The \texttt{listproc} package

Jesse A. Tov
tov@ccs.neu.edu

This document corresponds to \texttt{listproc} v0.2, dated 2011/08/03.

Contents

1 Introduction 1

2 Command Reference 2
  2.1 List Definition ............................................. 2
  2.2 Basic Imperative List Commands .............................. 2
  2.3 Control and Higher-Order List Commands ..................... 3
  2.4 Nice List Printing ........................................... 5
  2.5 Functional List Expressions ................................ 5

3 Implementation 6
  3.1 List Definition ............................................. 6
  3.2 Basic Imperative List Commands .............................. 7
  3.3 Control and Higher-Order List Commands ..................... 9
    3.3.1 Sorting .................................................. 10
    3.3.2 Compressing ............................................. 12
  3.4 Nice List Printing ........................................... 14
  3.5 Function List Expressions ................................ 15

1 Introduction

The purpose of this package is to support programming with lists, including higher-order programming with \texttt{map} and \texttt{sort}.

We use a representation of lists suggested in \textit{The \TeX\book}. A list of elements \((\toks)_1, (\toks)_2, \ldots (\toks)_k\) is represented as a macro that expands to

\begin{verbatim}
\listitem{(\toks)_1}\listitem{(\toks)_2}\ldots\listitem{(\toks)_k}
\end{verbatim}

It is easy to iterate over such a list by defining \texttt{\listitem} to do whatever should happen to each item in the list and then evaluating the list. (The empty list is represented by the empty macro, which may be written \texttt{\empty}.)
2 Command Reference

2.1 List Definition

\newlist{\langle dst-list\rangle}{\langle csv\rangle}
\renewlist{\langle dst-list\rangle}{\langle csv\rangle}
\deflist{\langle dst-list\rangle}{\langle csv\rangle}
\gdeflist{\langle dst-list\rangle}{\langle csv\rangle}

Define \langle dst-list\rangle to be a list whose contents are \langle csv\rangle, where \langle csv\rangle is a comma-separated list. This is mostly useful for handling input from user macros. These differ on how they handle (re)definition: \newlist requires that \langle dst-list\rangle not be defined, whereas \renewlist requires that it be defined already. Neither \deflist or \gdeflist cares whether \langle dst-list\rangle is defined, as both will overwrite it if necessary; \gdeflist does the definition globally.

2.2 Basic Imperative List Commands

Most commands work by destructively updating a list. For a functional interface, see §2.5.

\ConsTo{\langle item\rangle}{\langle list-macro\rangle}
\gConsTo{\langle item\rangle}{\langle list-macro\rangle}
\eConsTo{\langle item\rangle}{\langle list-macro\rangle}
\xConsTo{\langle item\rangle}{\langle list-macro\rangle}

Add \langle item\rangle to the beginning of the list in \langle list-macro\rangle, destructively updating \langle list-macro\rangle. \ConsTo and \eConsTo make the change locally to the current group, whereas \gConsTo and \xConsTo do a global update. \ConsTo and \gConsTo do not expand \langle item\rangle before adding it to the list, whereas \eConsTo and \xConsTo do expand \langle item\rangle.

\SnocTo{\langle item\rangle}{\langle list-macro\rangle}
\gSnocTo{\langle item\rangle}{\langle list-macro\rangle}
\eSnocTo{\langle item\rangle}{\langle list-macro\rangle}
\xSnocTo{\langle item\rangle}{\langle list-macro\rangle}

Add \langle item\rangle to the end of the list in \langle list-macro\rangle, destructively updating \langle list-macro\rangle. These are local or global, and expanding or non-expanding, as with the \ConsTo variants.

