The l3sort package
Sorting lists*

The LaTeX3 Project†

Released 2012/07/16

1 l3sort documentation

LaTeX3 comes with a facility to sort list variables (sequences, token lists, or comma-lists) according to some user-defined comparison. For instance,

```
\clist_set:Nn \l_foo_clist { 3 , 01 , -2 , 5 , +1 }
\clist_sort:Nn \l_foo_clist
  {
    \int_compare:nNnTF { #1 } > { #2 }
      { \sort_reversed: }
      { \sort_ordered: }
  }
```

will result in `\l_foo_clist` holding the values `{ -2 , 01 , +1 , 3 , 5 }` sorted in non-decreasing order.

The code defining the comparison should perform `\sort_reversed:` if the two items given as `#1` and `#2` are not in the correct order, and otherwise it should call `\sort_ordered:` to indicate that the order of this pair of items should not be changed.

For instance, a ⟨*comparison code*⟩ consisting only of `\sort_ordered:` with no test will yield a trivial sort: the final order is identical to the original order. Conversely, using a ⟨*comparison code*⟩ consisting only of `\sort_reversed:` will reverse the list (in a fairly inefficient way).

TeXhackers note: Internally, the code from l3sort stores items in `\toks`. Thus, the ⟨*comparison code*⟩ should not alter the contents of any `\toks`, nor assume that they hold a given value.

`\seq_sort:Nn`
`\seq_gsort:Nn`

`\seq_sort:Nn` ⟨*sequence*⟩ {⟨*comparison code*⟩}

Sorts the items in the ⟨*sequence*⟩ according to the ⟨*comparison code*⟩, and assigns the result to ⟨*sequence*⟩.

*This file describes v3991, last revised 2012/07/16.

†E-mail: latex-team@latex-project.org

\tl_sort:Nn
\tl_gsort:Nn

`\tl_sort:Nn` ⟨*tl var*⟩ {⟨*comparison code*⟩}

Sorts the items in the ⟨*tl var*⟩ according to the ⟨*comparison code*⟩, and assigns the result to ⟨*tl var*⟩.

\clist_sort:Nn
\clist_gsort:Nn

`\clist_sort:Nn` ⟨*clist var*⟩ {⟨*comparison code*⟩}

Sorts the items in the ⟨*clist var*⟩ according to the ⟨*comparison code*⟩, and assigns the result to ⟨*clist var*⟩.

\tl_sort:nN ⋆

`\tl_sort:nN` {⟨*token list*⟩} ⟨*conditional*⟩

Sorts the items in the ⟨*token list*⟩, using the ⟨*conditional*⟩ to compare items, and leaves the result in the input stream. The ⟨*conditional*⟩ should have signature `:nnTF`, and return `true` if the two items being compared should be left in the same order, and `false` if the items should be swapped.

2 l3sort implementation

```
⟨*initex | package⟩
⟨@@=sort⟩
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
⟨/package⟩
```

2.1 Variables

\c__sort_max_length_int

The maximum length of a sequence which will not overflow the available registers depends on which engine is in use. For 2^N registers, it is $3 \cdot 2^{N-2}$: for that number of items, at the last step the block size will be 2^{N-1}, and the two blocks to merge will be of sizes 2^{N-1} and 2^{N-2} respectively. When merging, one of the blocks must be copied to temporary registers; here, the smallest block, of size 2^{N-2}, will fill up exactly the 2^{N-2} free registers, totalling $2^{N-1} + 2^{N-2} + 2^{N-2} = 2^N$ registers.

```
\int_const:Nn \c__sort_max_length_int
  { \luatex_if_engine:TF { 49152 } { 24576 } }
```

(*End definition for* `\c__sort_max_length_int` *This variable is documented on page* ??.)

\l__sort_length_int

Length of the sequence which is being sorted.

```
\int_new:N \l__sort_length_int
```

(*End definition for* `\l__sort_length_int` *This variable is documented on page* ??.)

