to out.put() (see `PML_VAL`).

Out object methods

`out.put(`*string*`)` Prints *string*.

`out.put(`*number,precision*`)` Prints a floating *number* with the number of decimals given by *precision*.

`out.vput(`*string*`)` Only in verbose mode: Prints *string*.

`out.vput(`*number,precision*`)` Only in verbose: Prints a floating *number* with the number of decimals given by *precision*.

`out.dput(`*string*`)` Only in debug mode: Prints *string*.

`out.dput(`*number,precision*`)` Only in debug: Prints a floating *number* with the number of decimals given by *precision*.

`out.setContentType(`*mime*`)` Sets the content-type for the document. This method must be called in the top of the document, before any calls to print methods.

`out.redirect(`*url*`)` Redirects the browser to another *url*. This method must be called in the top of the document before any calls to print methods.

Date

The date class generates date objects. A number of methods available on the date objects make it possible to format and print the dates.

Date class attributes and methods

`date.get(`*date[, format]*`)` Constructor which creates a date object given a *date* string, formatted according to the *format* string (similar to strftime). If no format is given, the default is '%d/%m-%Y'.
If the *date* argument is given as a number instead of a string, the date is created as the according number of seconds since Jan 1 1970 00:00:00.
Examples:

```
d1:=date.get('20/04-1999')
d2:=date.get('990420', '%y%m%d')
d3:=date.get('17:45:05 April 20 - 1999', '%H:%M:%S %B %d - %Y')
d4:=date.get(924623105)
t1:=date.get('19:33:05', '%H:%M:%S')
```

`date.today` Returns the date object for today.

`date.now` Returns the number of seconds since Jan 1 1970 00:00:00.

Date object attributes and methods

`date.format(`*format[, language]*`)` Returns a string with the date in the given *format* (again, similar to strftime). The *language* argument is a two-letter code which denotes the language used for formatting of week days, month names etc. The two-letter code is the same as the one returned by the unix-command `locale -a`, typically formed by the two first letters in the language name (e.g. da=dansk, de=deutch or en_UK=British English). If no language is specified, Danish is used.
Examples:

```
d1.format('%a %d %b %y') => 'Man 20 Apr 99'
d2.format('%a %d %b %y', 'en') => 'Mon 20 Apr 99'
t1.format('%I:%M:%S %p') => '07:33:05 PM'
```

`date.year` The same as `date.format('%Y')`.

`date.month` The same as `date.format('%m')`.

`date.day` The same as `date.format('%d')`.

`date.hour` The same as `date.format('%H')`.

`date.minute` The same as `date.format('%M')`.

`date.second` The same as `date.format('%S')`.

`date.unit` The number of seconds since Jan 1 1970.

Furthermore, date objects can automatically be converted to strings of the default format '%d/%m - %Y %H:%M:%S'. For example, the tag

```
<PML_VAL>date.today</PML_VAL>
```

will write the string 31/12-1999 17:57:00 if today is New Year's Eve year 1999 at 17.57.

String

`string.match(`*string, pattern[, options]*`)` Returns an array of matches. *options* can be *i* which gives case-insensitive matching.

Example:

```
string.match('A little test', 't+') => ['tt', 't', 't']
```

`string.search(`*string, pattern[, options]*`)` Searches for the first match of *pattern* in *string* and returns the substring that starts with the match and includes the rest of the string. *options* can be *i* which gives case-insensitive matching.

Example:

```
string.search('A little test', 't+l') => 'ttle test'
```

`string.substring(`*string, pattern[, options]*`)` Returns the first substring of *string* that matches *pattern*. *options* can be *i* which gives case-insensitive matching.

Example:

```
string.substring('A little test', 't+l') => 'ttl'
```

string.substring(*string, index[, length]*) Returns the first substring that starts at index *index* and has length *length*. If *length* is not given, the rest of the string is returned.

string.split(*string, pattern*) Uses *pattern* to split *string*.

Example:

```
string.split('A little test', 't+') => ['A li', 'le', 'es', '']
```

string.splitLines(*string*) As `string.split(str, '\n')` (and better, since split does not work with this pattern).

string.join(*stringArray, bindString*) The elements in *stringArray* are joined in one string in the format `stringArray[0]` + `bindString` + `stringArray[1]` + `bindString` ...

Example:

```
string.join(['A li', 'le', 'es', ''], 'f') => 'A lifle fesf'
```

string.replace(*sourcestr, pattern, newstr, options*) *Pattern, newstr* are applied to *sourcestr* using the Perl model. The replacement model in Perl has the form "*sourcestr* = */pattern/newstr/options*" — in PML, this is replaced with a call to `string.replace(`*sourcestr, pattern, newstr, options*`)`. *options* can have the values *i* (case-insensitive match) or *g* (global replacement, where all occurrences of *pattern* are replaced by *newstr*). Please see the Perl manual for more information.

Examples:

```
string.replace('A little test', 't', 'f', 'g') => 'A liffle fesf'
string.replace('A little test', '(.)(t+)', '$2$1', 'g') =>'A lttilet
ets'
```

string.toUpper(*string*) *string* converted to upper case characters.

string.toLower(*string*) *string* converted to lower case characters.

string.charAt(*string,pos*) Returns the character at index *pos* in *string*.

string.HTMLencode(*string*) HTML encoded *string*.

string.URLencode(*string*) URL encoded *string*.

string.Base64Encode(*string*) Base 64 encoded *string*.

string.length(*string*) Number of characters in *string*.

string.md5(*string*) MD5 checksum of *string*.

string.precision(*number, precision*) Returns *number* with *precision* as a string.

Math

The math class contains a number of standard math methods:

abs(*x*) Absolute value of *x*.

acos(*x*) Arc cosine of *x*.

`asin`(*x*) Arc sine of *x*.

`atan`(*x*) Arc tangent of *x*.

`atan2`(*x*, *y*) Arc tangent of two variables *x* and *y*.

`ceil`(*x*) Rounds *x* up the the nearest integer.

`cos`(*x*) Cosine of *x*.

`exp`(*x*) *e* raised to the power of *x*.

`floor`(*x*) Rounds *x* down to the nearest integer.

`log`(*x*) Natural logarithm of *x*.

`max`(*x*, *y*) Maximum of *x* and *y*.

`min`(*x*, *y*) Minimum of *x* and *y*.

`pow`(*x*, *y*) *x* raised to the power of *y*.

`round`(*x*) Rounds *x* to the nearest integer (up or down).

`sin`(*x*) Sine of *x*.

`sqrt`(*x*) Non-negative square root of *x*.

`tan`(*x*) Tangent of *x*.