Module:Lang

From Blue Gold Program Wiki
The printable version is no longer supported and may have rendering errors. Please update your browser bookmarks and please use the default browser print function instead.

Documentation for this module may be created at Module:Lang/doc

```
--[=[

Lua support for the {{lang}}, {{lang-xx}}, and {{transl}} templates and
replacement of various supporting templates.

]=]

require('Module:No globals');

local initial_style_state;
-- set by lang_xx_normal() and lang_xx_italic()

local getArgs = require ('Module:Arguments').getArgs;
local unicode = require ("Module:Unicode data");
-- for is_latin() and is_rtl()
local yesno = require ('Module:Yesno');

local lang_name_table = mw.loadData ('Module:Language/name/data');

local synonym_table = mw.loadData ('Module:Lang/ISO 639 synonyms');
-- ISO 639-2/639-2T code translation to 639-1 code

local lang_data =  mw.loadData ('Module:Lang/data');
-- language name override and transliteration tool-tip tables

local namespace = mw.title.getCurrentTitle().namespace;
-- used for categorization

local this_wiki_lang = mw.language.getContentLanguage().code;
-- get this wiki's language

local maint_cats = {};
-- maintenance categories go here
local maint_msgs = {};
-- and their messages go here


--[[--------------------------< I S _ S E T >------------------------------
----------------------------------

Returns true if argument is set; false otherwise. Argument is 'set' when it
exists (not nil) or when it is not an empty string.

]]

local function is_set( var )
        return not (var == nil or var == '');
end


--[[--------------------------< I N V E R T  _ I T A L I C S >--------------
```

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

This function attempts to invert the italic markup a args.text by
adding/removing leading/trailing italic markup
in args.text.  Like |italic=unset, |italic=invert disables automatic italic
markup.  Individual leading/trailing
apostrophes are converted to their html numeric entity equivalent so that the
new italic markup doesn't become
bold markup inadvertently.

Leading and trailing wiki markup is extracted from args.text into separate
table elements.  Addition, removal,
replacement of wiki markup is handled by a string.gsub() replacement table
operating only on these separate elements.
In the string.gsub() matching pattern, '.*' matches empty string as well as
the three expected wiki markup patterns.

This function expects that markup in args.text is complete and correct; if it
is not, oddness may result.

]]

local function invert_italics (source)
        local invert_pattern_table = {
-- leading/trailing markup add/remove/replace patterns
                [""]="\'\'",
-- empty string becomes italic markup
                ["\'\'"]="",
-- italic markup becomes empty string
                ["\'\'\'"]="\'\'\'\'\'",
-- bold becomes bold italic
                ["\'\'\'\'\'"]="\'\'\'",
-- bold italic become bold
                };
        local seg = {};

        source = source:gsub ("%f[\']\'%f[^\']", '&#39;');
-- protect single quote marks from being interpreted as bold markup

        seg[1] = source:match ('^(\'\'+%f[^\']).+') or '';
-- get leading markup, if any; ignore single quote
        seg[3] = source:match ('.+(%f[\']\'\'+)$') or '';
-- get trailing markup, if any; ignore single quote

        if '' ~= seg[1] and '' ~= seg[3] then
-- extract the 'text'
                seg[2] = source:match ('^\'\'+%f[^\'](.+)%f[\']\'\'+$')
-- from between leading and trailing markup
        elseif '' ~= seg[1] then
                seg[2] = source:match ('^\'\'+%f[^\'](.+)')
-- following leading markup
```

```
        elseif '' ~= seg[3] then
                seg[2] = source:match ('(.+)%f[\']\'\'+$')
-- preceding trailing markup
        else
                seg[2] = source
-- when there is no markup
        end

        seg[1] = invert_pattern_table[seg[1]] or seg[1];
-- replace leading markup according to pattern table
        seg[3] = invert_pattern_table[seg[3]] or seg[3];
-- replace leading markup according to pattern table

        return table.concat (seg);
-- put it all back together and done
end


--[[--------------------------< V A L I D A T E _ I T A L I C >--------------
---------------------------------

validates |italic= or |italics= assigned values.

When |italic= is set and has an acceptible assigned value, return the
matching css font-style property value or,
for the special case 'default', return nil.

When |italic= is not set, or has an unacceptible assigned value, return nil
and a nil error message.

When both |italic= and |italics= are set, returns nil and a 'conflicting'
error message.

The return value nil causes the calling lang, lang_xx, or transl function to
set args.italic according to the template's
defined default ('inherit' for {{lang}}, 'inherit' or 'italic' for {{lang-
xx}} depending on
the individual template's requirements, 'italic' for {{transl}}) or to the
value appropriate to |script=, if set ({{lang}}
and {{lang-xx}} only).

Accepted values and the values that this function returns are are:
        nil                -        when |italic= absent or not set; returns
nil
        default        -        for completeness, should rarely if ever be
used; returns nil
        yes                -        force args.text to be rendered in italic
font; returns 'italic'
        no                -        force args.text to be rendered in normal
font; returns 'normal'
        unset        -        disables font control so that font-style
```

```
applied to text is dictated by markup inside or outside the template; returns
'inherit'
        invert        -        disables font control so that font-style
applied to text is dictated by markup outside or inverted inside the
template; returns 'invert'

]]

local function validate_italic (args)
        local properties = {['yes'] = 'italic', ['no'] = 'normal', ['unset']
= 'inherit', ['invert'] = 'invert', ['default'] = nil};
        local count = 0
        for _, arg in pairs {'italic', 'italics', 'i'} do
                if args[arg] then
                        count = count + 1
                end
        end

        if count > 1 then
-- return nil and an error message if more than one is set
                return nil, 'only one of &#124;italic=, &#124;italics=, or
&#124;i= can be specified';
        end
        return properties[args.italic or args.italics or args.i], nil;
-- return an appropriate value and a nil error message
end


--[=[-------------------------< V A L I D A T E _ C A T _ A R G S >---------
-----------------------------------------------

Default behavior of the {{lang}} and {{lang-xx}} templates is to add
categorization when the templates are used in mainspace.
This default functionality may be suppressed by setting |nocat=yes or
|cat=no.  This function selects one of these two parameters
to control categorization.

Because having two parameters with 'opposite' names and 'opposite' values is
confusing, this function accepts only affirmative
values for |nocat= and only negative values for |cat=; in both cases the
'other' sense (and non-sense) is not accepted and the
parameter is treated as if it were not set in the template.

Sets args.nocat to true if categorization is to be turned off; to nil if the
default behavior should apply.

Accepted values for |nocat= are the text strings:
        'yes', 'y', 'true', 't', on, '1'                -- [[Module:Yesno]]
returns logical true for all of these; false or nil else
for |cat=
        'no', 'n', 'false', 'f', 'off', '0'               --
```

```
[[Module:Yesno]] returns logical false for all of these; true or nil else

]=]

local function validate_cat_args (args)
        if not (args.nocat or args.cat) then
-- both are nil, so categorize
                return;
        end
        if false == yesno (args.cat) or true == yesno (args.nocat) then
                args.nocat = true;
-- set to true when args.nocat is affirmative; nil else (as if the parameter
were not set in the template)
        else
-- args.nocat is the parameter actually used.
                args.nocat = nil;
        end
end


--[[--------------------------< I N _ A R R A Y >--------------------------
----------------------------------

Whether needle is in haystack

]]

local function in_array ( needle, haystack )
        if needle == nil then
                return false;
        end
        for n,v in ipairs( haystack ) do
                if v == needle then
                        return n;
                end
        end
        return false;
end


--[[-------------------------< F O R M A T _ I E T F _ T A G >-------------
----------------------------------

prettify ietf tags to use recommended subtag formats:
        code: lower case
        script: sentence case
        region: upper case
        variant: lower case

]]
```

```
local function format_ietf_tag (code, script, region, variant)
	local out = {};

	table.insert (out, code:lower());
	if is_set (script) then
		script = script:lower():gsub ('^%a', string.upper);
		table.insert (out, script);
	end

	if is_set (region) then
		table.insert (out, region:upper());
	end
	if is_set (variant) then
		table.insert (out, variant:lower());
	end
	return table.concat (out, '-');
end


--[[--------------------------< G E T _ I E T F _ P A R T S >--------------
-----------------------------------

extracts and returns IETF language tag parts:
	primary language subtag (required) - 2 or 3 character IANA language
code
	script subtag - four character IANA script code
	region subtag - two-letter or three digit IANA region code
	variant subtag - four digit or 5-8 alnum variant code
	private subtag - x- followed by 1-8 alnum private code; only
supported with the primary language tag

in any one of these forms
	lang                                          lang-variant
	lang-script                                  lang-script-variant
	lang-region                                  lang-region-variant
	lang-script-region                lang-script-region-variant
	lang-x-private
each of lang, script, region, variant, and private, when used, must be valid

Languages with both two- and three-character code synonyms are promoted to
the two-character synonym because
the IANA registry file omits the synonymous three-character code; we cannot
depend on browsers understanding
the synonymous three-character codes in the lang= attribute.