\l__sort_block_int

Merge sort is done in several passes. In each pass, blocks of size `\l__sort_block_int` are merged in pairs. The block size starts at 1, and, for a length in the range $[2^k + 1, 2^{k+1}]$, reaches 2^k in the last pass.

```
\int_new:N \l__sort_block_int
```

(*End definition for* `\l__sort_block_int` *This variable is documented on page* ??.)

\l__sort_begin_int
\l__sort_end_int

When merging two blocks, `\l__sort_begin_int` marks the lowest index in the two blocks, and `\l__sort_end_int` marks the highest index, plus 1.

```
\int_new:N \l__sort_begin_int
\int_new:N \l__sort_end_int
```

(*End definition for* `\l__sort_begin_int` *This function is documented on page* ??.)

\l__sort_A_int
\l__sort_B_int
\l__sort_C_int

When merging two blocks (whose end-points are `beg` and `end`), A starts from the high end of the low block, and decreases until reaching `beg`. The index B starts from the top of the range and marks the register in which a sorted item should be put. Finally, C points to the copy of the high block in the interval of registers starting at `\l__sort_length_int`, upwards. C starts from the upper limit of that range.

```
\int_new:N \l__sort_A_int
\int_new:N \l__sort_B_int
\int_new:N \l__sort_C_int
```

(*End definition for* `\l__sort_A_int` *This function is documented on page* ??.)

2.2 Expandable sorting

Sorting expandably is very different from sorting and assigning to a variable. Since tokens cannot be stored, they must remain in the input stream, and be read through at every step. It is thus necessarily much slower (at best $O(n^2)$) than non-expandable sorting functions.

A prototypical version of expandable quicksort is as follows. If the argument has no item, return nothing, otherwise partition, using the first item as a pivot (argument `#4` of `\__sort:nnNnn`). The arguments of `\__sort:nnNnn` are 1. items less than `#4`, 2. items greater than `#4`, 3. comparison, 4. pivot, 5. next item to test. If `#5` is the tail of the list, call `\tl_sort:nN` on `#1` and on `#2`, placing `#4` in between; `\use:ff` expands the parts to make `\tl_sort:nN` `f`-expandable. Otherwise, compare `#4` and `#5` using `#3`. If they are ordered, place `#5` amongst the "greater" items, otherwise amongst the "lesser" items, and continue partitioning.

```
\cs_new:Npn \tl_sort:nN #1#2
  {
    \tl_if_blank:nF {#1}
      {
        \__sort:nnNnn { } { } #2
          #1 \q_recursion_tail \q_recursion_stop
      }
  }
\cs_new:Npn \__sort:nnNnn #1#2#3#4#5
  {
    \quark_if_recursion_tail_stop_do:nn {#5}
      { \use:ff { \tl_sort:nN {#1} #3 {#4} } { \tl_sort:nN {#2} #3 } }
    #3 {#4} {#5}
      { \__sort:nnNnn {#1} { #2 {#5} } #3 {#4} }
      { \__sort:nnNnn { #1 {#5} } {#2} #3 {#4} }
```

```
  }
\cs_generate_variant:Nn \use:nn { ff }
```

There are quite a few optimizations available here: the code below is less legible, but twice as fast.

\tl_sort:nN
__sort_quick_i:nNn
__sort_quick_ii:wn
__sort_quick_iii:nnNnn
__sort_quick_iv:nnNnwn

