rete - REcursive Template Engine

Tibor Pálinkás

igor2rete at inno dot bme dot hu

István Sándor

sandori at rht dot bme dot hu

rete - REcursive Template Engine

by Tibor Pálinkás and István Sándor

Table of Contents

1. User's manual 1
1.1. Introduction to template scripting 1
1.2. Template scripting example 1
1.3. Template scripting summary 2
1.4. Reference manual 2
1.4.1. String literals, escape sequences 3
1.4.2. Rete builtins 3
1.4.2.1. Variables 4
1.4.2.2. create 4
1.4.2.3. eval 5
1.4.2.4. expr 5
1.4.2.5. for 5
1.4.2.6. foreach 5
1.4.2.7. if 6
1.4.2.8. load 6
1.4.2.9. set 6
1.4.2.10. switch 6
1.4.3. Rete packages (plugins) 7
1.4.3.1. Package db_txt 7
1.4.3.2. Package geo 8
1.4.3.3. Package math 8
1.4.3.4. Package string 9
1.4.3.5. Package system 9
1.4.3.6. Package vars 10

List of Examples

1-1. index.tmpl 1
1-2. subpage.tmpl 2
1-3. Using string literals: template 3
1-4. Using string literals: output 3
1-5. if 6
1-6. switch 6
1-7. db_txt database file 7

Chapter 1. User’s manual

1.1. Introduction to template scripting

Rete is controlled by template files. On start, rete opens a single template file for processing. While processing a template file, normal text is copied verbatim while commands between %(and)% markers are interpreted as template scripts.

Template scripts can request another file to be generated. This normally involves processing another template file into another output file but appending to the current output and/or reprocessing the current template are both possible. Such recursive template processing should be regarded as a "child process" that may inherit variables from the caller. (Note: rete doesn’t actually fork or create threads, everything is done in a single process / single thread manner.) Since the template script language provides loops, arrays and branching, we often use the first template as a starting point or config file or run control file that can decide which output files should be generated from which templates. It’s also common that we generate a large number of files using a small number of templates.

Template scripting is divided into two parts: internal commands and external/extension commands. There are only a few internal commands, a small infrastructure on which external commands can be built. This infrastructure provides data storage, run control (loops, branches) and the above-mentioned recursive template processing. External commands are like function calls which have full access over the variables of the current running template processor.

1.2. Template scripting example

The following example has two templates, one of which generates an index.html that references the other 8 subpage html files, the other is the template for a single subpage.

Example 1-1. index.tmpl

```
<html>
<body>
 <H1> Rete template example </H1>
 <UL>
%(
 /* Change numberic format: print integers */
 format(0)

 for (n = 0; $n < 8; n = inc($n)) {
  /* Generate a link on the index page */
  "  <LI><a href=" $n ".html> subpage " $n "</a></LI>\n"

  /* Generate the subpage referenced from the index page */
  create("subpage.tmpl" -> $n ".html") {
   name = $n
  }
 }
)%
 </UL>
</body>
```

```
</html>
```

Example 1-2. subpage.tmpl

```
<html>
<body>
 <H2> This is subpage %($name)% </H2>
</body>
</html>
```

The above example is used as "rete index.tmpl > index.html"; all files will be generated in the current working directory. When the subprocess of generating a subpage is started, the global variables of the subprocess are set as defined in the parent template - in this case "name".

1.3. Template scripting summary

Core features:

- verbatim copy of the input to the output
- intepreting template scripts wrapped in %()%
- processing further template files (normally generating new output files)
- controls (if, switch, loops)
- variables (numbers, text, arrays)

External commands:

- read database from file to array
- math functions

Language summary: a template looks like the output expect for dynamic content wrapped in %()% markers. Dynamic parts can do calculation, can start recursive processing of further templates, can import data from files.

1.4. Reference manual

Rete template processing reads one input file and generates one output file and can run processing of subtemplates recursively. There are two modes of operation: copy and execute. When a template file is opened, copy mode is set. In copy mode the output is verbatim copy of the input. Template script blocks are wrapped in %(and)% markers; when rete reaches an script block open sequence (percent, bracket) it switches to execute mode and switches back to copy mode only when the closing sequence (brakcet, percent) is reached.

