Formatted references

A set of Scheme functions for formatting references

in a TeXmacs document - v 0.2

Description

Are formatRef and formatList the names we want?

This program is inspired by the LaTeX package `prettyref`.

It provides the macro `formatRef` (`<formatRef|lab>`) which formats a reference choosing in a list of formats saved in the global variable `formatList`. Each format is a list consisting of two members; the first member is a format code, which is a short string, whicle the second member is the corresponding format to apply to the reference. The function selects the format by comparing the format code with the beginning of the label: the first element of `formatList` that matches is selected and the corresponding format string `formatString` is prepended to the TeXmacs reference, so that the reference obtained with the macro `<formatRef|lab>` looks like

`formatString<reference|lab>`

The labels must be assigned according to the format codes so that the macros have an effect. If no format code matches, the macro returns

`<reference|lab>`

TeXmacs macros and their use

I have defined the TeXmacs macros in the preamble of this document

formatRef

`<formatRef|lab_string>`

Formats a reference so that it looks like

`formatString <reference|lab>`

where `formatString` is the format string of the first matching format (empty string if there is no match).

addToFormatRef

`<addToFormatRef|formatCode_string|formatString_string>`

Adds to the list of formats `formatList` (which is a global Scheme variable) the new format which is composed of the strings `formatCode` and `formatString`.

Example:

`<addToFormatRef|fig:|Figure >` makes the new format `("fig:" "Figure")` available, so that the application of the macro

`<formatRef|fig:figure1>`

returns

Figure <reference|fig:figure1>

deleteFromFormatRef

<deleteFromFormatRef|formatCode_string|n_integer>

Deletes from formatList the n-th format that matches formatCode. If no format that matches format code remains after deletion, then

<formatRef|lab>

will return

<reference|lab>

Example:

<deleteFromFormatRef|fig:|1>

deletes from formatList the first occurrence of the format with format code "fig:"

replaceInFormatRef

<replaceInFormatRef|formatCode_string|formatString_string|n_integer>

Replaces in formatList the n-th format with matching formatCode with the format ("formatCode" "formatString").

Example:

Let the initial value of formatList be (list (list "eq:" "eqn. ") (list "Sec:" "Section ") (list "eq:" "equation "))

<deleteFromFormatRef|eq:|Equation >

transforms formatList into

(list (list "eq:" "eqn. ") (list "Sec:" "Section ") (list "eq:" "Equation "))

in this case the application of the macro

<formatRef|eq:equation1>

returns

eqn. <reference|eq:equation1>

like it did originally, as the matching format is the first one that matches the format code.

Scheme functions and their use

formatList

Global variable, accessible from the whole document.

A list of formats. Each element of a list is a list of two elements (both strings). The first element is a "format code" and is examined by the program to match label and format; the second element is the string which is applied to format the reference.

extractFormatString

(extractFormatString label_string formatList_list)

Extracts from the list of formats formatList the matching format; returns the second member of the list that matches the format (that is, the format string).

A format matches if its first element (the "format code") is equal to the beginning of the `label` string.

Example:

```
(extractFormatString "fig:" (list (list "fig:" "Figure ") (list "Sec:" "Section
")))
```

returns `"Figure "`

addToFormatList

```
(addToFormatList format_list)
```

Adds to the list of formats `formatList` (which is a global variable) the new format `format`.

Checks that `format` is syntactically correct and that does not already exists in `formatList`.

Example:

Let the initial value of `formatList` be `(list (list "fig:" "Figure ") (list "Sec:" "Section "))`

```
(addToFormatList (list "eq:" "Equation "))
```

transforms `formatList` into

```
(list (list "fig:" "Figure ") (list "Sec:" "Section ") (list "eq:" "Equation "))
```

deleteFromFormatList

```
(deleteFromFormatList formatToDelete_list n_integer)
```

I would like to rewrite the function so that is accepts either lists or strings

Deletes from `formatList` the n-th format that matches `formatToDelete`.

`formatToDelete` must be a list; it must not a syntactically correct format, as only its first element will be matched to formats.

