The l3sort package
Sorting lists*

The LaTeX3 Project†

Released 2013/05/12

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 v4491, last revised 2013/05/12.

†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.

TEXhackers note: The result is returned within `\exp_not:n`, which means that the token list will not expand further when appearing in an `x`-type argument expansion.

2 l3sort implementation

```
1 ⟨*initex | package⟩
2 ⟨@@=sort⟩
3 ⟨*package⟩
4 \ProvidesExplPackage
5   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
6 ⟨/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.

```
7 \int_const:Nn \c__sort_max_length_int
8   { \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.

```
9 \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 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_clist: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_clist:NNn \tl_set:Nn }
\cs_new_protected_nopar:Npn \clist_gsort:Nn
  { \__sort_clist:NNn \tl_gset:Nn }
\cs_new_protected:Npn \__sort_clist: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* ??.)

2.3 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 for 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:`.)

2.4 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 $O(n^2 \ln n)$) than non-expandable sorting functions ($O(n \ln n)$).

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 or equal to `#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 more than twice as fast.

In the simple version of the code, `\__sort:nnNnn` is called $O(n \ln n)$ times on average (the number of comparisons required by the quicksort algorithm). Hence most of our focus will be on optimizing that function.

The first speed up is to avoid testing for the end of the list at every call to `\__sort:nnNnn`. For this, the list is prepared by changing each ⟨*item*⟩ of the original token list into ⟨*command*⟩ `{`⟨*item*⟩`}`, just like sequences are stored. We arrange things such that the ⟨*command*⟩ is the ⟨*conditional*⟩ provided by the user: the loop over the ⟨*prepared tokens*⟩ then looks like

```
\cs_new:Npn \__sort_loop:wNn ... #6#7
{
#6 {⟨pivot⟩} {#7} ⟨loop big⟩ ⟨loop small⟩
⟨extra arguments⟩
}
\__sort_loop:wNn ... ⟨prepared tokens⟩
⟨end-loop⟩ {} \q_stop
```

In this example, which matches the structure of `\__sort_quick_split_i:NnnnnNn` and a few other functions below, the `\__sort_loop:wNn` auxiliary normally receives the user's ⟨*conditional*⟩ as `#6` and an ⟨*item*⟩ as `#7`. This is compared to the ⟨*pivot*⟩ (the argument `#5`, not shown here), and the ⟨*conditional*⟩ leaves the ⟨*loop big*⟩ or ⟨*loop small*⟩ auxiliary, which both have the same form as `\__sort_loop:wNn`, receiving the next pair ⟨*conditional*⟩ `{`⟨*item*⟩`}` as `#6` and `#7`. At the end, `#6` is the ⟨*end-loop*⟩ function, which terminates the loop.

The second speed up is to minimize the duplicated tokens between the `true` and `false` branches of the conditional. For this, we introduce two versions of `\__sort:nnNnn`,

which receive the new item as `#1` and place it either into the list `#2` of items less than the pivot `#4` or into the list `#3` of items greater or equal to the pivot.

```
\cs_new:Npn \__sort_i:nnnnNn #1#2#3#4#5#6
  {
    #5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i:nnnnNn
      {#6} { #2 {#1} } {#3} {#4}
  }
\cs_new:Npn \__sort_ii:nnnnNn #1#2#3#4#5#6
  {
    #5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i:nnnnNn
      {#6} {#2} { #3 {#1} } {#4}
  }
```

Note that the two functions have the form of `\__sort_loop:wNn` above, receiving as `#5` the conditional or a function to end the loop. In fact, the lists `#2` and `#3` must be made of pairs ⟨*conditional*⟩ `{`⟨*item*⟩`}`, so we have to replace `{#6}` above by `{ #5 {#6} }`, and `{#1}` by `#1`. The actual functions have one more argument, so all argument numbers are shifted compared to this code.

The third speed up is to avoid `\use:ff` using a continuation-passing style: `\__sort_quick_split:NnNn` expects a list followed by `\q_mark` `{`⟨*code*⟩`}`, and expands to ⟨*code*⟩ ⟨*sorted list*⟩. Sorting the two parts of the list around the pivot is done with