For {{lang-xx}} templates, the parameters |script=, |region=, and |variant=
are supported (not supported in {{lang}}
because those parameters are superfluous to the IETF subtags in |code=)

returns six  values; all lower case.  Valid parts are returned as themselves;
omitted parts are returned as empty strings, invalid
```

```
parts are returned as nil; the sixth returned item is an error message (if an
error detected) or nil.

see http://www.rfc-editor.org/rfc/bcp/bcp47.txt section 2.1

]]

local function get_ietf_parts (source, args_script, args_region,
args_variant)
	local code, script, region, variant, private;
-- ietf tag parts

	if not is_set (source) then
		return nil, nil, nil, nil, nil, 'missing language tag';
	end

	local pattern = {
-- table of tables holding acceptibe ietf tag patterns and short names of the
ietf part captured by the pattern
		{'^(%a%a%a?)%-(%a%a%a%a)%-(%a%a)%-(%d%d%d%d)$', 's', 'r',
'v'},                                   -- 1 -  ll-Ssss-RR-variant (where
variant is 4 digits)
		{'^(%a%a%a?)%-(%a%a%a%a)%-(%d%d%d)%-(%d%d%d%d)$', 's', 'r',
'v'},                                  -- 2 -  ll-Ssss-DDD-variant (where
region is 3 digits; variant is 4 digits)
		{'^(%a%a%a?)%-(%a%a%a%a)%-(%a%a)%-(%w%w%w%w%w%w%w?%w?%w?)$',
's', 'r', 'v'},                -- 3 -  ll-Ssss-RR-variant (where variant is
5-8 alnum characters)
		{'^(%a%a%a?)%-(%a%a%a%a)%-(%d%d%d)%-(%w%w%w%w%w%w%w?%w?%w?)$',
's', 'r', 'v'},        -- 4 -  ll-Ssss-DDD-variant (where region is 3 digits;
variant is 5-8 alnum characters)
		{'^(%a%a%a?)%-(%a%a%a%a)%-(%d%d%d%d)$', 's', 'v'},
-- 5 -  ll-Ssss-variant (where variant is 4 digits)
		{'^(%a%a%a?)%-(%a%a%a%a)%-(%w%w%w%w%w%w%w?%w?%w?)$', 's', 'v'},
-- 6 -  ll-Ssss-variant (where variant is 5-8 alnum characters)
		{'^(%a%a%a?)%-(%a%a)%-(%d%d%d%d)$', 'r', 'v'},
-- 7 -  ll-RR-variant (where variant is 4 digits)
		{'^(%a%a%a?)%-(%d%d%d)%-(%d%d%d%d)$', 'r', 'v'},
-- 8 -  ll-DDD-variant (where region is 3 digits; variant is 4 digits)
		{'^(%a%a%a?)%-(%a%a)%-(%w%w%w%w%w%w%w?%w?%w?)$', 'r', 'v'},
-- 9 -  ll-RR-variant (where variant is 5-8 alnum characters)
		{'^(%a%a%a?)%-(%d%d%d)%-(%w%w%w%w%w%w%w?%w?%w?)$', 'r', 'v'},
-- 10 - ll-DDD-variant (where region is 3 digits; variant is 5-8 alnum
characters)
		{'^(%a%a%a?)%-(%d%d%d%d)$', 'v'},
-- 11 - ll-variant (where variant is 4 digits)
		{'^(%a%a%a?)%-(%w%w%w%w%w%w%w?%w?%w?)$', 'v'},
-- 12 - ll-variant (where variant is 5-8 alnum characters)
		{'^(%a%a%a?)%-(%a%a%a%a)%-(%a%a)$', 's', 'r'},
-- 13 - ll-Ssss-RR
		{'^(%a%a%a?)%-(%a%a%a%a)%-(%d%d%d)$', 's', 'r'},
```

```
-- 14 - ll-Ssss-DDD (region is 3 digits)
                {'^(%a%a%a?)%-(%a%a%a%a)$', 's'},
-- 15 - ll-Ssss
                {'^(%a%a%a?)%-(%a%a)$', 'r'},
-- 16 - ll-RR
                {'^(%a%a%a?)%-(%d%d%d)$', 'r'},
-- 17 - ll-DDD (region is 3 digits)
                {'^(%a%a%a?)$'},
-- 18 - ll
                {'^(%a%a%a?)%-x%-(%w%w?%w?%w?%w?%w?%w?%w?%w?)$', 'p'},
-- 19 - ll-x-pppppppp (private is 1-8 alnum characters)
                }

        local t = {};
-- table of captures; serves as a translator between captured ietf tag parts
and named variables

        for i, v in ipairs (pattern) do
-- spin through the pattern table looking for a match
                local c1, c2, c3, c4;
-- captures in the 'pattern' from the pattern table go here
                c1, c2, c3, c4 = source:match (pattern[i][1]);
-- one or more captures set if source matches pattern[i])
                        if c1 then
-- c1 always set on match
                                code = c1;
-- first capture is always code
                                t = {
                                        [pattern[i][2] or 'x'] = c2,
-- fill the table of captures with the rest of the captures
                                        [pattern[i][3] or 'x'] = c3,
-- take index names from pattern table and assign sequential captures
                                        [pattern[i][4] or 'x'] = c4,
-- index name may be nil in pattern[i] table so "or 'x'" spoofs a name for
this index in this table
                                        };
                                script = t.s or '';
-- translate table contents to named variables;
                                region = t.r or '';
-- absent table entries are nil so set named ietf parts to empty string for
concatenation
                                variant= t.v or '';
                                private = t.p or '';
                                break;
-- and done
                        end
        end

        if not code then
                return nil, nil, nil, nil, nil, table.concat ({'unrecognized
language tag: ', source});               -- don't know what we got but it is
```

