The coollist package*

nsetzer

October 11, 2007

The coollist package is a "sub" package of the cool package that seemed appropriate to publish independently since it may occur that one wishes to include the ability to manipulate lists without having to accept all the overhead of the cool package itself.

1 Basics

Lists are defined as a sequence of tokens separated by a comma. The coollist package allows the user to access certain elements of the list while neglecting others—essentially turning lists into a sort of array.

List elements are accessed by specifying the position of the object within the list (the index of the item) and all lists start indexing at `1`.

2 Commands & Descriptions

`\setlistStop` `\setlistEnd`

`\setlistStop{`⟨*string*⟩`}` and `\setlistEnd{`⟨*string*⟩`}` allow the user to set the end of a list 'character' in the rare event that the default values actually appear in the list. `listStop` is used to identify when the list actually terminates, while `listEnd` forces the reading macro to take in the entire list (without both entities, errors would occur if macros were included in the list).

The default values are

```
listStop   @@
listEnd    @@@
```

`\listval`

`\listval{`⟨*list*⟩`}{`⟨*index*⟩`}` returns the ⟨*index*⟩ item of the comma delimited list ⟨*list*⟩ or nothing if ⟨*index*⟩ is outside the number of elements of the list. The first element of the list has index 1.

`\liststore`

`\liststore{`⟨*list*⟩`}{`⟨*macro_base_name*⟩`}` stores the elements of comma delimited list ⟨*list*⟩ in a set of macros having the first part of ⟨*macro_base_name*⟩ and ending with the roman numermal index of the list.

For example, `\liststore{1,2,3}{list}` would define `\listi`, `\listii`, and `\listiii` each holding `1`, `2`, `3`, respectively.

`\listlen`

`\listlen{`⟨*list*⟩`}` returns the length of the comma delimited list ⟨*list*⟩, though

*This document corresponds to coollist v1.2, dated 2007/10/06.

it is not useful for storing this length. If you need to record the list's length for later use, it is better to use the function `\listlenstore`.

`\listlenstore` `\listlenstore{`⟨*counter*⟩`}{`⟨*list*⟩`}` stores the length of the comma delimited list ⟨*list*⟩ is the counter ⟨*counter*⟩.

`\listsum` `\listsum{`⟨*list*⟩`}{`⟨*macro*⟩`}` stores the sum of the comma delimited list ⟨*list*⟩ in the macro ⟨*macro*⟩. Integers are recognized and summed accordingly. All other tokens are summed as variables with some integer coefficient as the end result.

3 Test Cases

3.1 \listval

`\listval`

```
\listval{1,2,3,4}{0}                    (the null string)
$\listval{\alpha,\beta,\gamma}{2}$      β
\listval{a,b,c}{4}                      (the null string)
```

3.2 \liststore

`\liststore`

```
\liststore{1,2,3,4}{temp}
\tempi;\tempii;\tempiii;\tempiv         1;2;3;4

\liststore{a_1,a_2,a_3,a_4}{temp}
$\tempi;\tempii;\tempiii;\tempiv$       a_1; a_2; a_3; a_4

\liststore{a,b}{temp}
\tempi;\tempii                          a;b
```

3.3 \listlen

`\listlen`

```
\listlen{1,2,3,4,5}   5
\listlen{}            0
\listlen{1,2}         2
\listlen{1}           1
```

3.4 \listlenstore

`\listlen`

```
\newcounter{thelistlength}
```

```
\listlenstore{thelistlength}{1,2,3,4,5}
\arabic{thelistlength}
```
5

```
\listlenstore{thelistlength}{}
\arabic{thelistlength}
```
0

3.5 \listsum

`\listsum` Summing elements of lists

```
\listsum{1,2,3,4,5}{\thelistsum}
\thelistsum
```
15

```
\listsum{1,2,3,a,b,a,a}{\thelistsum}
\thelistsum
```
6+3a+b

```
\listsum{a,b,c,d}{\thelistsum}
\thelistsum
```
a+b+c+d