The `\__sort_quick_i:nNn` function sorts `#1` using the comparison `#2`, then places its third argument (⟨*continuation*⟩) before the result. This essentially avoids the rather slow `\use:ff`. First test if `#1` is blank: if it is, `\use_none:n` eats the break point, and `\__sort_quick_ii:wn` removes all until the ending break point, unbracing the ⟨*continuation*⟩. If it is not blank, `\__sort_quick_ii:wn` simply removes the end of that test until the break point. The third auxiliary plays the role of `\__sort:nnNnn` above, with an end-test replaced by `\__prg_break:`, which skips to the break point two lines later, unless `#5` is itself a break point. In that case, `\__sort_quick_iv:nnNnwn` calls the first auxiliary, which sorts `#2` and places another call to itself before the result.

```
\cs_new:Npn \tl_sort:nN #1#2 { \__sort_quick_i:nNn {#1} #2 { } }
\cs_new:Npn \__sort_quick_i:nNn #1#2
  {
    \exp_after:wN \__sort_quick_ii:wn
      \use_none:n #1 \__prg_break_point:
    \__sort_quick_iii:nnNnn { } { } #2
      #1
      { \__prg_break_point: }
      \__prg_break_point:
  }
\cs_new:Npn \__sort_quick_ii:wn #1 \__prg_break_point: #2 {#2}
\cs_new:Npn \__sort_quick_iii:nnNnn #1#2#3#4#5
  {
    \__prg_break: #5
      \__sort_quick_iv:nnNnwn {#1} {#2}
      \__prg_break_point:
    #3 {#4} {#5}
      { \__sort_quick_iii:nnNnn {#1} { #2 {#5} } }
      { \__sort_quick_iii:nnNnn { #1 {#5} } {#2} }
          #3 {#4}
  }
\cs_new:Npn \__sort_quick_iv:nnNnwn
    #1#2 \__prg_break_point: #3#4 #5 \__prg_break_point: #6
  {
    \__sort_quick_i:nNn {#2} #3
      { \__sort_quick_i:nNn {#1} #3 {#6} {#4} }
  }
```

(*End definition for* `\tl_sort:nN` *This function is documented on page 2.*)

2.3 Protected user commands

__sort_main:NNNnNn

Sorting happens in three steps. First store items in `\toks` registers ranging from 0 to the length of the list, while checking that the list is not too long. If we reach the maximum

length, all further items are entirely ignored after raising an error. Secondly, sort the array of `\toks` registers, using the user-defined sorting function, `#5`. Finally, unpack the `\toks` registers (now sorted) into a variable of the right type, by `x`-expanding the code in `#3`, specific to each type of list.

```
\cs_new_protected:Npn \__sort_main:NNNnNn #1#2#3#4#5#6
  {
    \group_begin:
      \l__sort_length_int \c_zero
      #2 #5
        {
          \if_int_compare:w \l__sort_length_int = \c__sort_max_length_int
            \__sort_too_long_error:NNw #3 #5
          \fi:
          \tex_toks:D \l__sort_length_int {##1}
          \tex_advance:D \l__sort_length_int \c_one
        }
      \cs_set:Npn \sort_compare:nn ##1 ##2 { #6 }
      \l__sort_block_int \c_one
      \__sort_level:
      \use:x
        {
          \group_end:
          #1 \exp_not:N #5 {#4}
        }
  }
```

(*End definition for* `\__sort_main:NNNnNn`)

`\seq_sort:Nn` `\seq_gsort:Nn`

The first argument to `\__sort_main:NNNnNn` is the final assignment function used, either `\tl_set:Nn` or `\tl_gset:Nn` to control local versus global results. The second argument is what mapping function is used when storing items to `\toks` registers. The third is used to build back the correct kind of list from the contents of the `\toks` registers. Fourth and fifth arguments are the variable to sort, and the sorting method as inline code.

```
\cs_new_protected_nopar:Npn \seq_sort:Nn
  {
    \__sort_main:NNNnNn \tl_set:Nn
      \seq_map_inline:Nn \seq_map_break:
      { \__sort_toks:NNw \exp_not:N \__seq_item:n 0 ; }
  }
\cs_new_protected_nopar:Npn \seq_gsort:Nn
  {
    \__sort_main:NNNnNn \tl_gset:Nn
      \seq_map_inline:Nn \seq_map_break:
      { \__sort_toks:NNw \exp_not:N \__seq_item:n 0 ; }
  }
```

(*End definition for* `\seq_sort:Nn` *and* `\seq_gsort:Nn` *These functions are documented on page* 1.)