\AppendTo{\langle src-list\rangle}{\langle dst-list\rangle}
\gAppendTo{\langle src-list\rangle}{\langle dst-list\rangle}

Append \langle src-list\rangle to \langle dst-list\rangle, leaving the result in \langle dst-list\rangle. As above, the \gAppendTo variant makes the change globally.
Remove the first element of ⟨src-list⟩ and put it in ⟨dst⟩. If ⟨src-list⟩ is empty, this is an error. These differ on the scope of the change:

\LopTo  Both ⟨src-list⟩ and ⟨dst⟩ are changed locally.
\gLopTo  Both ⟨src-list⟩ and ⟨dst⟩ are changed globally.
\glLopTo ⟨src-list⟩ is changed globally and ⟨dst⟩ is changed locally.
\lgLopTo ⟨src-list⟩ is changed locally and ⟨dst⟩ is changed globally.

Place the first element of ⟨src-list⟩ in ⟨dst⟩ without changing ⟨src-list⟩. The \gFirstTo variant makes the change globally.

Place all but the first element of ⟨src-list⟩ in ⟨dst-list⟩ without changing ⟨src-list⟩. The \gLastTo variant makes the change globally.

2.3 Control and Higher-Order List Commands

Tests whether ⟨cs⟩, which must be a control sequence, looks like a non-empty list. If so, it does ⟨then⟩, and otherwise it does ⟨else⟩. The control sequence ⟨cs⟩ “looks like a non-empty list” if its expansion starts with \listitem.

Iterate over ⟨list⟩, running ⟨body⟩ with ⟨cs⟩ set to each element.

Define ⟨dst⟩ to be the length of ⟨list⟩.


Map \( \langle each-expr \rangle \) over \( \langle src-list \rangle \), storing the result in \( \langle dst-list \rangle \). \( each-expr \) is not a command, but an expression that uses \#1 to refer to the list element. For example, to make every element of list \lst\ bold, storing the result bad to \lst, we could write:

\MapListTo{\textbf{#1}}\lst\lst

\MapAndAppendTo{\langle each-expr \rangle}{\langle src-list \rangle}{\langle dst-list \rangle}
\gMapAndAppendTo{\langle each-expr \rangle}{\langle src-list \rangle}{\langle dst-list \rangle}

Like \MapListTo and \gMapListTo, but appends the results of mapping to the end of \( \langle dst-list \rangle \) rather than overwriting it entirely.

\SortList{\langle key \rangle}{\langle list \rangle}

Sort the list \( \langle list \rangle \), putting the result back into \( \langle list \rangle \). By default, this sorts a list of integers. To sort a list with other contents, use \( \langle key \rangle \) to supply a command to extract integer keys from each element. (We currently don’t allow supplying an arbitrary comparison, but require a map from list elements to the integers.) Since \( \langle key \rangle \) is given each element as a single argument, it may need to go to a bit of trouble to split them up. For example, suppose we have a list of words and page numbers, perhaps for an index:

\def\lst\listitem{\IfList{3}}%  
\listitem{\SortList{5}}%  
\listitem{\MapList{4}}

Then to sort, we need a function that extracts the page numbers. However, the \( \langle key \rangle \) will be called with arguments like \{\SortList{5}\}, so it will need to unpack each item and then select the right thing:

\def\getkey#1{\@secondoftwo#1}
\SortList[\getkey]\lst

Note that it would not be sufficient to use \@secondoftwo directly, since it expects two arguments but \( \langle key \rangle \) gets only one.

\CompressList{\langle key \rangle}{\langle list \rangle}

Combines adjacent elements with adjacent keys into ranges and drops elements with duplicate keys. For this to work properly, the list should (usually) already be sorted.

If no \( \langle key \rangle \) is given, then like \SortList, it works on lists of integers. For example, given the list

\listitem{3}\listitem{4}\listitem{4}\listitem{5}\listitem{7},
it produces the list

\listitem{@\range{3}{5}}\listitem{@\single{7}}.

The macros @\range and @\single to indicate what is a single item and what is a range. To use such a list, usually one would redefine those macros to do the right thing in the particular instance.

If ⟨key⟩ is given, it is used, like in \SortList, to extract integer keys from the list elements. Note that when ranges are compressed, intermediate elements are lost.