Example:

Let the initial value of `formatList` be `(list (list "eq:" "eqn.  ") (list "Sec:" "Section ") (list "eq:" "equation "))`

```
(deleteFromFormatList (list "eq:") 2)
```

transforms `formatList` into

```
(list (list "eq:" "eqn.  ") (list "Sec:" "Section "))
```

replaceFormat

```
(replaceFormat newFormat_list n_integer)
```

Replaces in `formatList` the n-th format that matches `newFormat` with `newFormat` itself.

`newFormat` must be a syntactically correct format. The match is done checking the first element of the list `newFormat` (like the match done by `extractFormatString`).

Example:

Let the initial value of `formatList` be `(list (list "eq:" "eqn.  ") (list "Sec:" "Section ") (list "eq:" "equation "))`

```
(replaceFormat (list "eq:" "Equation ") 2)
```

transforms `formatList` into

```
(list (list "eq:" "eqn.  ") (list "Sec:" "Section ") (list "eq:" "Equation "))
```

Functions - code and detailed description

Global variables

`formatList` is the list of all formats (for the moment empty)

```
Scheme] (define formatList (list))
Scheme]
```

Comparison to format code

`ref` is a bad name here as the string is is already called "reference"

We compare the reference string `str` to a format code `ref`. To do this, we extract first from the string the intial substring of length equal to the length of `ref`; we then compare the extracted substring to the format code.

Check that `str` is at least as long as `ref` before extracting the initial substring from `str`.

```
Scheme] (define (checkLength str ref)
          (<= (string-length ref) (string-length str)))
Scheme] (define (extractBase str ref)
            (substring str 0 (string-length ref)))
```

If `str` is at least as long as `ref`, extract the substring; if not, return the empty string.
If the function returns the empty string, the format will not match, as we define it not to match in the matching function.

```
Scheme] (define (extract str ref)
            (if (checkLength str ref)
                (extractBase str ref)
                 ""))
Scheme]
```

Comparison between string and format code

The matching function will return false if either `str` or `ref` is the empty string (i.e., no format will be applied to an empty reference and no format with empty format code will be applied to a reference)

```
Scheme] (define (checkLabelInputs str ref)
            (and
             (not (equal? str ""))
             (not (equal? ref ""))))
```

Use the `extract` function to compare `str` and `ref`.

```
Scheme] (define (compareSubstrBase str ref)
            (equal? ref (extract str ref)))
```

Check first for empty `ref` or `str`, then compare.

```
Scheme] (define (compareSubstr str ref)
            (if (checkLabelInputs str ref)
                (compareSubstrBase str ref)
                 #f))
```

Compare the string to the first element of a format

The format is a list of two elements, the format code and the format itself
We want to match only on correctly formed formats.
Check that `lst` is a list of length 2.

The check on `lst` needs to be complete

We probably need a comprehensive check that the format is correctly formed, that is `lst` is a list of length 2 and both elements are strings.

Note 1. I already wrote the complete check in the functions the compose the list; I need to apply it consistently

```
Scheme] (define (listCond lst)
           (if (not (list? lst))
               #f
               (= (length lst) 2)))
```

If `ft` is correctly formed, compare `str` to the first element of `ft`; otherwise return `#f`

```
Scheme] (define (compareSubstrFt str ft)
          (if (listCond ft)
              (compareSubstr str (list-ref ft 0))
              #f))
Scheme]
```

Find a good way of referring to the "format string" part of the format

Extraction of format string

We extract from the format list the first matching format and from the format its format proper.

Filter format list `labelFormats` according to the `compareSubstrFt` predicate. In a subsequent function, we will select the first element of the filtered list; if the filtered list is empty we will return an empty list.

```
Scheme] (define (filterFormats str labelFormats)
          (filter
           (lambda (x) (compareSubstrFt str x))
              labelFormats))
```

Check again `car` on an empty list

We select here the first element of the filtered list; if the filtered list is empty we return an empty list (we need to do that in a separate case, because `car` does not work on an empty list)

```
Scheme] (define (extractFormat str labelFormats)
           (let ((matchList (filterFormats str labelFormats)))
             (if (null? matchList)
                 (list)
                 (car matchList))))
```