`\tl_sort:Nn` `\tl_gsort:Nn`

Again, use `\tl_set:Nn` or `\tl_gset:Nn` to control the scope of the assignment. Mapping through the token list is done with `\tl_map_inline:Nn`, and producing the token list is very similar to sequences, removing `\seq_item:Nn`.

```
\cs_new_protected_nopar:Npn \tl_sort:Nn
  {
    \__sort_main:NNNnNn \tl_set:Nn
      \tl_map_inline:Nn \tl_map_break:
      { \__sort_toks:NNw \prg_do_nothing: \prg_do_nothing: 0 ; }
  }
\cs_new_protected_nopar:Npn \tl_gsort:Nn
  {
    \__sort_main:NNNnNn \tl_gset:Nn
      \tl_map_inline:Nn \tl_map_break:
      { \__sort_toks:NNw \prg_do_nothing: \prg_do_nothing: 0 ; }
  }
```

(*End definition for* `\tl_sort:Nn` *and* `\tl_gsort:Nn` *These functions are documented on page 2.*)

`\clist_sort:Nn`
`\clist_gsort:Nn`
`\__sort_sort:NNn`

The case of empty comma-lists is a little bit special as usual, and filtered out: there is nothing to sort in that case. Otherwise, the input is done with `\clist_map_inline:Nn`, and the output requires some more elaborate processing than for sequences and token lists. The first comma must be removed. An item must be wrapped in an extra set of braces if it contains either the space or the comma characters. This is taken care of by `\clist_wrap_item:n`, but `\__sort_toks:NNw` would simply feed `\tex_the:D` `\tex_toks:D` ⟨*number*⟩ as an argument to that function; hence we need to expand this argument once to unpack the register.

```
\cs_new_protected_nopar:Npn \clist_sort:Nn
  { \__sort_sort:NNn \tl_set:Nn }
\cs_new_protected_nopar:Npn \clist_gsort:Nn
  { \__sort_sort:NNn \tl_gset:Nn }
\cs_new_protected:Npn \__sort_sort:NNn #1#2#3
  {
    \clist_if_empty:NF #2
      {
        \__sort_main:NNNnNn #1
          \clist_map_inline:Nn \clist_map_break:
          {
            \exp_last_unbraced:Nf \use_none:n
              { \__sort_toks:NNw \exp_args:No \__clist_wrap_item:n 0 ; }
          }
          #2 {#3}
      }
  }
```

(*End definition for* `\clist_sort:Nn` *and* `\clist_gsort:Nn` *These functions are documented on page 2.*)

`\__sort_toks:NNw`

Unpack the various `\toks` registers, from 0 to the length of the list. The functions `#1` and `#2` allow us to treat the three data structures in a unified way:

- for sequences, they are `\exp_not:N` `\__seq_item:n`, expanding to the `\__seq_item:n` separator, as expected;
- for token lists, they expand to nothing;

- for comma lists, they expand to `\exp_args:No \clist_wrap_item:n`, taking care of unpacking the register before letting the undocumented internal clist function `\clist_wrap_item:n` do the work of putting a comma and possibly braces.

```
\cs_new:Npn \__sort_toks:NNw #1#2#3 ;
  {
    \if_int_compare:w #3 < \l__sort_length_int
      #1 #2 { \tex_the:D \tex_toks:D #3 }
      \exp_after:wN \__sort_toks:NNw \exp_after:wN #1 \exp_after:wN #2
      \int_use:N \__int_eval:w #3 + \c_one \exp_after:wN ;
    \fi:
  }
```

(*End definition for* `\__sort_toks:NNw` *This function is documented on page* ??.)