2.4 Nice List Printing

This subsection describes a utility for presenting lists of things in typeset text:

\FormatList {⟨sing⟩} {⟨pl⟩} {⟨fmt⟩} {⟨csv⟩}

Format the comma-separated list in ⟨csv⟩, formatting each item with ⟨fmt⟩ and introducing the list with ⟨sing⟩ if it has only one item, or ⟨pl⟩ if it has more than one. Uses the separators defined below. For example:

<table>
<thead>
<tr>
<th>To get…</th>
<th>type…</th>
</tr>
</thead>
<tbody>
<tr>
<td>the letter A</td>
<td>\FormatList{the letter&quot;}{letters&quot;}{{}{A}</td>
</tr>
</tbody>
</table>

The commas, spacing, and word “and” are determined by three macros:

\FormatListSepTwo

Separates the two items of a two-item list. Initial definition is { and }.

\FormatListSepMore

Separates all but the last two items of a more-than-two-item list. Initial definition is {, }.

\FormatListSepLast

Separates the last two items of a more-than-two-item list. Initial definition is {, and }.

2.5 Functional List Expressions

We define a tiny, domain-specific language for writing functional list expressions. Rather than write a sequence of imperative list commands, it’s possible to compose an operation from several functional list operations.
Evaluate the list expression \( \langle \text{list-expr} \rangle \). The first, \ListExpr, then evaluates the resulting list in place, in the document. The other two define \( \langle \text{dst-list} \rangle \) to be the result. List expressions are built out of several list functions that are not defined generally, but only available inside of \( \langle \text{list-expr} \rangle \). Here is the syntax of list expressions:

\[
\langle \text{list-expr} \rangle ::= \langle \text{list-cs} \rangle \quad \text{(a list macro)}
| \Nil \quad \text{(the empty list)}
| \List \{ \langle \text{csv} \rangle \} \quad \text{(a comma-separated list)}
| \Q \{ \langle \text{text} \rangle \} \quad \text{(a list element)}
| \Cons \{ \langle \text{list-expr} \rangle \} \{ \langle \text{list-expr} \rangle \} \quad \text{(like \ConsTo)}
| \Snoc \{ \langle \text{list-expr} \rangle \} \{ \langle \text{list-expr} \rangle \} \quad \text{(like \SnocTo)}
| \Append \{ \langle \text{list-expr} \rangle \} \{ \langle \text{list-expr} \rangle \} \quad \text{(like \AppendTo)}
| \First \{ \langle \text{list-expr} \rangle \} \quad \text{(like \FirstTo)}
| \Rest \{ \langle \text{list-expr} \rangle \} \quad \text{(like \RestTo)}
| \Map \{ \langle \text{each-expr} \rangle \} \{ \langle \text{list-expr} \rangle \} \quad \text{(like \MapListTo)}
| \Sort \{ \langle \text{key} \rangle \} \{ \langle \text{list-expr} \rangle \} \quad \text{(like \SortList)}
| \Compress \{ \langle \text{key} \rangle \} \{ \langle \text{list-expr} \rangle \} \quad \text{(like \CompressList)}
| \langle \text{text} \rangle \quad \text{(arbitrary list element)}
\]

For example, we can sort and range-compress a comma-separated list, and then print it as an \itemize list with en dashes for ranges, like this:

\begin{verbatim}
\let\listitem\item
\let@single\relax
\def@range#1#2{#1--#2}
\begin{itemize}
  \ListExpr{\Compress{\Sort{\Cons{4}{\List{2,3,7,9,7,2,5}}}}}
\end{itemize}
\end{verbatim}

Here is the result:

- 2–5
- 7
- 9

3 Implementation

3.1 List Definition

The list defining commands are all defined in terms of the lower-level \@lstp@def, which takes two extra arguments—the first specifies whether the definition should be \global, and the second is the command to use to make the original definition.
\newcommand\newlist{\@lstp@def{}}
\newcommand\renewlist{\@lstp@def{}}
\newcommand\deflist{\@lstp@def{}}
\newcommand\gdeflist{\@lstp@def\global{}}