From the format, extract the format proper, which is the element in second position; return an empty string if the format is an empty list (i.e. we did not find any match for the reference string)

```
Scheme] (define (extractFormatString str labelFormats)
           (let ((format (extractFormat str labelFormats)))
             (if (not (= (length format) 2))
                         ""
                         (car (cdr format)))))
Scheme]
```

Manipulation of format list

Is `checkGroup` a good name for this function? Same question for the other function names in this section

Check that a format list is syntactically correct

Checks that a single format `formatGroup` is a list of two elements (to do: must issue error).

```
Scheme] (define (checkGroup formatGroup)
          (cond
             ((not (list? formatGroup)) #f)
              ((not (= (length formatGroup) 2)) #f)
              (else #t)))
```

Checks that all elements of `formatGroup` are strings (to do: must issue error).

```
Scheme] (define (checkFormatStrings formatGroup)
           (apply and-list (map string? formatGroup)))
```
Checks that the format is syntactically correct: it must be a list of two elements, each of which is a string.
```
Scheme] (define (checkFormatGroup formatGroup)
          (and (checkGroup formatGroup)
               (checkFormatStrings formatGroup)))
Scheme]
```

Check whether a format is present in the list

Compare two formats (different from the "matching" function because it compares the lists that define formats)
```
Scheme] (define (checkFormatEqual formatGroup1 formatGroup2)
           (equal? (car formatGroup1) (car formatGroup2)))
```
Check that a format is present in the global variable `formatList` using `checkFormatEqual` and the `find` function of Scheme.
```
Scheme] (define (checkFormatPresent formatGroup)
          (not
           (equal?
            (filter (lambda (x) (checkFormatEqual formatGroup x)) formatList)
            (list))))
Scheme]
```

Adding a format to the list

Check on input:

- The format is a list of two elements
- The format list consists of strings
- The format is not already present

Note 2. The first two checks should be combined into one, the test of the correct syntax of the format

```
Scheme] (define (checkFormatGroupAdd formatGroup)
           (cond
              ((not (checkGroup formatGroup)) (begin (display "Format must be a
        list of two elements") #f))
               ((not (checkFormatStrings formatGroup)) (begin (display "Format
        must be a list of two strings") #f))
                ((checkFormatPresent formatGroup) (begin (display "Format is
        already present, use replaceFormatGroup to replace") #f))
                 (else #t)))
Scheme] (define (addToFormatList formatGroup)
          (if (checkFormatGroupAdd formatGroup)
              (set! formatList (append formatList (list formatGroup)))
               (display "\nincorrect application of addToFormatList")))
Scheme]
```

I could call the input parameter `cd` with the name `pred`

Deleting a format from the list

Delete the n-th element of a list for which the predicate `cd` (for condition) is true.

In part copied from a Stackexchange question.

```
Scheme] (define (delete-nth-cond cd lst n)
           (cond ((null? lst) '()) ; base case
                 ((cd (car lst)) (cond ((= n 1) (cdr lst)); if n = 1 return the
        cdr, otherwise we call it again with n-1
                                       (else (cons (car lst)
                                                   (delete-nth-cond cd (cdr lst)
        (+ -1 n))))))
                 (else (cons (car lst) ; if the condition is not satisfied retain
        the last element
                                    ; and apply the function to the rest of the
        list
                             (delete-nth-cond cd (cdr lst) n)))))
```

Delete the n-th occurrence of a format from the list

```
Scheme] (define (deleteFromFormatList formatGroup n)
          (set! formatList
                (delete-nth-cond
                  (lambda (x) (checkFormatEqual formatGroup x)) formatList n)))
Scheme]
```

Here the input parameter for the condition is called `test`; perhaps it is a good name for `delete-nth-cond` too

Replace the n-th element of a list for which the predicate `test` (for condition) is true.