```
malformed
        end

        code = code:lower();
-- ensure that we use and return lower case version of this
        if not (lang_data.override[code] or lang_name_table.lang[code]) then
                return nil, nil, nil, nil, nil, table.concat ({'unrecognized
language code: ', code});                -- invalid language code, don't know
about the others (don't care?)
        end
        if synonym_table[code] then
-- if 639-2/639-2T code has a 639-1 synonym
                table.insert (maint_cats, table.concat ({'Lang and lang-xx
code promoted to ISO 639-1|', code}));
                table.insert (maint_msgs, table.concat ({'code: ', code, '
promoted to code: ', synonym_table[code]}));
                code = synonym_table[code];
-- use the synonym
        end

        if is_set (script) then
                if is_set (args_script) then
                        return code, nil, nil, nil, nil, 'redundant script
tag';                        -- both code with script and |script= not
allowed
                end
        else
                script = args_script or '';
-- use args.script if provided
        end

        if is_set (script) then
                script = script:lower();
-- ensure that we use and return lower case version of this
                if not lang_name_table.script[script] then
                        return code, nil, nil, nil, nil, table.concat
({'unrecognized script: ', script, ' for code: ', code});        -- language
code ok, invalid script, don't know about the others (don't care?)
                end
        end
        if lang_name_table.suppressed[script] then
-- ensure that code-script does not use a suppressed script
                if in_array (code, lang_name_table.suppressed[script]) then
                        return code, nil, nil, nil, nil, table.concat
({'script: ', script, ' not supported for code: ', code});        -- language
code ok, script is suppressed for this code
                end
        end

        if is_set (region) then
                if is_set (args_region) then
```

```
                        return code, nil, nil, nil, nil, 'redundant region
tag';                       -- both code with region and |region= not
allowed
                end
        else
                region = args_region or '';
-- use args.region if provided
        end

        if is_set (region) then
                region = region:lower();
-- ensure that we use and return lower case version of this
                if not lang_name_table.region[region] then
                        return code, script, nil, nil, nil, table.concat
({'unrecognized region: ', region, ' for code: ', code});
                end
        end
        if is_set (variant) then
                if is_set (args_variant) then
                        return code, nil, nil, nil, nil, 'redundant variant
tag';                       -- both code with variant and |variant= not
allowed
                end
        else
                variant = args_variant or '';
-- use args.variant if provided
        end

        if is_set (variant) then
                variant = variant:lower();
-- ensure that we use and return lower case version of this
                if not lang_name_table.variant[variant] then
-- make sure variant is valid
                        return code, script, region, nil, nil, table.concat
({'unrecognized variant: ', variant});
                end
-- does this duplicate/replace tests in lang() and lang_xx()?
                if is_set (script) then
-- if script set it must be part of the 'prefix'
                        if not in_array (table.concat ({code, '-', script}),
lang_name_table.variant[variant]['prefixes']) then
                                return code, script, region, nil, nil,
table.concat ({'unrecognized variant: ', variant, ' for code-script pair: ',
code, '-', script});
                        end
                else
                        if not in_array (code,
lang_name_table.variant[variant]['prefixes']) then
                                return code, script, region, nil, nil,
table.concat ({'unrecognized variant: ', variant, ' for code: ', code});
                        end
```

```
                end
        end
        if is_set (private) then
                private = private:lower();
-- ensure that we use and return lower case version of this
                if not lang_data.override[table.concat ({code, '-x-',
private})] then        -- make sure private tag is valid; note that index
                         return code, script, region, nil, nil, table.concat
({'unrecognized private tag: ', private});
                end
        end
        return code, script, region, variant, private, nil;
-- return the good bits; make sure that msg is nil
end


--[[--------------------------< M A K E _ E R R O R _ M S G >----------------
----------------------------------

assembles an error message from template name, message text, help link, and
error category.

]]

local function make_error_msg (msg, args, template)
        local out = {};
        local category;
        if 'transl' == template then
                category = 'transl';
        else
                category = 'lang and lang-xx'
        end
        table.insert (out, table.concat ({'&#x5B;', args.text or 'undefined',
'&#x5D; '}));        -- for error messages output args.text if available
        table.insert (out, table.concat ({'<span style=\"font-size:100%;
font-style:normal;\" class=\"error\">error: {{', template, '}}: '}));
        table.insert (out, msg);
        table.insert (out, table.concat ({' ([[:Category:', category, '
template errors|help]])'}));
        table.insert (out, '</span>');
        if (0 == namespace) and not args.nocat then
-- only categorize in article space
                table.insert (out, table.concat ({'[[Category:', category, '
template errors]]'}));
        end

        return table.concat (out);
end

--[=[-------------------------< M A K E _ W I K I L I N K >------------------
----------------------------------
```

```
Makes a wikilink; when both link and display text is provided, returns a
wikilink in the form [[L|D]]; if only
link is provided, returns a wikilink in the form [[L]]; if neither are
provided or link is omitted, returns an
empty string.

]=]

local function make_wikilink (link, display)
	if is_set (link) then
		if is_set (display) then
			return table.concat ({'[[', link, '|', display,
']]'});
		else
			return table.concat ({'[[', link, ']]'});
		end
	else
		return '';
	end
end


--[[--------------------------< D I V _ M A R K U P _ A D D >----------------
----------------------------------

adds <i> and </i> tags to list-item text or to implied <p>..</p> text.  mixed
not supported

]]

local function div_markup_add (text, style)
local implied_p = {};

	if text:find ('^\n[%*:;#]') then
-- look for list markup; list markup must begin at start of text
		if 'italic' == style then
			return mw.ustring.gsub (text, '(\n[%*:;#]+)([^\n]+)',
'%1<i>%2</i>');		-- insert italic markup at each list item
		else
			return text;
		end
	end

	if text:find ('\n+') then
-- look for any number of \n characters in text
		text = text:gsub ('([^\n])\n([^\n])', '%1 %2');
-- replace single newline characters with a space character which mimics
mediawiki

		if 'italic' == style then
			text = text:gsub('[^\n]+', '<p><i>%1</i></p>');
```

```
-- insert p and italic markup tags at each impled p (two or more consecutive
'\n\n' sequences)
                else
                        text = text:gsub ('[^\n]+', '<p>%1</p>');
-- insert p markup at each impled p
                        text = text:gsub ('\n', '');
-- strip newline characters
                end
        end

        return text;
end


--[[--------------------------< M A K E _ T E X T _ H T M L >---------------
---------------------------------

Add the html markup to text according to the type of content that it is:
<span> or <i> tags for inline content or
<div> tags for block content

]]

local function make_text_html (code, text, tag, rtl, style, size, language)
        local html = {};
        local style_added = '';

        if text:match ('^%*') then
                table.insert (html, '&#42;');
-- move proto language text prefix outside of italic markup if any; use
numeric entity because plain splat confuses MediaWiki
                text = text:gsub ('^%*', '');
-- remove the splat from the text
        end

        if 'span' == tag then
-- default html tag for inline content
                if 'italic' == style then
-- but if italic
                        tag = 'i';
-- change to <i> tags
                end
        else
-- must be div so go
                text = div_markup_add (text, style);
-- handle implied <p>, implied <p> with <i>, and list markup (*;:#) with <i>
        end

        table.insert (html, table.concat ({'<', tag}));
-- open the <i>, <span>, or <div> html tag
        table.insert (html, table.concat ({' lang="', code, '\"'}));
```

```
-- add language attribute

        if rtl or unicode.is_rtl(text) then
                table.insert (html, ' dir="rtl"');
-- add direction attribute for right to left languages
        end

        if 'normal' == style then
-- when |italic=no
                table.insert (html, ' style=\"font-style:normal;');
-- override external markup, if any
                style_added = '\"';
-- remember that style attribute added and is not yet closed
        end

        if is_set (size) then
-- when |size=<something>
                if is_set (style_added) then
                        table.insert (html, table.concat ({' font-size:',
size, ';'}));                -- add when style attribute already inserted
                else
                        table.insert (html, table.concat ({' style=\"font-
size:', size, ';'}));        -- create style attribute
                        style_added = '\"';
-- remember that style attribute added and is not yet closed
                end
        end

        if is_set (language) then
                table.insert (html, table.concat ({style_added, ' title=\"',
language}));        --start the title text
                if language:find ('languages') then
                        table.insert (html, ' collective text\"');
-- for collective languages
                else
                        table.insert (html, ' language text\"');
-- for individual languages
                end
                table.insert (html, '>');
-- close the opening html tag
        else
                table.insert (html, table.concat ({style_added, '>'}));
-- close the style attribute and close opening html tag
        end
        table.insert (html, text);
-- insert the text

        table.insert (html, table.concat ({'</', tag, '>'}));
-- close the <i>, <span>, or <div> html tag

        if rtl then
```

```
-- legacy; shouldn't be necessary because all of the rtl text is wrapped
inside an html tag with dir="rtl" attribute
                table.insert (html, '&lrm;');
-- make sure the browser knows that we're at the end of the rtl
        end
        return table.concat (html);
-- put it all together and done
end