\@lstp@def
The low-level list defining command:
\newcommand\@lstp@def[4]{%
Initialize the output macro to empty, using #2, which is the requested defining command:
#2#3%
Iterate through the comma-separated list, adding elements to the end of the output macro:
\@for\lstp@def@temp:=#4\do{%
\expandafter\SnocTo\expandafter{\lstp@def@temp}#3%
}
This \let makes the result global if #1 is \global:
#1\let#3#3%
\let\lstp@def@temp\@undefined
}%

3.2 Basic Imperative List Commands
\lstp@ta
Token registers for list operations.
\newtoks\lstp@ta
\newtoks\lstp@tb

\ConsTo
The \ConsTo family of macros are defined in terms of the lower-level \@lstp@ConsTo macro, which takes two extra arguments: The first is a modifier that may be \global for global effect, and the second specifies which version of \def to use:
\newcommand\ConsTo{\@lstp@ConsTo\relax\def}
\newcommand\gConsTo{\@lstp@ConsTo\global\def}
\newcommand\eConsTo{\@lstp@ConsTo\relax\edef}
\newcommand\xConsTo{\@lstp@ConsTo\global\edef}

\@lstp@ConsTo
The low-level implementation of \ConsTo.
\newcommand\@lstp@ConsTo[4]{%
We define a temporary macro to be the item to cons, which may expand the item if #2 is \edef.
\@for\lstp@temp:=#4\do{%
\expandafter\listitem\expandafter{\lstp@temp}
}
#1\edef#4{\the\lstp@ta\the\lstp@tb}%
Similarly, the \SnocTo family is defined in terms of the lower-level \@lstp@SnocTo:
\begin{verbatim}
\newcommand\gSnocTo{\@lstp@SnocTo\global\def}
\newcommand\eSnocTo{\@lstp@SnocTo\relax\edef}
\newcommand\xSnocTo{\@lstp@SnocTo\global\edef}
\end{verbatim}
Like \@lstp@ConsTo, but appends the result in the other order.
\begin{verbatim}
\newcommand\@lstp@SnocTo[4]{\long#2\lstp@temp{#3}\lstp@ta=\expandafter{\expandafter\listitem\expandafter{\lstp@temp}}\lstp@tb=\expandafter{#4}\#1\edef#4{\the\lstp@tb\the\lstp@ta}}}
\end{verbatim}
As with \ConsTo and \SnocTo, we define these in terms of a lower-level macro that takes \global as an argument.
\begin{verbatim}
\newcommand\AppendTo{\@lstp@AppendTo\relax}
\newcommand\gAppendTo{\@lstp@AppendTo\global}
\newcommand\@lstp@AppendTo[3]{\lstp@ta=\expandafter{#2}\lstp@tb=\expandafter{#3}\#1\edef#3{\the\lstp@ta\the\lstp@tb}}}
\end{verbatim}
The key to taking lists apart is \@LopOff, which uses \TeX's pattern matching to find the first item in a list. In particular, we give \@LopOff the contents of a list, followed by \@LopOff to mark the end, and then parameters #3 and #4 tell us what to do to the first and the rest of the list. Then \@LopOff pattern matches the first list item and the rest of the list and passes the results to #3 and #4:
\begin{verbatim}
\long\def\@LopOff\listitem#1#2\@LopOff#3#4{#3#1#4#2}
\end{verbatim}
These are the low level commands for grabbing the first or rest of a list and sending it somewhere. Each uses \@LopOff to get the first and the rest, but passes it different continuations. \@lstp@LopTo passes it arguments that will \def its third and fourth arguments to be the first and rest of the lopped list, using global modifiers #1 and #2. On the other hand, \@lstp@RestTo ignores the first of the list and just defines its third argument to be the rest.
\begin{verbatim}
\newcommand\@lstp@LopTo[4]{\expandafter\@LopOff#3\@LopOff{#1\def#4}{#2\def#3}}
\newcommand\@lstp@RestTo[3]{\expandafter\@LopOff#2\@LopOff{\@gobble}{#1\def#3}}
\end{verbatim}
The \LopTo family is defined in terms of \@lstp@LopTo.
The \FirstTo family is defined in terms of \@lstp@LopTo, and the \RestTo family is defined using \@lstp@RestTo.
\newcommand\FirstTo{\@lstp@LopTo\relax\@gobblethree}
\newcommand\gFirstTo{\@lstp@LopTo\global\@gobblethree}
\newcommand\RestTo{\@lstp@RestTo\relax}
\newcommand\gRestTo{\@lstp@RestTo\global}