4 Implementation

This is just an internal counter for dealing with the lists, most often used for the length of the list.

```
\newcounter{COOL@listlen}%
```

\setlistEnd
\setlistStop

`\setlistStop{`⟨*string*⟩`}` and `\setlistEnd{`⟨*string*⟩`}` allow the user to set the end of a list 'character's in the rare event that the default values actually appear in the list. Both of these entities are required to properly deliminate the list and avoid errors when macros are included in the list. The default values are

```
\newcommand{\COOL@listEnd}{@@@}%
\newcommand{\COOL@listStop}{@@}%
```

and they may be changed by the following commands (which utilize the `\renewcommand`):

```
\newcommand{\setlistStop}[1]{\renewcommand{\COOL@listStop}{#1}}%
\newcommand{\setlistEnd}[1]{\renewcommand{\COOL@listEnd}{#1}}%
```


This area defines the core technology behind the coollist package: the list "gobbler". To properly eat a list a counter and a boolean need to be used. `listpointer` acts just like the name implies, as the current "position" of the list. `found` indicates that the position has been found

```
\newcounter{COOL@listpointer}%
\newboolean{COOL@found}%
```

Now we come to "the gobbler"—a recursive function that eats up a list and gives back the appropriate item. This must be done in TEX primatives.

The idea behind this is that "the gobbler" eats up everything before the desired item and everything after the desired item.

```
\def\COOL@listgobble[#1]#2,#3,\COOL@listEnd{%
\ifthenelse{\equal{#3}{\COOL@listStop}}%
    {%
```

we have reached the end of the list, just need to check if we need to output something

```
    \ifthenelse{\value{COOL@listpointer}=#1}%
        {%
```

```
        \setboolean{COOL@found}{true}%
        #2%
        }%
    % Else
        {%
        }%
    }%
% Else
    {%
    \ifthenelse{\value{COOL@listpointer}=#1}%
        {%
        \setboolean{COOL@found}{true}%
        #2%
        }%
    % Else
        {%
        }%
    \stepcounter{COOL@listpointer}%
```

We must eat up the whole list no matter what or else the stuff beyond `#1` will be displayed. so we need to call "the gobbler" again.

```
    \COOL@listgobble[#1]#3,\COOL@listEnd%
    }%
}%
```

`\listval` `\listval{`⟨*comma_deliminated_list*⟩`}{`⟨*index*⟩`}`
gives the ⟨*index*⟩ value of ⟨*comma_deliminated_list*⟩—as in
`\listval{1,2,3,4,5,6}{3}` = \listval{1,2,3,4,5,6}{3}
`$\listval{\alpha,\beta,\gamma}{2}$` = $\listval{\alpha,\beta,\gamma}{2}$

```
\newcommand{\listval}[2]{%
```

check to see if the submitted list is empty. if it is, do nothing

```
\ifthenelse{\equal{#1}{}}%
    {%
```

set the listpointer to zero because the list has no length

```
    \setcounter{COOL@listpointer}{0}%
    }%
```

Else

```
    {%
```

start at the beginning of the list, so initialize `listpointer`

```
    \setcounter{COOL@listpointer}{1}%
```

Assume that the target will not be found—it will be set to true by "the gobbler" if it is

```
    \setboolean{COOL@found}{false}%
```


Now call the gobbler—since the user shouldn't be forced to submit the end character (in fact he or she shouldn't even need to worry that an end character exists nor what it is), we add it on along with the 'optional' parameter that tells us which element to retreive. To ensure that the entire list is read in by `\COOL@listgobbler` we need the list stop 'character' too.

```
    \COOL@listgobble[#2]#1,\COOL@listStop,\COOL@listEnd%
    }%
}%
```

`\liststore` `\COOL@liststore@gobbler`

The list may be stored in a macro of the user's choosing with the function. The syntax is

`\liststore{`⟨*csv_list*⟩`}{`⟨*macro_base_name*⟩`}`

and the resulting list elements are stored in

⟨*macro_base_name*⟩⟨*list_index_roman*⟩

where ⟨*list_index_roman*⟩ is the list index in roman numerals.