--[=[-------------------------< M A K E _ C A T E G O R Y >-----------------
---------------------------------

For individual language, <language>, returns:
        [[Category:Articles containing <language>-language text]]

for English:
        [[Category:Articles containing explicitly cited English-language
text]]
for artificial languages (code: art)
        [[Category:Articles containing constructed-language text]]

for ISO 639-2 collective languages (and for 639-1 bh):
        [[Category:Articles with text from the <language> languages
collective]]

]=]

local function make_category (code, language_name, nocat)
        local cat = {};
        if (0 ~= namespace) or nocat then
-- only categorize in article space
                return '';
-- return empty string for concatenation
        end
        if language_name:find ('languages') then
                return table.concat ({'[[Category:Articles with text from the
', language_name, ' collective]]'});
        end
        table.insert (cat, '[[Category:Articles containing ');
        if 'en' == code then
                table.insert (cat, 'explicitly cited English');
        elseif 'art' == code then
                table.insert (cat, 'constructed')
        else
                table.insert (cat, language_name);
        end
        table.insert (cat, '-language text]]');

        return table.concat (cat);
end
```

```
--[[--------------------------< M A K E _ T R A N S L I T >----------------
--------------------------------

return translit <i lang=xx-Latn>...</i> where xx is the language code; else
return empty string

The value |script= is not used in {{transl}} for this purpose; instead it
uses |code.  Because language scripts
are listed in the {{transl}} switches they are included in the data tables.
The script parameter is introduced
at {{Language with name and transliteration}}.  If |script= is set, this
function uses it in preference to code.

To avoid confusion, in this module and the templates that use it, the
transliteration script parameter is renamed
to be |translit-script= (in this function, tscript)

This function is used by both lang_xx() and transl()
	lang_xx() always provides code, language_name, and translit; may
provide tscript; never provides style
	transl() always provides language_name, translit, and one of code or
tscript, never both; always provides style

For {{transl}}, style only applies when a language code is provided
]]

local function make_translit (code, language_name, translit, std, tscript,
style)
	local title;
	local tout = {};
	local title_table = lang_data.translit_title_table;
-- table of transliteration standards and the language codes and scripts that
apply to those standards
	if is_set (code) then
-- when a language code is provided (always with {{lang-xx}} templates, not
always with {{transl}})
		if not style then
-- nil for the default italic style
			table.insert (tout, "<i lang=\"");
-- so use <i> tag
		else
			table.insert (tout, table.concat ({'<span
style=\"font-style:', style, '\" lang=\"'}));	-- non-standard style,
construct a span tag for it
		end
		table.insert (tout, code);
		table.insert (tout, "-Latn\" title=\"");
-- transliterations are always Latin script
	else
		table.insert (tout, "<span title=\"");
-- when no language code: no lang= attribute, not italic ({{transl}} only)
```

```
        end
        std = std and std:lower();
-- lower case for table indexing
        if not is_set (std) and not is_set (tscript) then
-- when neither standard nor script specified
                table.insert (tout, language_name);
-- write a generic tool tip
                if not language_name:find ('languages') then
-- collective language names (plural 'languages' is part of the name)
                        table.insert (tout, '-language')
-- skip this text (individual and macro languages only)
                end
                table.insert (tout, ' romanization');
-- finish the tool tip; use romanization when neither script nor standard
supplied
        elseif is_set (std) and is_set (tscript) then
-- when both are specified
                if title_table[std] then
-- and if standard is legitimate
                        if title_table[std][tscript] then
-- and if script for that standard is legitimate
                                table.insert (tout, table.concat
({title_table[std][tscript:lower()], ' (',
lang_name_table.script[tscript][1], ' script) transliteration'}));        --
add the appropriate text to the tool tip
                        else
                                table.insert (tout,
title_table[std]['default']);                                   -- use the
default if script not in std table; TODO: maint cat? error message because
script not found for this standard?
                        end
                else
                        return '';
-- invalid standard, setup for error message
                end

        elseif is_set (std) then
-- translit-script not set, use language code
                if not title_table[std] then return ''; end
-- invalid standard, setup for error message
                if title_table[std][code] then
-- if language code is in the table (transl may not provide a language code)
                        table.insert (tout, table.concat
({title_table[std][code:lower()], ' (', lang_name_table.lang[code][1], '
language) transliteration'}));        -- add the appropriate text to the tool
tip
                else
-- code doesn't match
                        table.insert (tout, title_table[std]['default']);
-- so use the standard's default
                end
```

```
        else
-- here if translit-script set but translit-std not set
                if title_table['no_std'][tscript] then
                        table.insert (tout, title_table['no_std'][tscript]);
-- use translit-script if set
                elseif title_table['no_std'][code] then
                        table.insert (tout, title_table['no_std'][code]);
-- use language code
                else
                        if is_set (tscript) then
                                table.insert (tout, table.concat
({language_name, '-script transliteration'}));        -- write a script tool
tip
                        elseif is_set (code) then
                                if not language_name:find ('languages') then
-- collective language names (plural 'languages' is part of the name)
                                        table.insert (tout, '-language')
-- skip this text (individual and macro languages only)
                                end
                                table.insert (tout, ' transliteration');
-- finish the tool tip
                        else
                                table.insert (tout, ' transliteration');
-- generic tool tip (can we ever get here?)
                        end
                end
        end

        table.insert (tout, '">');
        table.insert (tout, translit);
        if is_set (code) and not style then
-- when a language code is provided (always with {{lang-xx}} templates, not
always with {{transl}})
                table.insert (tout, "</i>");
-- close the italic tag
        else
                table.insert (tout, "</span>");
-- no language code so close the span tag
        end
        return table.concat (tout);
end


--[[--------------------------< V A L I D A T E _ T E X T >------------------
----------------------------------

This function checks the content of args.text and returns empty string if
nothing is amiss else it returns an
error message.  The tests are for empty or missing text and for improper or
disallowed use of apostrophe markup.
```

```
Italic rendering is controlled by the |italic= template parameter so italic
markup should never appear in args.text
either as ''itself'' or as '''''bold italic''''' unless |italic=unset or
|italic=invert.

]]

local function validate_text (template, args)
	if not is_set (args.text) then
		return make_error_msg ('no text', args, template);
	end

	if args.text:find ("%f[\']\'\'\'\'%f[^\']") or args.text:find
("\'\'\'\'\'[\']+") then	-- because we're looking, look for 4
appostrophes or 6+ appostrophes
		return make_error_msg ('text has malformed markup', args,
template);
	end

	local style = args.italic;

	if ('unset' ~= style) and ('invert' ~=style) then
		if args.text:find ("%f[\']\'\'%f[^\']") or args.text:find
("%f[\']\'\'\'\'\'%f[^\']") then	-- italic but not bold, or bold
italic
			return make_error_msg ('text has italic markup',
args, template);
		end
	end
end


--[[--------------------------< R E N D E R _ M A I N T >--------------------
---------------------------------

render mainenance messages and categories

]]

local function render_maint(nocat)
	local maint = {};
	if 0 < #maint_msgs then
-- when there are maintenance messages
		table.insert (maint, table.concat ({'<span class="lang-
comment" style="font-style:normal; display:none; color:#33aa33; margin-
left:0.3em">'}));	-- opening <span> tag
		for _, msg in ipairs (maint_msgs) do
			table.insert (maint, table.concat ({msg, ' '}));
-- add message strings
		end
		table.insert (maint, '</span>');
```

```
-- close the span
        end
        if (0 < #maint_cats) and (0 == namespace) and not nocat then
-- when there are maintenance categories; article namespace only
                for _, cat in ipairs (maint_cats) do
                        table.insert (maint, table.concat ({'[[Category:',
cat, ']]'}));        -- format and add the categories
                end
        end
        return table.concat (maint);
end


--[[--------------------------< P R O T O _ P R E F I X >--------------------
---------------------------------

for proto languages, text is prefixed with a splat.  We do that here as a
flag for make_text_html() so that a splat
will be rendered outside of italic markup (if used).  If the first character
in text here is already a splat, we
do nothing

proto_param is boolean or nil; true adds splat prefix regardless of language
name; false removes and / or inhibits
regardless of language name; nil does nothing; presumes that the value in
text is correct but removes extra splac

]]

local function proto_prefix (text, language_name, proto_param)
        if false == proto_param then
-- when forced by |proto=no
                return text:gsub ('^%**', '');
-- return text without splat prefix regardless of language name or existing
splat prefix in text
        elseif (language_name:find ('^Proto%-') or (true == proto_param))
then                -- language is a proto or forced by |proto=yes
                return text:gsub ('^%**', '*');
-- prefix proto-language text with a splat; also removes duplicate prefixing
splats
        end
        return text:gsub ('^%*+', '*');
-- return text unmolested except multiple splats reduced to one splat
end


--[[--------------------------< H A S _ P O E M _ T A G >--------------------
---------------------------------

looks for a poem strip marker in text; returns true when found; false else
```

```
auto-italic detection disabled when text has poem stripmarker because it is
not possible for this code to know
the content that will replace the stripmarker.