### 3.3 Control and Higher-Order List Commands

To detect if a macro is a list, we expand it and pass it to \@IfList, which grabs the first word or group in the macro and compares it to \listitem.
\newcommand*\IfList[1]{% 
  \expandafter\@IfList#1\@IfList
}
\def\@IfList#1#2\@IfList{\ifx\listitem#1\relax
\aftergroup\@firstoftwo\else
\aftergroup\@secondoftwo\fi}
\def\@forList#1:=#2\do#3{\long\def\lstp@for@listitem##1{\long\def#1{##1}#3\let\listitem\lstp@for@listitem\let#2\let\listitem\@undefined}}

The high-level \setToListLength command delegates to the lower-level \lstp@length, which defines \listitem to ignore its argument and increment a counter.
\newcommand\SetToListLength[2]{\lstp@length{#2}{\value{#1}}}
\MapListTo \MapAndAppendTo
\gMapListTo \gMapAndAppendTo

The mapping commands are defined, as usual, in terms of a lower level command
that takes a parameter that specifies whether to make a global definition or not.

Here we delegate to \@lstp@MapAndAppendTo to do the actual iteration. The main
work here is to save the contents of the source list before we clear the destination
list, in case they’re the same list. Then we map-and-append the copy of the source
list to the now-empty destination.

We define \listitem to add the map expression for each element to the end of
the list.

For sorting, we use an insertion sort, since that’s easy to define recursively using
the list operations that we have thus far. First, we define the insert function,
which takes as arguments the element to insert, the key projection, and a list to
insert it into.

Get the sorting key for the new item:

Save the input so we can use the same macro for output, and then clear the output:

We define two macros for looping, one which checks and inserts if necessary, and
the other which assumes insertion has happened:

Compute the key of the item to compare against, and then do the comparison.
If the item-to-insert should be inserted here, then we add it to the end of the list and redefine \listitem to insert without checking for here on out, which will insert the rest of the list after the new element.

\SnocTo{#1}{#3}\
\let\listitem\lstp@insert@listitem@done

Otherwise, it’s not time to insert it yet, so we just define list item to do this check again the next time around:

\let\listitem\lstp@insert@listitem

Add the next item in the input list to the end of the output list.

\SnocTo{##1}{#3}\

This is what we set \listitem to once we’ve inserted the item-to-add, so that it just adds the rest of the items one by one.

Initially, \listitem should compare the new item against each old item, so we define \listitem to be that function, and then run the loop by evaluating the (copy of the) input list.

\let\listitem\lstp@insert@listitem
\lstp@insert@temp@i%

If we haven’t inserted the new item yet by the end of the loop, we need to add it at the end:

\ifx\listitem\lstp@insert@listitem%
\SnocTo{#1}{#3}%
\fi%

Cleanup:

\let\lstp@insert@temp@i\@undefined%
\let\listitem\@undefined%
}

Once the single insertion command is defined, the insertion sort itself is straightforward. As the default key projection, we use \@apply@group{}, which merely removes the braces around the argument, which is sufficient to make something like \{5\} suitable for numeric comparison.

\providecommand\@apply@group[2]{#1#2}
\newcommand\SortList[2][\@apply@group{}]{%
Save the input so we can use the same macro for output, and then clear the output:

\let\lstp@sort@temp@i\@empty

For each item, insert it into the output. We redefine \listitem each time because \lstp@insert changes it.