In execute mode the input is split into tokens and the tokens are evaluated. Tokens may be strings, which are copied to the output (after processing of some escape sequences, see below) or rete builtin instructions or external commands. Tokens are separated by whitespaces. As a common practice, each instruction starts in a new line and indenting similar to the one used in C should be applied.

This section is a reference manual for the execute mode and covers escape sequences, all builtin instructions and generally the features of the rete template scripting language.

1.4.1. String literals, escape sequences

Words which are neither builtin commands nor external function calls are threated as string literals and are copied to the output. Whitespaces are ignored.

It is possible to protect string literals with double quotes (""). In this case everything between the double quotes is threated as a single string literal, including white spaces.

Example 1-3. Using string literals: template

```
This is a %(
 te xt.    /* "te" and "xt." are string literals, the space between them will be ignored */
 "We can"  /* space is preserved */
 " "       /* a single space protected with dobuble quotes is a string literal */
 "protect spaces,"
)% and then we can go on with verbatim text.
```

Example 1-4. Using string literals: output

```
This is a text.We can protect spaces, and then we can go on with verbatim text.
```

Text outside of rete template scripting is copied without further processing. However, string literals inside the scripting blocks are processed and escape sequences are substituted. Currently the following escape sequences are in use:

- \t: tab character
- \n: newline character
- \q: double quote character

1.4.2. Rete builtins

Rete is a compact language with minimal amount of builtins. Task specific or acceleration or system dependent commands are implemented in dynamic loadable plugins as external commands (see later). Builtin commands are groupped in 4 groups:

- variables: set, expr (and operators)

- control: if, for, foreach, switch
- template generation: create
- misc: eval, load

All builtin commands are described below, in alphabetic order after the section about variables.

1.4.2.1. Variables

Variables are referenced by their names. A name must be a valid identifier (starting with a letter, containing any amount of letters, numbers and underscores). A dollar sign and the name of the variable is substituted with the value of the array, for example if the value of variable breakfast is string literal "ham and eggs", writing $breakfast in the script is the same as writing string literal "ham and eggs". Of course the value of a variable can be changed any time using command set.

Variable names may contain nonalphanumeric values or even white spaces. In this case the whole name must be protected with {} signs after the dollar sign. For example ${long name ^ with spaces} is a valid variable name.

Variables are either scalar or arrays; arrays are indexed using indices between [] braces. Using such braces after the dollar sign would break the rule of using alphanumeric characters only so referencing array elements implies using {} protection: ${my_array[42]}.

An array is either a linear array (when indexed by numbers) or a hash table (when indexed by string literals). Rete will automaticly decide about using a linear array or a hash table, the user doesn't need to worry about choosing one. However, once an array type is decided, as of the current version, rete will not convert the array to the other type. A rule of thumb is: *as long as an array is indexed only by integers, it will be a linear array*. This implies that if the first few indices are numbers, the array becomes a linear array and later string literal indices can not be added. This bug will be fixed later.

A special feature of linear arrays is auto indexing. When a new element is added with an empty index (using []), rete takes the last index of the array, increases it by one and uses this number as the new index. This makes it easy to grow an array without needing to store the last index or count the elements. This feature does not work with hash arrays.

Some rete commands like set, for or foreach will require variable names. A variable name is the name of the variable without the dollar ($) so variable substitution is not performed. Using a dollar sign there causes indirect variable usage. For example if the value of variable "name" is "breakfast", then usign $name where a variable name is expected will cause variable "breakfast" to be used as the variable name.

1.4.2.2. create

Create causes rete to suspend the interpreatation of the current template and start interpreting another template (called the child template). When interpreting the child template is finished, rete resumes interpreting the parent template. The parent template sets up the child template's global variables. This syntax of create is *create(template -> output) { variables }*

Argument template is the name of the template file and output is the name of the output file. If the output file is a single dash ("-"), the output of the child template is inserted in the output of the parent template; this feature can be considered as a form of "include".

In the variables section the parent template can set up the global variables of the child template. It is a comma separated list of name=value pairs pairs or single names. When name=value is used, the given name will be a global variable in the child template initialized with the corresponding value while listing a single name will be the same as

if name=$name was written (that is, the variable named "name" in the parent template is passed to the child template). Except for this last feature, the variable list of create is very similar to the argument of command *set*.

Note: the variable list may be empty; in this case an empty {} should be used.