]]

local function has_poem_tag (text)
        return text:find ('\127[^\127]*UNIQ%-%-poem%-[%a%d]+%-
QINU[^\127]*\127') and true or false;
end


--[[--------------------------< H T M L _ T A G _ S E L E C T >--------------
----------------------------------

Inspects content of and selectively trims text.  Returns text and the name of
an appropriate html tag for text.

If text contains:
        \n\n        text has implied <p>..</p> tags - trim leading and
trailing whitespace and return
If text begins with list markup:
        \n*                unordered
        \n;                definition
        \n:                definition
        \n#                ordered
trim all leading whitespace except  \n and trim all trailing whitespace

If text contains <poem>...</poem> stripmarker, return text unmodified and
choose <div>..</div> tags because
the stripmarker is replaced with text wrapped in <div>..</div> tags.

]]

local function html_tag_select (text)
        local tag;
        if has_poem_tag (text) then
-- contains poem stripmarker (we can't know the content of that)
                tag = 'div';
-- poem replacement is in div tags so lang must use div tags
        elseif mw.text.trim (text):find ('\n\n+') then
-- contains implied p tags
                text = mw.text.trim (text);
-- trim leading and trailing whitespace characters
                tag = 'div';
-- must be div because span may not contain p tags (added later by
MediaWiki); poem replacement is in div tags
        elseif text:find ('\n[%*:;%#]') then
-- if text has list markup
                text = text:gsub ('^[\t\r\f ]*', ''):gsub ('%s*$', '');
-- trim all whitespace except leading newline character '\n'
```

```
                tag = 'div';
-- must be div because span may not contain ul, dd, dl, ol tags (added later
by MediaWiki)
        else
                text = mw.text.trim (text);
-- plain text
                tag = 'span';
-- so span is fine
        end
        return text, tag;
end


--[[--------------------------< V A L I D A T E _ P R O T O >---------------
---------------------------------

validates value assigned to |proto=; permitted values are yes and no; yes
returns as true, no returns as false,
empty string (or parameter omitted) returns as nil; any other value returns
as nil with a second return value of
true indicating that some other value has been assigned to |proto=

]]

local function validate_proto (proto_param)
        if 'yes' == proto_param then
                return true;
        elseif 'no' == proto_param then
                return false;
        elseif is_set (proto_param) then
                return nil, true;
-- |proto= something other than 'yes' or 'no'
        else
                return nil;
-- missing or empty
        end
end


--[[--------------------------< L A N G >----------------------------------
---------------------------------

entry point for {{lang}}

there should be no reason to set parameters in the {{lang}} {{#invoke:}}
        <includeonly>{{#invoke:lang|lang}}</includeonly>

parameters are received from the template's frame (parent frame)

]]
```

```
local function _lang (args)
	local out = {};
	local language_name;
-- used to make category names
	local subtags = {};
-- IETF subtags script, region, variant, and private
	local code;
-- the language code
	local msg;
-- for error messages
	local tag = 'span';
-- initial value for make_text_html()
	local template = args.template or 'lang';

	if args[1] and args.code then
		return make_error_msg ('conflicting: {{{1}}} and
&#124;code=', args, template);
	else
		args.code = args[1] or args.code;
-- prefer args.code
	end

	if args[2] and args.text then
		return make_error_msg ('conflicting: {{{2}}} and
&#124;text=', args, template);
	else
		args.text = args[2] or args.text;
-- prefer args.text
	end
	msg = validate_text (template, args);
-- ensure that |text= is set
	if is_set (msg) then
-- msg is an already-formatted error message
		return msg;
	end
	args.text, tag = html_tag_select (args.text);
-- inspects text; returns appropriate html tag with text trimmed accordingly

	validate_cat_args (args);
-- determine if categorization should be suppressed

	args.rtl = args.rtl == 'yes';
-- convert to boolean: 'yes' -> true, other values -> false

	args.proto, msg = validate_proto (args.proto);
-- return boolean, or nil, or nil and error message flag
	if msg then
		return make_error_msg (table.concat ({'invalid &#124;proto=:
', args.proto}), args, template);
	end
```

```
	code, subtags.script, subtags.region, subtags.variant,
subtags.private, msg = get_ietf_parts (args.code);		-- |script=,
|region=, |variant= not supported because they should be part of args.code
({{{1}}} in {{lang}})

	if msg then
		return make_error_msg ( msg, args, template);
	end

	args.italic, msg = validate_italic (args);
	if msg then
		return make_error_msg (msg, args, template);
	end

	if nil == args.italic then
-- nil when |italic= absent or not set or |italic=default; args.italic
controls
		if ('latn' == subtags.script) or
-- script is latn
			(this_wiki_lang ~= code and not is_set
(subtags.script) and not has_poem_tag (args.text) and unicode.is_Latin
(args.text)) then -- text not this wiki's language, no script specified and
not in poem markup but is wholly latn script (auto-italics)
				args.italic = 'italic';
-- DEFAULT for {{lang}} templates is upright; but if latn script set for
font-style:italic
		else
			args.italic = 'inherit';
-- italic not set; script not latn; inherit current style
		end
	end
	if is_set (subtags.script) then
-- if script set, override rtl setting
		if in_array (subtags.script, lang_data.rtl_scripts) then
			args.rtl = true;
-- script is an rtl script
		else
			args.rtl = false;
-- script is not an rtl script
		end
	end

	args.code = format_ietf_tag (code, subtags.script, subtags.region,
subtags.variant);		-- format to recommended subtag styles; private
omitted because private

	subtags.private = subtags.private and table.concat ({code, '-x-',
subtags.private}) or nil;		-- assemble a complete private ietf
subtag; args.code does not get private subtag

	if is_set (subtags.private) and lang_data.override[subtags.private]
```

```
then        -- get the language name for categorization
                language_name = lang_data.override[subtags.private][1];
-- first look for private use tag language name
        elseif lang_data.override[code] then
                language_name = lang_data.override[code][1]
-- then language names taken from the override table
        elseif lang_name_table.lang[code] then
                language_name = lang_name_table.lang[code][1];
-- table entries sometimes have multiple names, always take the first one
        end

        if 'invert' == args.italic and 'span' == tag then
-- invert only supported for in-line content
                args.text = invert_italics (args.text)
        end

        args.text = proto_prefix (args.text, language_name, args.proto);
-- prefix proto-language text with a splat

        table.insert (out, make_text_html (args.code, args.text, tag,
args.rtl, args.italic, args.size, language_name));
        table.insert (out, make_category (code, language_name, args.nocat));
        table.insert (out, render_maint(args.nocat));
-- maintenance messages and categories

        return table.concat (out);
-- put it all together and done
end


--[[--------------------------< L A N G >----------------------------------
----------------------------------

entry point for {{lang}}

there should be no reason to set parameters in the {{lang}} {{#invoke:}}
        <includeonly>{{#invoke:lang|lang}}</includeonly>

parameters are received from the template's frame (parent frame)

]]

local function lang (frame)
        local args = getArgs (frame, {
-- this code so that we can detect and handle wiki list markup in text
                valueFunc = function (key, value)
                        if 2 == key or 'text' == key then
-- the 'text' parameter; do not trim wite space
                                return value;
-- return untrimmed 'text'
                        elseif value then
```

```
-- all other values: if the value is not nil
                                value = mw.text.trim (value);
-- trim whitespace
                                if '' ~= value then
-- empty string when value was only whitespace
                                        return value;
                                end
                        end
                        return nil;
-- value was empty or contained only whitespace
                end
-- end of valueFunc
                });

        return _lang (args);
end


--[[--------------------------< L A N G _ X X >----------------------------
----------------------------------

For the {{lang-xx}} templates, the only parameter required to be set in the
template is the language code.  All
other parameters can, usually should, be written in the template call.  For
{{lang-xx}} templates for languages
that can have multiple writing systems, it may be appropriate to set |script=
as well.