\long\def\lstp@sort@listitem##1{%
Run the loop and cleanup.
\let\lstp@sort@temp@i\@undefined
\let\listitem\@undefined}

3.3.2 Compressing

To compress ranges, we need to determine if one integer is the successor of another.
\newcounter{lstp@ifsucc}
\newcommand\lstp@ifsucc[2]{% 
\setcounter{lstp@ifsucc}{#1}%
\addtocounter{lstp@ifsucc}{1}%
\ifnum#2=\value{lstp@ifsucc}%
\let\@lstp@ifsucc@kont\@firstoftwo
\else
\let\@lstp@ifsucc@kont\@secondoftwo
\fi
\@lstp@ifsucc@kont
}
\CompressList
List compression works like a state machine. State \texttt{start}, represented by the command \texttt{\lstp@compress@listitem@start}, is the starting state where we initialize the first value in the list. When we aren’t in a range, we stay in the \texttt{single} state, which compares the next item to the previous item and determines whether we have a duplicate (stay in \texttt{single}, but drop it), and non-successor (stay in \texttt{single}, and output the previous item), or the previous items successor (go to \texttt{range}, and no output). In the \texttt{range} state, we remember the start and end of the range, and compare the next item in the input list to the end of the range. The logic is similar to the \texttt{single} state: when we see a jump, we output the range and go back to \texttt{single}, and when we see a successor (or repeat), we extend (or leave alone) the current range.
\newcommand\CompressList[2]{\@apply@group{}{}}%
Save the input so we can use the same macro for output, and clear the output:
\let\lstp@compress@temp@i#2%
\let#2\empty
These are helper commands to add a single item or range to the list:
\def\lstp@compress@add@single{% 
\expandafter\SnocTo\expandafter
{\expandafter\@single\expandafter{\lstp@compress@temp@a}}{#2}%
}%
\def\lstp@compress@add@range{%
\CompressList
The state machine states are `\lstitem` callbacks. The first state is `start`, which saves the first item in `\lstp@compress@temp@a` (henceforce `a`) and its key in `\lstp@compress@temp@a@key` (henceforce `a@key`). It then transitions to the `single` state.

In the `single` state, we set `b` and `b@key` to the next item and its key. We then compare the keys. If they’re the same, we drop `b` on the floor and stay in this state. If `b@key` is the successor of `a@key`, then we transition to the `range` state. Otherwise, we output `a` as a singleton and let `a` be `b` for the next iteration.

In the `range` state, `a` is the start of the range and `b` is the end. We set `c` and `c@key` to the next item, and compare it to `b@key`. If they’re the same, we discard `c` and stay in the same state. If `c@key` is the successor of `b@key`, then `c` is the new upper bound of the range, so we let `b` be `c` for the next iteration. Otherwise, we’ve found the end of a range, so we output the range `a` to `b`, and transition back `single`, letting `a` be `c` for the next iteration.
We start in state `start` and go run the loop:

```
\let\listitem\lstp@compress@listitem@start
\lstp@compress@temp@i
```

When the loop terminates, if we were in `single`, we still need to output the single item `a`; if we were in `range`, we need to output the range `a` to `b`.

```
\ifx\listitem\lstp@compress@listitem@single
\lstp@compress@add@single
\else
\ifx\listitem\lstp@compress@listitem@range
\lstp@compress@add@range
\fi
\fi
```

Cleanup:

```
\let\lstp@compress@temp@i\@undefined
\let\lstp@compress@temp@c\@undefined
\let\lstp@compress@temp@c@key\@undefined
\let\lstp@compress@temp@b\@undefined
\let\lstp@compress@temp@b@key\@undefined
\let\lstp@compress@temp@a\@undefined
\let\lstp@compress@temp@a@key\@undefined
```

### 3.4 Nice List Printing

The default values for the list printing separators.

```
\FormatListSepTwo{ and }
\FormatListSepMore{, }
\FormatListSepLast{, and }
```

We use two counters for formatting a list: one for the whole length, and the other to keep track of the current position.

```
\c@lstp@FormatList@length
\c@lstp@FormatList@posn
```