1.4.2.3. eval

Eval takes a string literal argument, executes it as a rete template script and substitues the output. For example *eval("expr(1+2)")* will be substituted with 3.0000; of course in this case a plain expr(1+2) would result in the same output. Eval may be useful to call create and use the output of templates as string literals.

1.4.2.4. expr

Altough rete is not designed for doing calculations, it provides expr() which evaluates a mathematical expression and the result is substituted. Currently the following operations are supported:

- + is addition
- - is substraction
- * is multiplication
- / is division
- < evaluated to true if the first operand is less than the second
- <= evaluated to true if the first operand is less than or equal to the second
- > evaluated to true if the first operand is greater than the second
- >= evaluated to true if the first operand is greater than or equal to the second
- == evaluated to true if the two operands are equal
- != evaluated to true if the two operands are not equal
- ! is logical not
- && is logical and
- || is logical or

For example *expr(3*(1+2))* will be substituted by 9. Note: simply writing 1+1 in the template script part will not result in 2 without wrapping it in an expr().

1.4.2.5. for

In rete the syntax of for is very similar to the one in C (but less general): *for(initial-set;while;loop-set) { body }*.

Initial-set is a single name=value pair that initializes a variabe. The while part is evaluated as an expression (see expr()); the loop ends when the result of the expression is false. Loop-set is run at the end of each cycle and is similar to the initial-set argument (a single name=value pair).

1.4.2.6. foreach

Foreach can be used to consider each element of an array, either linear or hash. The syntax is *foreach index in array* where index is a variable name and array is the name of the array. Index will take each existing index value that presents in the array in random order.

1.4.2.7. if

Example 1-5. if

```
"1 is "
if (1 < 2) {
 "less"
} else {
 "greater"
}
" than 2. "
```

In rete the if statement is similar to the one on C. There's an expression in the () braces (the syntax is the same as in the argument of an expr() call). If the expression is evaluated to true, the "then" section is run, otherwise the "else" section is choosen. The else section may be omitted.

1.4.2.8. load

Load is used to load rete plugins which will provide external commands. The syntax is simple: *load(name)* where name is the name of the package (without leading rete/).

1.4.2.9. set

Set is used to create variables and change the value of variables. A single set command can perform these action on arbitrary amount of variables. The syntax of set is the following: *set { name1=value1, name2=value2, ..., nameN=valueN }* where each nameX is the name of a variable and the corresponding valueX is the new value of the variable.

1.4.2.10. switch

Example 1-6. switch

```
switch $n {
 case one   { "1" }
 case two   { "2" }
 case three { "3" }
 case four  { "4" }
 default { "none" }
}
```

The above example demonstrates the syntax of switch. Unlike in C, cases are not fall-trough. For a given input ($n in this example), exactly one of the cases or the default will run.

1.4.3. Rete packages (plugins)

With command *load()* the user may load rete plugins to access external commands. This section is a reference to the packages shipped with rete. Currently the following packages are available:

- db_txt - loads text databases into variables and arrays
- geo - geographic functions
- math - basic arithmetics
- string - string manipulation
- system - system calls
- vars - advanced variable manipulation

1.4.3.1. Package db_txt

This package can parse simple text database files and set rete global variables (including arrays) accordingly. The file format is simple:

Example 1-7. db_txt database file

```
# this is a comment
v1     :: one  :: comment1
v2     :: two  :: comment2

%%a1
 first

 third
 fourth
%%a2
 first
 second
 third
%% empty
%%

v3::three::comment3
```

Lines starting with a hashmark (#) are comments. The next two lines will create variable $v1 (with value "one") and $v2 (with value "two"). $COMMENT__v1 and $COMMENT__v2 are also created and loaded with strings "comment1" and "comment2" respectively. A line starting with %% loads an array whose name is given after the %%. Each line will become a new element of the integer indexed array after starting and trailing white spaces have been stripped off. A single "%%" line terminates array loading mode and returns to normal variable loading.

1.4.3.1.1. db_txt_use(file_name)

Call db_txt_use will load the text db file called *file_name* into the current global variable namespace.

1.4.3.2. Package geo

This package provides functions for geographic calculations.