For each {{lang-xx}} template choose the appropriate entry-point function so
that this function knows the default
styling that should be applied to text.

For normal, upright style:
        <includeonly>{{#invoke:lang|lang_xx_inherit|code=xx}}</includeonly>
For italic style:
        <includeonly>{{#invoke:lang|lang_xx_italic|code=xx}}</includeonly>

All other parameters should be received from the template's frame (parent
frame)

Supported parameters are:
        |code = (required) the IANA language code
        |script = IANA script code; especially for use with languages that
use multiple writing systems
        |region = IANA region code
        |variant = IANA variant code
        |text = (required) the displayed text in language specified by code
        |link = boolean false ('no') does not link code-spcified language
name to associated language article
        |rtl = boolean true ('yes') identifies the language specified by code
as a right-to-left language
```

```
        |nocat = boolean true ('yes') inhibits normal categorization; error
categories are not affected
        |cat = boolian false ('no') opposite form of |nocat=
        |italic = boolean true ('yes') renders displayed text in italic font;
boolean false ('no') renders displayed text in normal font; not set renders
according to initial_style_state
        |lit = text that is a literal translation of text
        |label = 'none' to suppress all labeling (language name, 'translit.',
'lit.')
                                any other text replaces language-name label -
automatic wikilinking disabled
        for those {{lang-xx}} templates that support transliteration (those
templates where |text= is not entirely latn script):
        |translit = text that is a transliteration of text
        |translit-std = the standard that applies to the transliteration
        |translit-script = ISO 15924 script name; falls back to code

For {{lang-xx}}, the positional parameters are:
        {{{1}}}        text
        {{{2}}}        transliterated text
        {{{3}}}        literal translation text
no other positional parameters are allowed

]]

local function _lang_xx (args)
        local out = {};
        local language_name;
-- used to make display text, article links
        local category_name;
-- same as language_name except that it retains any parenthetical
disambiguators (if any) from the data set
        local subtags = {};
-- IETF subtags script, region, and variant
        local code;
-- the language code

        local translit_script_name;
-- name associated with IANA (ISO 15924) script code
        local translit;
        local translit_title;
        local msg;
-- for error messages
        local tag = 'span';
-- initial value for make_text_html()
        local template = args.template or 'lang-xx';

        if args[1] and args.text then
                return make_error_msg ('conflicting: {{{1}}} and
&#124;text=', args, template);
        else
```

```
                args.text = args[1] or args.text;
-- prefer args.text
        end
        msg = validate_text (template, args);
-- ensure that |text= is set, does not contain italic markup and is protected
from improper bolding
        if is_set (msg) then
                return msg;
        end

        args.text, tag = html_tag_select (args.text);
-- inspects text; returns appropriate html tag with text trimmed accordingly

        if args[2] and args.translit then
                return make_error_msg ('conflicting: {{{2}}} and
&#124;translit=', args, template);
        else
                args.translit = args[2] or args.translit
-- prefer args.translit
        end
        if args[3] and (args.translation or args.lit) then
                return make_error_msg ('conflicting: {{{3}}} and &#124;lit=
or &#124;translation=', args, template);
        elseif args.translation and args.lit then
                return make_error_msg ('conflicting: &#124;lit= and
&#124;translation=', args, template);
        else
                args.translation = args[3] or args.translation or args.lit;
-- prefer args.translation
        end

        if args.links and args.link then
                return make_error_msg ('conflicting: &#124;links= and
&#124;link=', args, template);
        else
                args.link = args.link or args.links;
-- prefer args.link
        end

        validate_cat_args (args);
-- determine if categorization should be suppressed

        args.rtl = args.rtl == 'yes';
-- convert to boolean: 'yes' -> true, other values -> false

        code, subtags.script, subtags.region, subtags.variant,
subtags.private, msg = get_ietf_parts (args.code, args.script, args.region,
args.variant);        -- private omitted because private

        if msg then
-- if an error detected then there is an error message
```

```
                return make_error_msg (msg, args, template);
        end
        args.italic, msg = validate_italic (args);
        if msg then
                return make_error_msg (msg, args, template);
        end

        if nil == args.italic then
-- args.italic controls
                if is_set (subtags.script) then
                        if 'latn' == subtags.script then
                                args.italic = 'italic';
-- |script=Latn; set for font-style:italic
                        else
                                args.italic = initial_style_state;
-- italic not set; script is not latn; set for font-
style:<initial_style_state>
                        end
                else
                        args.italic = initial_style_state;
-- here when |italic= and |script= not set; set for font-
style:<initial_style_state>
                end
        end
        if is_set (subtags.script) then
-- if script set override rtl setting
                if in_array (subtags.script, lang_data.rtl_scripts) then
                        args.rtl = true;
-- script is an rtl script
                else
                        args.rtl = false;
-- script is not an rtl script
                end
        end

        args.proto, msg = validate_proto (args.proto);
-- return boolean, or nil, or nil and error message flag
        if msg then
                return make_error_msg (table.concat ({'invalid &#124;proto=:
', args.proto}), args, template);
        end

        args.code = format_ietf_tag (code, subtags.script, subtags.region,
subtags.variant);        -- format to recommended subtag styles
        subtags.private = subtags.private and table.concat ({code, '-x-',
subtags.private}) or nil;                -- assemble a complete private ietf
subtag; args.code does not get private subtag

        if is_set (subtags.private) and lang_data.override[subtags.private]
then        -- get the language name for categorization
                language_name = lang_data.override[subtags.private][1];
```

```
-- first look for private use tag language name
        elseif lang_data.override[args.code:lower()] then
-- look for whole IETF tag in override table
                language_name = lang_data.override[args.code:lower()][1];
-- args.code:lower() because format_ietf_tag() returns mixed case
        elseif lang_data.override[code] then
-- not there so try basic language code
                language_name = lang_data.override[code][1];
        elseif not is_set (subtags.variant) then
                if lang_name_table.lang[code] then
                        language_name = lang_name_table.lang[code][1];
-- table entries sometimes have multiple names, always take the first one
                end
        else
-- TODO: is this the right thing to do: take language display name from
variants table?
                if lang_name_table.variant[subtags.variant] then
-- TODO: there is some discussion at Template talk:Lang about having a label
parameter for use when variant name is not desired among other things
                        language_name =
lang_name_table.variant[subtags.variant]['descriptions'][1];        -- table
entries sometimes have multiple names, always take the first one
                end
        end

        category_name = language_name;
-- category names retain IANA parenthetical diambiguators (if any)
        language_name = language_name:gsub ('%s+%b()', '');
-- remove IANA parenthetical disambiguators or qualifiers from names that
have them

        if args.label then
                if 'none' ~= args.label then
                        table.insert (out, table.concat ({args.label, ':
'}));                                   -- custom label
                end
        else
                if 'no' == args.link then
                        table.insert (out, language_name);
-- language name without wikilink
                else
                        if language_name:find ('languages') then
                                table.insert (out, make_wikilink
(language_name));                               -- collective language name
uses simple wikilink
                        elseif lang_data.article_name[code] then
                                table.insert (out, make_wikilink
(lang_data.article_name[code][1], language_name));        -- language name
with wikilink from override data
                        else
                                table.insert (out, make_wikilink
```