14
The list formatting macro:

\newcommand\FormatList[4]{{%
Break the comma-separated input into a list and count it. Initialize the position
counter to 0.
\def\lstp@FormatList@list{#4}%
\SetToListLength{\lstp@FormatList@length}{\lstp@FormatList@list}%
\setcounter{lstp@FormatList@posn}{0}%
Introduce the list, using #1 for singular or #2 for plural:
\ifnum\value{lstp@FormatList@length}=1%
#1%
\else%
#2%
\fi%
What do to for each item—a decision tree:
\def\listitem##1{%
\addtocounter{lstp@FormatList@posn}{1}%
\ifnum1<\value{lstp@FormatList@posn}%
\ifnum2=\value{lstp@FormatList@length}%
\FormatListSepTwo%
\else%
\ifnum\value{lstp@FormatList@length}=\value{lstp@FormatList@posn}%
\FormatListSepLast%
\else%
\FormatListSepMore%
\fi
\fi
#3{##1}%
}%
Run the loop:
\lstp@FormatList@list%
}}

3.5 Function List Expressions

The main list expression commands are defined in terms of \lstp@ListExpr, which takes a list expression and a continuation for its result.
\newcommand\ListExpr[1]{\@lstp@ListExpr{#1}\relax}
\newcommand\ListExprTo[2]{\@lstp@ListExpr{#1}{\def#2}}
\newcommand\gListExprTo[2]{\@lstp@ListExpr{#1}{\gdef#2}}

\@lstp@defbinop\@lstp@defunop\@lstp@definplaceunopopt

These are helper macros for turning imperative list operations into the function list
operations for expressions. For example, \lstp@defbinop{⟨new-op⟩}{⟨old-op⟩}
defines ⟨new-op⟩ to be a two-argument list expression function, given ⟨old-op⟩, which is a two-argument list operation that redefines its second argument to be its result. Similarly, \lstp@defunop is helpful for defining unary list expression
functions. Finally \lstp@definplaceunopopt is for defining unary operations in terms of imperative unary operations that return their result in place and return an optional argument. This selection of helpers isn’t particularly general, but it handles the three necessary cases below.

Each of these depends on an \Eval operation, which given a list expression, evaluates it and globally defines \lstp@acc to be the result. Thus, to evaluate a binary operation, we evaluate the first operand, then save the result in a temporary, then evaluate the second operand, and then apply the operation to the temporary and the global accumulator. We do the second operand in a new group, so that it doesn’t overwrite the temporary. (The protocol is such that changes to the temporary must be local.)

\lstp@ListExpr Evaluating an expression mostly comes down to defining all the list expression functions and then recursively evaluating.

\lstp@ListExpr is the main evaluator, which dispatches on whether we’re looking at a list macro, one of the list expression operations, or some other arbitrary text. In the first case, it sets the accumulator to the list macro. If we have a list expression operation, it dispatches to that by evaluating it directly. Otherwise, it expands the argument and saves it in the accumulator.
Here we define the list expression operations:

\begin{verbatim}
\def\Q##1{\gdef\@lstp@acc{##1}}%
\def\Nil{\global\let\@lstp@acc\empty}%
\def\List##1{\gdef\@lstp@acc{##1}}%
\@lstp@defbinop\Cons\xConsTo
\@lstp@defbinop\Snoc\xSnocTo
\@lstp@defunop\First\gFirstTo
\@lstp@defunop\Rest\gRestTo
\@lstp@defbinop\Append\gAppendTo
\@lstp@definplaceunop\Sort\SortList
\@lstp@definplaceunop\Compress\CompressList
\newcommand\Map[2][]{\Eval{##2}{\gMapListTo{##1}\@lstp@acc\@lstp@acc}}%
\@lstp@finish##1{#2{##1}}%
\expandafter\@lstp@finish\expandafter{\@lstp@acc}%
\end{verbatim}

Start evaluating at the root:
\begin{verbatim}
\Eval{#1}%
\end{verbatim}

Call the continuation \texttt{#2} with the result:
\begin{verbatim}
\def\@lstp@finish##1{#2{##1}}%
\expandafter\@lstp@finish\expandafter{\@lstp@acc}
\end{verbatim}