```
\__sort_quick_split:NnNn #2 ...  \q_mark
{
\__sort_quick_split:NnNn #1 ...  \q_mark {⟨code⟩}
{⟨pivot⟩}
}
```

Items which are larger than the ⟨*pivot*⟩ are sorted, then placed after code that sorts the smaller items, and after the (braced) ⟨*pivot*⟩.

The fourth speed up is avoid the recursive call to `\tl_sort:nN` with an empty first argument. For this, we introduce functions similar to the `\__sort_i:nnnnNn` of the last example, but aware of whether the list of ⟨*conditional*⟩ `{`⟨*item*⟩`}` read so far that are less than the pivot, and the list of those greater or equal, are empty or not: see `\__sort_quick_split:NnNn` and functions defined below. Knowing whether the lists are empty or not is useless if we do not use distinct ending codes as appropriate. The splitting auxiliaries communicate to the ⟨*end-loop*⟩ function (that is initially placed after the "prepared" list) by placing a specific ending function, ignored when looping, but useful at the end. In fact, the ⟨*end-loop*⟩ function does nothing but place the appropriate ending function in front of all its arguments. The ending functions take care of sorting non-empty sublists, placing the pivot in between, and the continuation before.

The final change in fact slows down the code a little, but is required to avoid memory issues: schematically, when TEX encounters

```
\use:n { \use:n { \use:n { ... } ... } ... }
```

the argument of the first `\use:n` is not completely read by the second `\use:n`, hence must remain in memory; then the argument of the second `\use:n` is not completely read when grabbing the argument of the third `\use:n`, hence must remain in memory, and so on. The memory consumption grows quadratically with the number of nested `\use:n`. In practice, this means that we must read everything until a trailing `\q_stop` once in a while, otherwise sorting lists of more than a few thousand items would exhaust a typical TEX's memory.

`\tl_sort:nN`
`\__sort_quick_prepare:Nnnn`
`\__sort_quick_prepare_end:NNNnw`
`\__sort_quick_cleanup:w`

The code within the `\exp_not:f` sorts the list, leaving in most cases a leading `\exp_not:f`, which stops the expansion, letting the result be return within `\exp_not:n`. We filter out the case of a list with no item, which would otherwise cause problems. Then prepare the token list `#1` by inserting the conditional `#2` before each item. The `prepare` auxiliary receives the conditional as `#1`, the prepared token list so far as `#2`, the next prepared item as `#3`, and the item after that as `#4`. The loop ends when `#4` contains `\__prg_break_point:`, then the `prepare_end` auxiliary finds the prepared token list as `#4`. The scene is then set up for `\__sort_quick_split:NnNn`, which will sort the prepared list and perform the post action placed after `\q_mark`, namely removing the trailing `\s__stop` and `\q_stop` and leaving `\exp_stop_f:` to stop `f`-expansion.

```
\cs_new:Npn \tl_sort:nN #1#2
  {
    \exp_not:f
      {
        \tl_if_blank:nF {#1}
          {
            \__sort_quick_prepare:Nnnn #2 { } { }
              #1
              { \__prg_break_point: \__sort_quick_prepare_end:NNNnw }
            \q_stop
          }
      }
  }
\cs_new:Npn \__sort_quick_prepare:Nnnn #1#2#3#4
  {
    \__prg_break: #4 \__prg_break_point:
    \__sort_quick_prepare:Nnnn #1 { #2 #3 } { #1 {#4} }
  }
\cs_new:Npn \__sort_quick_prepare_end:NNNnw #1#2#3#4#5 \q_stop
  {
    \__sort_quick_split:NnNn #4 \__sort_quick_end:nnTFNn { }
    \q_mark { \__sort_quick_cleanup:w \exp_stop_f: }
    \s__stop \q_stop
  }
\cs_new:Npn \__sort_quick_cleanup:w #1 \s__stop \q_stop {#1}
```

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

`\__sort_quick_split:NnNn`
`\__sort_quick_only_i:NnnnnNn`
`\__sort_quick_only_ii:NnnnnNn`
`\__sort_quick_split_i:NnnnnNn`
`\__sort_quick_split_ii:NnnnnNn`

The `only_i`, `only_ii`, `split_i` and `split_ii` auxiliaries receive a useless first argument, the new item `#2` (that they append to either one of the next two arguments), the list `#3` of items less than the pivot, bigger items `#4`, the pivot `#5`, a ⟨*function*⟩ `#6`, and an

item #7. The ⟨*function*⟩ is the user's ⟨*conditional*⟩ except at the end of the list where it is `\__sort_quick_end:nnTFNn`. The comparison is applied to the ⟨*pivot*⟩ and the ⟨*item*⟩, and calls the `only_i` or `split_i` auxiliaries if the ⟨*item*⟩ is smaller, and the `only_ii` or `split_ii` auxiliaries otherwise. In both cases, the next auxiliary goes to work right away, with no intermediate expansion that would slow down operations. Note that the argument #2 left for the next call has the form ⟨*conditional*⟩ `{`⟨*item*⟩`}`, so that the lists #3 and #4 keep the right form to be fed to the next sorting function. The `split` auxiliary differs from these in that it is missing three of the arguments, which would be empty, and its first argument is always the user's ⟨*conditional*⟩ rather than an ending function.