2.4 Sorting itself

`\__sort_level:` This function is called once blocks of size `\l__sort_block_int` (initially 1) are each sorted. If the whole list fits in one block, then we are done (this also takes care of the case of an empty list or a list with one item). Otherwise, go through pairs of blocks starting from 0, then double the block size, and repeat.

```
\cs_new_protected_nopar:Npn \__sort_level:
  {
    \if_int_compare:w \l__sort_block_int < \l__sort_length_int
      \l__sort_end_int \c_zero
      \__sort_merge_blocks:
      \tex_multiply:D \l__sort_block_int \c_two
      \exp_after:wN \__sort_level:
    \fi:
  }
```

(*End definition for* `\__sort_level:`)

`\__sort_merge_blocks:` This function is called to merge a pair of blocks, starting at the last value of `\l__sort_end_int` (end-point of the previous pair of blocks). If shifting by one block to the right we reach the end of the list, then this pass has ended: this end of the list is sorted already. Store the result of that shift in A, which will index the first block starting from the top end. Then locate the end-point (maximum) of the upper block: shift `end` upwards by one more block, checking that we don't go beyond the length of the list. Copy this upper block of `\toks` registers in registers above `length`, indexed by C: this is covered by `\sort_copy_block:`. Once this is done we are ready to do the actual merger using `\__sort_merge_blocks_aux:`, after shifting A, B and C so that they point to the largest index in their respective ranges rather than pointing just beyond those ranges. Of course, once that pair of blocks is merged, move on to the next pair.

```
\cs_new_protected_nopar:Npn \__sort_merge_blocks:
  {
    \l__sort_begin_int \l__sort_end_int
    \tex_advance:D \l__sort_end_int \l__sort_block_int
    \if_int_compare:w \__int_eval:w \l__sort_end_int < \l__sort_length_int
      \l__sort_A_int \l__sort_end_int
```

```
      \tex_advance:D \l__sort_end_int \l__sort_block_int
      \if_int_compare:w \l__sort_end_int > \l__sort_length_int
        \l__sort_end_int \l__sort_length_int
      \fi:
      \l__sort_B_int \l__sort_A_int
      \l__sort_C_int \l__sort_length_int
      \sort_copy_block:
      \tex_advance:D \l__sort_A_int \c_minus_one
      \tex_advance:D \l__sort_B_int \c_minus_one
      \tex_advance:D \l__sort_C_int \c_minus_one
      \__sort_merge_blocks_aux:
      \exp_after:wN \__sort_merge_blocks:
    \fi:
  }
```

(*End definition for* `\__sort_merge_blocks:`)

`\sort_copy_block:` We wish to store a copy of the "upper" block of `\toks` registers, ranging between the initial value of `\l__sort_B_int` (included) and `\l__sort_end_int` (excluded) into a new range starting at the initial value of `\l__sort_C_int`, namely `\l__sort_length_int`.

```
\cs_new_protected_nopar:Npn \sort_copy_block:
  {
    \tex_toks:D \l__sort_C_int \tex_toks:D \l__sort_B_int
    \tex_advance:D \l__sort_C_int \c_one
    \tex_advance:D \l__sort_B_int \c_one
    \if_int_compare:w \l__sort_B_int = \l__sort_end_int
      \use_i:nn
    \fi:
    \sort_copy_block:
  }
```

(*End definition for* `\sort_copy_block:`)