1.4.3.2.1. coord_offset(base_point, distance_offset, direction)

Calculate coordinates of a geographic point given by a base point, a distance offset and a direction. The base point must be given as a string like this: "N12:34:56.78/E12:34:56.78", distance_offset is the distance in meters, direction is the direction in degrees, N=0, E=90, S=180, W=270

1.4.3.3. Package math

This package provides mathematical functions.

1.4.3.3.1. inc(value)

Increase *value* by one and return the result. Useful in cycles to avoid a long expr() which would be also slower.

1.4.3.3.2. dec(value)

Decrease *value* by one and return the result. Useful in cycles to avoid a long expr() which would be also slower.

1.4.3.3.3. alog(value)

Return the logarithm of *value*.

1.4.3.3.4. asqrt(value)

Return the square root of *value*.

1.4.3.3.5. apow(base, pwr)

Return *base* raised to the power of *pwr*.

1.4.3.3.6. format(num_digits)

Changes the standard numeric output format: any time a number is printed *num_digits* digits will be used after the decimal point.

1.4.3.4. Package string

This package provides functions for string manipulations.

1.4.3.4.1. strip(str)

Strip leading and trailing white spaces from*str* and return the resulting string.

1.4.3.4.2. field(str, fieldno, sep)

Extract a specific field of a *sep* separated string *str*. Argument *fieldno* selects the field, counting from 0. For example consider str="a|b|c", field(str, 1, "|") will return "b".

1.4.3.4.3. canon(str)

Strip leading and trailing white spaces and replace any other white space with "_" in *str* and return the result.

1.4.3.4.4. upper(str)

Converts *str* to all uppercase and return the result.

1.4.3.4.5. lower(str)

Converts *str* to all lowercase and returns the result.

1.4.3.4.6. strcomp(str1, str2)

Compares strings *str1* and *str2* and returns 0 if they are equal or their alphabetic order if not. (Note: it's a call to strcmp(3).)

1.4.3.4.7. split(str, arr, sep)

Splits *str* into fields using *sep* and creates/loads array named *arr* with the fields. The array will be integer indexed from 0.

1.4.3.5. Package system

This package provides system calls, mostly for manipulate the file system.

1.4.3.5.1. sys_mkdir(name [, perm])

Creates directory *name* with permissions *perm (optional). If permissions is given, it must be a number (in decimal, octal or hexadecimal format as documented in strtol(3)). NOTE: on windows permission is ignored.*

1.4.3.5.2. sys_access(name, mode)

Checks access of file *name* against *mode* and return "1" if it was OK. The following values are accepted for *mode*:

- R_OK - the file is readable
- W_OK - the file is writable
- X_OK - the file is executable
- F_OK - the file is a plain file

1.4.3.5.3. sys_system(cmd)

Call popen(3) to execute *cmd* in the background and returns the full output of the run.

1.4.3.6. Package vars

The `vars` package provides basic functions for querying and manipulating variables, e.g. checking the existence, emptyness of a variable, deleting a variable etc..

1.4.3.6.1. present(variable_name)

return 1 if a variable exists, otherwise return 0

```
%(
 load(vars)
 set {
  x = "1"
 }

 if (present(x)) {
  "there is a variable named x\n" /* this will show up in the output */
 }

 if (present(y)) {
  "there is a variable named y\n" /* this won't */
 } else {
  "variable named 'y' does not exist\n" /* this will be printed instead */
 }
```

```
)%
```

1.4.3.6.2. isempty(variable_name)

return 1 if a variable is nonexistent or an empty string, array or hash, otherwise return 0

```
%(
 load(vars)
 set {
  x = "",
  y = "hello"
 }

 if (isempty(x)) {
  "x is empty\n" /* this will show up in the output */
 }

 if (isempty(y)) {
  "y is empty\n" /* this won't */
 } else {
  "y is: " $y "\n" /* this will be printed instead */
 }
)%
```

1.4.3.6.3. length(variable_name)

return the length of a string or the number of elements in an array or hash

1.4.3.6.4. uniq(element, array)

insert `element` into `array` if it's not already present in array

```
%(
 load(vars)
 set {
  x[] = "i"
 }

 uniq("i", x) /* does not insert "i" */
 "x is: " $x "\n"
 uniq("j", x) /* insert "j" */
 "x is: " $x "\n"
)%
```

1.4.3.6.5. delete(variable_name)

delete a variable, after this call present(variable_name) will return false (0)