Replace the n-th element of a list that satisfied test

This is copied from `replace-nth-Test2` in the Emacs file, I have renamed it here

```
Scheme] (define (replace-nth-ElementCond lst test to n)
           (cond ((null? lst) '()) ; base case = end of input - issue warning
        here?
                 ((test (car lst)) (if (= n 1) (cons to (cdr lst))
                                       (cons (car lst)
                                             (replace-nth-ElementCond (cdr lst)
        test to (+ n -1)))))
                          (else (cons (car lst)
                                      (replace-nth-ElementCond (cdr lst) test to
        n)))))
```

Replace the n-th occurrence of a format in the list

Uses ideas from a Stackexchange question. Checks that there are at least n elements of the format list that match `formatGroup`.

```
Scheme] (define (replaceFormatCore formatGroup n)
          (let ((condition (lambda (x) (checkFormatEqual formatGroup x))))
             (let ((lengthFts  (length (filter condition formatList))))
               (if (>= (length (filter condition formatList)) n)
                    (set! formatList (replace-nth-ElementCond formatList condition
        formatGroup n))
                     (begin (display "Format list contains only ")
                            (display lengthFts)
                              (display " formats corresponding to the
        input"))))))
```

Checks that `formatGroup` is sintactically correct, then replaces the n-th format whose format code matches.

```
Scheme] (define (replaceFormat formatGroup n)
          (if (checkFormatGroup formatGroup)
               (replaceFormatCore formatGroup n)
                (display "did not execute replacement function")))
```

```
Scheme]
```

Helper functions

An `and` function that can be applied to lists (using the Scheme function `apply`). It returns `#t` if all elements of the list are `#t`. Copied from a Stackexchange question.

```
Scheme] (define and-list (lambda x
                           (if (null? x)
                               #t
                               (if (car x) (apply and-list (cdr x)) #f))))
```

I do not understand how the parameters of `and-list` are determined. Is there a general rule that applied to this case and I do not know?

Note 3. `and-list` is lambda variadic, so it can be applied to either an arbitrary number of arguments - for example `(and-l #t #t #f)` - or to a list using `apply` - example `(apply and-list (list #t #t #f))`matches"?

```
Scheme]
```

Interface to TEXMACS

```
Scheme] (set! formatList (list (list "eq:" "Equation ") (list "fig:" "Figure ")))
Scheme] (tm-define (formatRefScheme str)
          (set! str (tree->stree str))
          (stree->tree (extractFormatString str formatList)))

   ((guile-user) (guile-user))

Scheme] (extractFormatString "eq:" formatList)

   "Equation "

Scheme]
```

Test

$$\sin(x)^2 + \cos(x)^2 = 1 \tag{1}$$

Equation 1

Manipulation of format list

Add to format list

```
Scheme] (tm-define (addToFormatListScheme formatCode formatString)
          (set! formatCode (tree->stree formatCode))
          (set! formatString (tree->stree formatString))
          (let ((ft (cons formatCode (cons formatString (list) ))
                    ))
                    (addToFormatList ft)))

   ((guile-user) (guile-user))

Scheme]
```

?

Note 4. I need to execute the macro addToFormatRefList without returning anything

```
Scheme] formatList
   (("eq:" "Equation ") ("fig:" "Figure ") ("sec:" "Section "))
Scheme]
```

Test

sect. 6.1

Delete from format list

```
Scheme] (tm-define (deleteFromFormatListScheme formatCode n)
          (set! formatCode (tree->stree formatCode))
          (set! n (string->number (tree->stree n)))
                (let (
        (ft (cons formatCode (list)))
        )
                  (deleteFromFormatList ft n)))
   ((guile-user) (guile-user))
Scheme]
```

Test

?

```
Scheme] formatList
   (("eq:" "Equation ") ("fig:" "Figure "))
Scheme]
```

6.1

Replace in format list

```
Scheme] (tm-define (replaceInFormatListScheme formatCode formatString n)
          (set! formatCode (tree->stree formatCode))
          (set! formatString (tree->stree formatString))
          (set! n (string->number (tree->stree n)))
          (let (
        (ft (cons formatCode (cons formatString (list))))
         )
            (replaceFormat ft n)))
   ((guile-user) (guile-user))
Scheme]
```

Test

?

?

```
Scheme] formatList
   (("eq:" "Equation ") ("fig:" "Figure ") ("sec:" "sect. "))
Scheme]
```

sect. 6.1