Some examples will clarify:

`\liststore{1,2,3,4}{temp}`

`\tempi;\tempii;\tempiii;\tempiv` yields 1;2;3;4

`\liststore{a_1,a_2,a_3,a_4}{temp}`

`\tempi;\tempii;\tempiii;\tempiv` yields $a_1; a_2; a_3; a_4$

```
\def\COOL@liststore@gobbler[#1]#2,#3,\COOL@listEnd{%
\ifthenelse{\equal{#3}{\COOL@listStop}}%
    {%
    \expandafter\gdef\csname #1\roman{COOL@listpointer}\endcsname{#2}%
    }%
% Else
    {%
    \expandafter\gdef\csname #1\roman{COOL@listpointer}\endcsname{#2}%
    \stepcounter{COOL@listpointer}%
    \COOL@liststore@gobbler[#1]#3,\COOL@listEnd%
    }%
}
\newcommand{\liststore}[2]{%
\setcounter{COOL@listpointer}{1}%
\COOL@liststore@gobbler[#2]#1,\COOL@listStop,\COOL@listEnd%
}%
```

`\listlen` This returns the length of the list, though it is not useful for storing this length. If you need to record the list's length for later use, it is better to use the next function `\listlenstore`.

The format is `\listlen{`⟨*comma deliminated list*⟩`}`. It works by recording the value of `listpointer` after it has complete traversed the list. Since indexing starts at `1`, it uses the index `0` which will never ever be an index of the list, so "the gobbler" will not return any value.

Example: `\listlen{1,2,3,4,5}` = 5

```
\newcommand{\listlen}[1]{%
\listval{#1}{0}%
\arabic{COOL@listpointer}
}%listlength
```

`\listlenstore` This store the length of the list. The format is `\listlenstore{`⟨*counter*⟩`}{`⟨*comma deliminated list*⟩`}`.

```
\newcommand{\listlenstore}[2]{%
```

```
\listval{#2}{0}%
\setcounter{#1}{\value{COOL@listpointer}}
}%listlength
```

`\listsum` Sum the contents of the list. Integers are recognized and summed, tokens are treated as independent variables. The function returns a string of the sum

Counter for the coefficients

```
\newcounter{COOL@intsum}
```

Counter for the register index

```
\newcounter{COOL@register@ct}
\newcounter{COOL@register@len}
```

boolean for identifying integers

```
\newboolean{COOL@listsum@isint}
```

Now the function

∞

```
\newcommand{\listsum}[2]{%
```

First store the entire list

```
\liststore{#1}{COOL@listtosum@element@}%
```

store the length of the list

```
\listlenstore{COOL@listlen}{#1}%
```

check for the list having a non-zero length

```
\ifthenelse{ \value{COOL@listlen} < 1 }%
    {%
    \PackageWarning{cool}{List is empty}%
    \xdef#2{0}%
    }%
```

Else

```
    {%
```

put the first list element into the register

```
    \isint{\COOL@listtosum@element@i}{COOL@listsum@isint}%
    \ifthenelse{ \boolean{COOL@listsum@isint} }%
        {%
        \xdef\COOL@listsum@register@integers{\COOL@listtosum@element@i}%
        \setcounter{COOL@register@len}{0}%
        }%
```

Else

```
        {%
```

Initialize the integers register to zero; store the character and its coefficient

```
        \gdef\COOL@listsum@register@integers{0}%
        \xdef\COOL@listsum@register@i{\COOL@listtosum@element@i}%
        \gdef\COOL@listsum@register@coef@i{1}%
        \setcounter{COOL@register@len}{1}%
        }%
```

Now go through each additional element making an index of the symbols and summing identical ones

```
    \forloop{COOL@listpointer}{2}{\NOT \value{COOL@listpointer} > \value{COOL@listlen}}%
        {%
```