```
(language_name .. ' language', language_name));        -- language name with
wikilink
                        end
                end
                table.insert (out, ': ');
-- separator
        end

        if 'invert' == args.italic then
                args.text = invert_italics (args.text)
        end
        args.text = proto_prefix (args.text, language_name, args.proto);
-- prefix proto-language text with a splat

        table.insert (out, make_text_html (args.code, args.text, tag,
args.rtl, args.italic, args.size))

        if is_set (args.translit) and not unicode.is_Latin (args.text) then
-- transliteration (not supported in {{lang}}); not supported when args.text
is wholly latn text (this is an imperfect test)
                table.insert (out, ', ');
-- comma to separate text from translit
                if 'none' ~= args.label then
                        table.insert (out, '<small>');
                        if lang_name_table.script[args['translit-script']]
then                                -- when |translit-script= is set, try to
use the script's name
                                translit_script_name =
lang_name_table.script[args['translit-script'][1]];
                        else
                                translit_script_name = language_name;
-- fall back on language name
                        end
                        translit_title = mw.title.makeTitle (0, table.concat
({'Romanization of ', translit_script_name}));                -- make a title
object
                        if translit_title.exists and ('no' ~= args.link) then
                                table.insert (out, make_wikilink
('Romanization of ' .. translit_script_name or language_name, 'romanized') ..
':');        -- make a wikilink if there is an article to link to
                        else
                                table.insert (out, 'romanized:');
-- else plain text
                        end
                        table.insert (out, ' </small>');
-- close the small tag
                end
                translit = make_translit (args.code, language_name,
args.translit, args['translit-std'], args['translit-script'])
                if is_set (translit) then
                        table.insert (out, translit);
```

```
                else
                        return make_error_msg (table.concat ({'invalid
translit-std: \'', args['translit-std'] or '[missing]'}), args, template);
                end
        end
        if is_set (args.translation) then
-- translation (not supported in {{lang}})
                table.insert (out, ', ');
                if 'none' ~= args.label then
                        table.insert (out, '<small>');
                        if 'no' == args.link then
                                table.insert (out, '<abbr title="literal
translation">lit.</abbr>');
                        else
                                table.insert (out, make_wikilink ('Literal
translation', 'lit.'));
                        end
                        table.insert (out, " </small>");
                end
                table.insert (out, table.concat ({'&#39;', args.translation,
'&#39;'}));        -- use html entities to avoid wiki markup confusion
        end
        table.insert (out, make_category (code, category_name, args.nocat));
        table.insert (out, render_maint(args.nocat));
-- maintenance messages and categories

        return table.concat (out);
-- put it all together and done
end


--[[--------------------------< L A N G _ X X _ A R G S _ G E T >------------
---------------------------------

common function to get args table from {{lang-??}} templates

returns table of args

]]

local function lang_xx_args_get (frame)
        local args = getArgs(frame,
                {
                parentFirst= true,
-- parameters in the template override parameters set in the {{#invoke:}}
                valueFunc = function (key, value)
                        if 1 == key then
-- the 'text' parameter; do not trim wite space
                                return value;
-- return untrimmed 'text'
                        elseif value then
```

```
-- all other values: if the value is not nil
                                value = mw.text.trim (value);
-- trim whitespace
                                if '' ~= value then
-- empty string when value was only whitespace
                                        return value;
                                end
                        end
                        return nil;
-- value was empty or contained only whitespace
                end
-- end of valueFunc
        });

        return args;
end


--[[--------------------------< L A N G _ X X _ I T A L I C >---------------
---------------------------------

Entry point for those {{lang-xx}} templates that call lang_xx_italic().  Sets
the initial style state to italic.

]]

local function lang_xx_italic (frame)
        local args = lang_xx_args_get (frame);
        initial_style_state = 'italic';
        return _lang_xx (args);
end


--[[-------------------------< _ L A N G _ X X _ I T A L I C >-------------
---------------------------------

Entry point ffrom another module.  Sets the initial style state to italic.

]]

local function _lang_xx_italic (args)
        initial_style_state = 'italic';
        return _lang_xx (args);
end


--[[--------------------------< L A N G _ X X _ I N H E R I T >-------------
---------------------------------

Entry point for those {{lang-xx}} templates that call lang_xx_inherit().
Sets the initial style state to inherit.
```

```
]]

local function lang_xx_inherit (frame)
	local args = lang_xx_args_get (frame);

	initial_style_state = 'inherit';
	return _lang_xx (args);
end


--[[--------------------------< _ L A N G _ X X _ I N H E R I T >------------
---------------------------------

Entry point from another module.  Sets the initial style state to inherit.

]]

local function _lang_xx_inherit (args)
	initial_style_state = 'inherit';
	return _lang_xx (args);
end


--[[--------------------------< _ I S _ I E T F _ T A G >--------------------
---------------------------------

Returns true when a language name associated with IETF language tag exists;
nil else.  IETF language tag must be valid.

All code combinations supported by {{lang}} and the {{lang-xx}} templates are
supported by this function.

The purpose of this function is to replace {{#ifexist:Template:ISO 639 name
xx|<exists>|<does not exist>}} in
templates that are better served by using
{{#invoke:lang|name_from_tag|<code>}}

Module entry point from another module

]]

local function _is_ietf_tag (tag)
-- entry point when this module is require()d into another
	local c, s, r, v, p, err;
-- code, script, region, private, error message
	c, s, r, v, p, err = get_ietf_parts (tag);
-- disassemble tag into constituent part and validate
	return ((c and not err) and true) or nil;
-- return true when code portion has a value without error message; nil else
end
```

```
--[[--------------------------< I S _ I E T F _ T A G >--------------------
---------------------------------

Module entry point from an {{#invoke:}}

]]

local function is_ietf_tag (frame)
-- entry point from an {{#invoke:Lang|is_ietf_tag|<ietf tag>}}
	return _is_ietf_tag (frame.args[1]);
-- frame.args[1] is the ietf language tag
end


--[[--------------------------< _ N A M E _ F R O M _ C O D E >-------------
---------------------------------

Returns language name associated with IETF language tag if valid; empty
string else.

All code combinations supported by {{lang}} and the {{lang-xx}} templates are
supported by this function.

Set invoke's |link= parameter to yes to get wikilinked version of the
language name.

Module entry point from another module

]]

local function _name_from_tag (args)
	local subtags = {};
-- IETF subtags script, region, variant, and private
	local raw_code = args[1];
-- save a copy of the input IETF subtag
	local link = 'yes' == args['link'];
-- make a boolean
	local code;
-- the language code
	local msg;
-- gets an error message if IETF language tag is malformed or invalid
	local language_name = '';
	code, subtags.script, subtags.region, subtags.variant,
subtags.private, msg = get_ietf_parts (raw_code);
	if msg then
		local template = (args['template'] and table.concat ({'{{',
args['template'], '}}: '})) or '';	-- make template name (if provided
by the template)
		return table.concat ({'<span style=\"font-size:100%; font-
style:normal;\" class=\"error\">error: ', template, msg, '</span>'});
	end
```