To test whether a sequence of tokens represents a list expression functions, we check whether the first token is in a list of known list expression functions:
\begin{verbatim}
\def\@lstp@ifListOp#1#2\@lstp@ifListOp{\@lstp@ifInToks#1{\Q\Nil\List\Cons\Snoc\Append}
\@lstp@ifInToks
\newcommand\@lstp@ifInToks[2][]{%\def\@tester##1#1##2\@tester{\ifx\@notfound##2\relax\aftergroup\@secondoftwo\else\aftergroup\@firstoftwo\fi\gFirst\gRest\gAppend\Sort\Compress\Map}}%
\@lstp@finish##1{#2{##1}}%
\expandafter\@lstp@finish\expandafter{\@lstp@acc}
\end{verbatim}

We use \TeX's pattern matching to determine whether \#1 is present in \#2:
\begin{verbatim}
\def\@tester##1#1##2\@tester{%\if\notfound\#2\relax\aftergroup\@secondoftwo\else\aftergroup\@firstoftwo\fi\gFirst\gRest\gAppend\Sort\Compress\Map}}%
\@lstp@finish##1{#2{##1}}%
\expandafter\@lstp@finish\expandafter{\@lstp@acc}
\end{verbatim}
Change History

v0.1
   General: Initial documented release 1 and friends, which were expanding the contents of the initializer list, and shouldn’t be. ....... 7

v0.2
   \lstpdef: Fixed bug in \deflist

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.

Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>\IfList</td>
<td>56, 58, 61</td>
</tr>
<tr>
<td>\LopOff</td>
<td>42, 42, 46, 47</td>
</tr>
<tr>
<td>\apply@group</td>
<td></td>
</tr>
<tr>
<td>\forList</td>
<td>3, 68, 68</td>
</tr>
<tr>
<td>\gobblethree</td>
<td>52, 53</td>
</tr>
<tr>
<td>\lstp@appendTo</td>
<td>35, 35, 36, 37</td>
</tr>
<tr>
<td>\lstp@consTo</td>
<td>15, 16, 17, 18, 19</td>
</tr>
<tr>
<td>\lstp@listExpr</td>
<td>251, 252, 253, 274</td>
</tr>
<tr>
<td>\lstp@lopTo</td>
<td>46, 46, 48, 49, 50, 51, 52, 53</td>
</tr>
<tr>
<td>\lstp@mapAndappendTo</td>
<td>88, 89, 93, 96, 96</td>
</tr>
<tr>
<td>\lstp@mapListTo</td>
<td>86, 87, 90, 90</td>
</tr>
<tr>
<td>\lstp@nestTo</td>
<td>46, 47, 54, 55</td>
</tr>
<tr>
<td>\lstp@snocTo</td>
<td>25, 26, 27, 28, 29, 97</td>
</tr>
<tr>
<td>\lstp@acc</td>
<td>256, 258, 264, 270, 271, 276, 279, 284, 288, 289, 290, 300, 305</td>
</tr>
<tr>
<td>\lstp@def</td>
<td>1, 2, 3, 4, 5, 5</td>
</tr>
<tr>
<td>\lstp@defbinop</td>
<td>254, 254, 291, 292, 295</td>
</tr>
<tr>
<td>\lstp@definplaceunopopt</td>
<td></td>
</tr>
<tr>
<td>\lstp@defunop</td>
<td>254, 254, 291, 292, 295</td>
</tr>
<tr>
<td>\lstp@ifinToks</td>
<td>308, 313, 313, 322</td>
</tr>
<tr>
<td>\lstp@iflistUp</td>
<td>281, 307, 307</td>
</tr>
</tbody>
</table>

A

Append 6, 295, 309
AppendTo 2, 35, 35

C

\c@lstp@formatList@length 223
\c@lstp@formatList@posn 223
\c@lstp@ifsucc 132
\compressList 6, 297, 310
\compress 4, 148, 148, 297
\cons 6, 291, 309
\consto 2, 15, 15

D

\deflist 1, 3, 226
\do 3

E

\econsTo 2, 15, 17
\empty 1
\esnocTo 2, 25, 27
\eval 256, 257, 263, 269, 277, 299, 302
\Sort \SortList \xConsTo \xSnocTo

\Sort \SortList \xConsTo

6, 296, 310  2, 25, 28, 292
4, 124, 125, 296
2, 15, 18, 291