Expand the element to a convenient storage macro

```
        \xdef\COOL@listsum@element{\csname COOL@listtosum@element@\roman{COOL@listpointer}\endcsname}%
```

Check if this element is an integer

```
        \isint{\COOL@listsum@element}{COOL@listsum@isint}%
        \ifthenelse{ \boolean{COOL@listsum@isint} }%
            {%
```

Grab the current value of the integers and store it in the register counter

```
            \setcounter{COOL@intsum}{\COOL@listsum@register@integers}%
            \addtocounter{COOL@intsum}{\COOL@listsum@element}%
            \xdef\COOL@listsum@register@integers{\arabic{COOL@intsum}}%
            }%
```

Else, it's not an integer so search to see if it matches known elements

```
        {%
        \setboolean{COOL@found}{false}%
        \forloop{COOL@register@ct}{1}{\NOT \value{COOL@register@ct} > \value{COOL@register@len}}%
            {%
            \xdef\COOL@listsum@known@element{%
                \csname COOL@listsum@register@\roman{COOL@register@ct}\endcsname%
                }%
            \ifthenelse{ \equal{\COOL@listsum@element}{\COOL@listsum@known@element} }%
                {%
```

found the element so increment the coefficient (grab coefficient, store in ct, increment ct, store new ct)

```
                \xdef\COOL@listsum@known@element@coef{%
                    \csname COOL@listsum@register@coef@\roman{COOL@register@ct}\endcsname%
                    }%
                \setcounter{COOL@intsum}{\COOL@listsum@known@element@coef}%
                \addtocounter{COOL@intsum}{1}%
                \expandafter\xdef\csname COOL@listsum@register@coef@\roman{COOL@register@ct}\endcsname{\arabic{COOL@intsum}}%
```

flag the element as found and set the counter to the length of the register +1

```
                \setboolean{COOL@found}{true}%
                }%
```

Else do nothing

```
                {%
                }%
            }%
```

Check to see if the element is a known element. If not, add it to the register

```
        \ifthenelse{ \boolean{COOL@found} }%
            {}%
```

Else

```
            {%
```

```
                \addtocounter{COOL@register@len}{1}%
                \expandafter\xdef\csname COOL@listsum@register@\roman{COOL@register@len}\endcsname{\COOL@listsum@element}%
                \expandafter\xdef\csname COOL@listsum@register@coef@\roman{COOL@register@len}\endcsname{1}%
                }%
            }%
        }%
```

Finally, create and store the sum

```
    \xdef\COOL@listsum@result{}%
    \ifthenelse{ \NOT \COOL@listsum@register@integers = 0 }%
        {
        \xdef\COOL@listsum@result{\COOL@listsum@result\COOL@listsum@register@integers}%
        \ifthenelse{ \NOT \value{COOL@register@len} = 0 }%
            {%
            \xdef\COOL@listsum@result{\COOL@listsum@result+}%
            }{}%
        }{}%
    \forloop{COOL@register@ct}{1}{ \NOT \value{COOL@register@ct} > \value{COOL@register@len} }%
        {%
        \edef\COOL@listsum@curcoef{\csname COOL@listsum@register@coef@\roman{COOL@register@ct}\endcsname}%
        \ifthenelse{ \NOT \COOL@listsum@curcoef = 1}%
            {%
            \xdef\COOL@listsum@result{\COOL@listsum@result\COOL@listsum@curcoef}%
            }{}%
        \xdef\COOL@listsum@result{\COOL@listsum@result\csname COOL@listsum@register@\roman{COOL@register@ct}\endcsname}%
        \ifthenelse{ \NOT \value{COOL@register@ct} = \value{COOL@register@len} }%
            {%
            \xdef\COOL@listsum@result{\COOL@listsum@result+}%
            }{}%
        }%
    \xdef#2{\COOL@listsum@result}%
    }%
```

```
157 }
```

Change History

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.