`\__sort_merge_blocks_aux:` At this stage, the first block starts at `\l__sort_begin_int`, and ends at `\l__sort_A_int`, and the second block starts at `\l__sort_length_int` and ends at `\l__sort_C_int`. The result of the merger is stored at positions indexed by `\l__sort_B_int`, which starts at `\l__sort_end_int` -1 and decreases down to `\l__sort_begin_int`, covering the full range of the two blocks. In other words, we are building the merger starting with the largest values. The comparison function is defined to return either `reversed` or `ordered`. Of course, this means the arguments need to be given in the order they appear originally in the list.

```
\cs_new_protected_nopar:Npn \__sort_merge_blocks_aux:
  {
    \exp_after:wN \sort_compare:nn \exp_after:wN
      { \tex_the:D \tex_toks:D \exp_after:wN \l__sort_A_int \exp_after:wN }
      \exp_after:wN { \tex_the:D \tex_toks:D \l__sort_C_int }
  }
```

(*End definition for* `\__sort_merge_blocks_aux:`)

\sort_ordered: If the comparison function returns `ordered`, then the second argument fed to `\sort_compare:nn` should remain to the right of the other one. Since we build the merger starting from the right, we copy that `\toks` register into the allotted range, then shift the pointers B and C, and go on to do one more step in the merger, unless the second block has been exhausted: then the remainder of the first block is already in the correct register and we are done with merging those two blocks.

```
\cs_new_protected_nopar:Npn \sort_ordered:
  {
    \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
    \tex_advance:D \l__sort_B_int \c_minus_one
    \tex_advance:D \l__sort_C_int \c_minus_one
    \if_int_compare:w \l__sort_C_int < \l__sort_length_int
      \use_i:nn
    \fi:
    \__sort_merge_blocks_aux:
  }
```

(*End definition for* `\sort_ordered:`)

\sort_reversed: If the comparison function returns `reversed`, then the next item to add to the merger is the first argument, contents of the `\toks` register A. Then shift the pointers A and B to the left, and go for one more step fo the merger, unless the left block was exhausted (A goes below the threshold). In that case, all remaining `\toks` registers in the second block, indexed by C, should be copied to the merger (see `\__sort_merge_blocks_end:`).

```
\cs_new_protected_nopar:Npn \sort_reversed:
  {
    \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_A_int
    \tex_advance:D \l__sort_B_int \c_minus_one
    \tex_advance:D \l__sort_A_int \c_minus_one
    \if_int_compare:w \l__sort_A_int < \l__sort_begin_int
      \__sort_merge_blocks_end: \use_i:nn
    \fi:
    \__sort_merge_blocks_aux:
  }
```

(*End definition for* `\sort_reversed:`)

__sort_merge_blocks_end: This function's task is to copy the `\toks` registers in the block indexed by C to the merger indexed by B. The end can equally be detected by checking when B reaches the threshold `begin`, or when C reaches `length`.

```
\cs_new_protected_nopar:Npn \__sort_merge_blocks_end:
  {
    \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
    \tex_advance:D \l__sort_B_int \c_minus_one
    \tex_advance:D \l__sort_C_int \c_minus_one
    \if_int_compare:w \l__sort_B_int < \l__sort_begin_int
      \use_i:nn
    \fi:
    \__sort_merge_blocks_end:
  }
```

(*End definition for* `\__sort_merge_blocks_end:`)

2.5 Messages

`\__sort_too_long_error:NNw` When there are too many items in a sequence, this is an error, and we clean up properly the mapping over items in the list: break using the type-specific breaking function `#1`.

```
\cs_new_protected:Npn \__sort_too_long_error:NNw #1#2 \fi:
  {
    \fi:
    \__msg_kernel_error:nnx { sort } { too-large } { \token_to_str:N #2 }
    #1
  }
\__msg_kernel_new:nnnn { sort } { too-large }
  { The~list~#1~is~too~long~to~be~sorted~by~TeX. }
  {
    TeX~has~\int_eval:n { \c_max_register_int + 1 }~registers~available:~
    this~only~allows~to~sorts~with~up~to~\int_use:N \c__sort_max_length_int
    \ items.~All~extra~items~will~be~ignored.
  }
```

(*End definition for* `\__sort_too_long_error:NNw` *This function is documented on page* **??**.)

```
⟨/initex | package⟩
```

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.