```
        if lang_data.override[raw_code:lower()] then
-- look for whole IETF tag in override table (force lower case)
                language_name = lang_data.override[raw_code:lower()][1];
        elseif lang_data.override[code] then
-- not there so try basic language code in override table
                language_name = lang_data.override[code][1];
        elseif not is_set (subtags.variant) then
                if lang_name_table.lang[code] then
                        language_name = lang_name_table.lang[code][1];
-- table entries sometimes have multiple names, always take the first one
                end
        else
-- TODO: is this the right thing to do: take language display name from
variants table?
                if lang_name_table.variant[subtags.variant] then
-- TODO: there is some discussion at Template talk:Lang about having a label
parameter for use when variant name is not desired among other things
                        language_name =
lang_name_table.variant[subtags.variant]['descriptions'][1];        -- table
entries sometimes have multiple names, always take the first one
                end
        end

        language_name = language_name:gsub ('%s+%b()', '');
-- remove IANA parenthetical disambiguators or qualifiers from names that
have them

        if link then
-- when |link=yes, wikilink the language name
                if language_name:find ('languages') then
                        return make_wikilink (language_name);
-- collective language name uses simple wikilink
                elseif lang_data.article_name[code] then
                        return make_wikilink
(lang_data.article_name[code][1], language_name);        -- language name
with wikilink from override data
                else
                        return make_wikilink (language_name .. ' language',
language_name);        -- language name with wikilink
                end
        end

        return language_name;

end


--[[--------------------------< N A M E _ F R O M _ C O D E >---------------
----------------------------------

Module entry point from an {{#invoke:}}
```

```
]]

local function name_from_tag (frame)
-- entry point from an {{#invoke:Lang|name_from_tag|<ietf
tag>|link=<yes>|template=<template name>}}
        return _name_from_tag (frame.args);
-- pass-on the args table, nothing else
end


--[[--------------------------< _ T A G _ F R O M _ N A M E >---------------
---------------------------------

Returns the ietf language tag associated with the language name.  Spelling of
language name must be correct
according to the spelling in the source tables.  When a standard language
name has a parenthetical disambiguator,
that disambiguator must be omitted (they are not present in the data name-to-
tag tables).

Module entry point from another module

]]

local function _tag_from_name (args)
-- entry point from another module
        local msg;

        if args[1] and '' ~= args[1] then
                local data = mw.loadData ('Module:Lang/name to tag');
-- get the reversed data tables
                local lang = args[1]:lower();
-- allow any-case for the language name (speeling must till be correct)
                local tag = data.rev_lang_data[lang] or
data.rev_lang_name_table[lang];-- get the code; look first in the override
then in the standard

                if tag then
                        return tag;
-- language name found so return tag and done
                else
                        msg = 'language: ' .. args[1] .. ' not found'
-- language name not found, error message
                end
        else
                msg = 'missing language name'
-- language name not provided, error message
        end

        local template = '';
        if args.template and '' ~= args.template then
```

```
                template = table.concat ({'{{', args['template'], '}}: '});
-- make template name (if provided by the template)
        end
        return table.concat ({'<span style=\"font-size:100%; font-
style:normal;\" class=\"error\">error: ', template, msg, '</span>'});
end


--[[--------------------------< T A G _ F R O M _ N A M E >----------------
-----------------------------------

Module entry point from an {{#invoke:}}

]]

local function tag_from_name (frame)
-- entry point from an {{#invoke:Lang|tag_from_name|<language
name>|link=<yes>|template=<template name>}}
        return _tag_from_name (frame.args);
-- pass-on the args table, nothing else
end


--[[--------------------------< _ T R A N S L >----------------------------
-----------------------------------

Module entry point from another module

]]

local function _transl (args)
        local title_table = lang_data.translit_title_table;
-- table of transliteration standards and the language codes and scripts that
apply to those standards
        local language_name;
-- language name that matches language code; used for tool tip
        local translit;
-- translitterated text to display
        local script;
-- IANA script
        local msg;
-- for when called functions return an error message

        if is_set (args[3]) then
-- [3] set when {{transl|code|standard|text}}
                args.text = args[3];
-- get the transliterated text
                args.translit_std = args[2] and args[2]:lower();
-- get the standard; lower case for table indexing

                if not title_table[args.translit_std] then
```

```
                        return make_error_msg (table.concat ({'unrecognized
transliteration standard: ', args.translit_std}), args, 'transl');
                end
        else
                if is_set (args[2]) then
-- [2] set when {{transl|code|text}}
                        args.text = args[2];
-- get the transliterated text
                else
                        if args[1] and args[1]:match ('^%a%a%a?%a?$') then
-- args[2] missing; is args[1] a code or its it the transliterated text?
                                return make_error_msg ('no text', args,
'transl');                                      -- args[1] is a code so we're
missing text
                        else
                                args.text = args[1];
-- args[1] is not a code so we're missing that; assign args.text for error
message
                                return make_error_msg ('missing language /
script code', args, 'transl');
                        end
                end
        end

        if is_set (args[1]) then
-- IANA language code used for html lang= attribute; or ISO 15924 script code
                if args[1]:match ('^%a%a%a?%a?$') then
-- args[1] has correct form?
                        args.code = args[1]:lower();
-- use the language/script code; only (2, 3, or 4 alpha characters); lower
case because table indexes are lower case
                else
                        return make_error_msg (table.concat ({'unrecognized
language / script code: ', args[1]}), args, 'transl');        -- invalid
language / script code
                end
        else
                return make_error_msg ('missing language / script code',
args, 'transl');                        -- missing language / script code so
quit
        end

        args.italic, msg = validate_italic (args);
        if msg then
                return make_error_msg (msg, args, 'transl');
        end
        if 'italic' == args.italic then
-- 'italic' when |italic=yes; because that is same as absent or not set and
|italic=default
                args.italic = nil;
-- set to nil;
```

```
        end

        if lang_data.override[args.code] then
-- is code a language code defined in the override table?
                language_name = lang_data.override[args.code][1];
        elseif lang_name_table.lang[args.code] then
-- is code a language code defined in the standard language code tables?
                language_name = lang_name_table.lang[args.code][1];
        elseif lang_name_table.script[args.code] then
-- if here, code is not a language code; is it a script code?
                language_name = lang_name_table.script[args.code][1];
                script = args.code;
-- code was an ISO 15924 script so use that instead
                args.code = '';
-- unset because not a language code
        else
                return make_error_msg (table.concat ({'unrecognized language
/ script code: ', args.code}), args, 'transl');        -- invalid language /
script code
        end
-- here only when all parameters passed to make_translit() are valid
        return make_translit (args.code, language_name, args.text,
args.translit_std, script, args.italic);
end


--[[--------------------------< T R A N S L >------------------------------
----------------------------------

Module entry point from an {{#invoke:}}

]]

local function transl (frame)
        local args = getArgs(frame);
-- no {{#invoke:}} parameters
        return _transl (args);
end


--[[--------------------------< E X P O R T E D   F U N C T I O N S >--------
----------------------------------
]]

return {
        lang = lang,
-- entry point for {{lang}}
        lang_xx_inherit = lang_xx_inherit,
-- entry points for {{lang-??}}
        lang_xx_italic = lang_xx_italic,
        is_ietf_tag = is_ietf_tag,
```

```
        tag_from_name = tag_from_name,
-- returns ietf tag associated with language name
        name_from_tag = name_from_tag,
-- used for template documentation; possible use in ISO 639 name from code
templates
        transl = transl,
-- entry point for {{transl}}

        _lang = _lang,
-- entry points when this module is require()d into other modules
        _lang_xx_inherit = _lang_xx_inherit,
        _lang_xx_italic = _lang_xx_italic,
        _is_ietf_tag = _is_ietf_tag,
        _tag_from_name = _tag_from_name,
        _name_from_tag = _name_from_tag,
        _transl = _transl,
        };
```

actions taken to prevent or repair the deterioration of water management infrastructure and to keep the physical components of a water management system in such a state that they can serve their intended function.

Namespaces

- Module
- Discussion

Variants

Blue Gold Program Wiki

The wiki version of the Lessons Learnt Report of the Blue Gold program, documents the experiences of a technical assistance (TA) team working in a development project implemented by the Bangladesh Water Development Board (BWDB) and the Department of Agricultural Extension (DAE) over an eight+ year period from March 2013 to December 2021. The wiki lessons learnt report (LLR) is intended to complement the BWDB and DAE project completion reports (PCRs), with the aim of recording lessons learnt for use in the design and implementation of future interventions in the coastal zone.

- 

Blue Gold Program Wiki