```
\cs_new:Npn \__sort_quick_split:NnNn #1#2#3#4
  {
    #3 {#2} {#4} \__sort_quick_only_ii:NnnnnNn \__sort_quick_only_i:NnnnnNn
      \__sort_quick_single_end:nnnwnw
      { #3 {#4} } { } { } {#2}
  }
\cs_new:Npn \__sort_quick_only_i:NnnnnNn #1#2#3#4#5#6#7
  {
    #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn \__sort_quick_only_i:NnnnnNn
      \__sort_quick_only_i_end:nnnwnw
      { #6 {#7} } { #3 #2 } { } {#5}
  }
\cs_new:Npn \__sort_quick_only_ii:NnnnnNn #1#2#3#4#5#6#7
  {
    #6 {#5} {#7} \__sort_quick_only_ii:NnnnnNn \__sort_quick_split_i:NnnnnNn
      \__sort_quick_only_ii_end:nnnwnw
      { #6 {#7} } { } { #4 #2 } {#5}
  }
\cs_new:Npn \__sort_quick_split_i:NnnnnNn #1#2#3#4#5#6#7
  {
    #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn \__sort_quick_split_i:NnnnnNn
      \__sort_quick_split_end:nnnwnw
      { #6 {#7} } { #3 #2 } {#4} {#5}
  }
\cs_new:Npn \__sort_quick_split_ii:NnnnnNn #1#2#3#4#5#6#7
  {
    #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn \__sort_quick_split_i:NnnnnNn
      \__sort_quick_split_end:nnnwnw
      { #6 {#7} } {#3} { #4 #2 } {#5}
  }
```

(*End definition for* `\__sort_quick_split:NnNn` *and others.*)

`\__sort_quick_end:nnTFNn`
`\__sort_quick_single_end:nnnwnw`
`\__sort_quick_only_i_end:nnnwnw`
`\__sort_quick_only_ii_end:nnnwnw`
`\__sort_quick_split_end:nnnwnw`

The `\__sort_quick_end:nnTFNn` appears instead of the user's conditional, and receives as its arguments the pivot #1, a fake item #2, a `true` and a `false` branches #3 and #4, followed by an ending function #5 (one of the four auxiliaries here) and another copy #6 of the fake item. All those are discarded except the function #5. This function receives lists #1 and #2 of items less than or greater than the pivot #3, then a continuation code #5 just after `\q_mark`. To avoid a memory problem described earlier, all of the ending functions read #6 until `\q_stop` and place #6 back into the input stream. When

the lists #1 and #2 are empty, the single auxiliary simply places the continuation #5 before the pivot {#3}. When #2 is empty, #1 is sorted and placed before the pivot {#3}, taking care to feed the continuation #5 as a continuation for the function sorting #1. When #1 is empty, #2 is sorted, and the continuation argument is used to place the continuation #5 and the pivot {#3} before the sorted result. Finally, when both lists are non-empty, items larger than the pivot are sorted, then items less than the pivot, and the continuations are done in such a way to place the pivot in between.

```
\cs_new:Npn \__sort_quick_end:nnTFNn #1#2#3#4#5#6 {#5}
\cs_new:Npn \__sort_quick_single_end:nnnwnw #1#2#3#4 \q_mark #5#6 \q_stop
  { #5 {#3} #6 \q_stop }
\cs_new:Npn \__sort_quick_only_i_end:nnnwnw #1#2#3#4 \q_mark #5#6 \q_stop
  {
    \__sort_quick_split:NnNn #1
      \__sort_quick_end:nnTFNn { } \q_mark {#5}
    {#3}
    #6 \q_stop
  }
\cs_new:Npn \__sort_quick_only_ii_end:nnnwnw #1#2#3#4 \q_mark #5#6 \q_stop
  {
    \__sort_quick_split:NnNn #2
      \__sort_quick_end:nnTFNn { } \q_mark { #5 {#3} }
    #6 \q_stop
  }
\cs_new:Npn \__sort_quick_split_end:nnnwnw #1#2#3#4 \q_mark #5#6 \q_stop
  {
    \__sort_quick_split:NnNn #2 \__sort_quick_end:nnTFNn { } \q_mark
      {
        \__sort_quick_split:NnNn #1
          \__sort_quick_end:nnTFNn { } \q_mark {#5}
        {#3}
      }
    #6 \q_stop
  }
```

(*End definition for* `\__sort_quick_end:nnTFNn` *and others.*)

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.

Symbols

`\__clist_wrap_item:n` 73
`\__int_eval:w` 83, 99
`\__msg_kernel_error:nnx` 245
`\__msg_kernel_new:nnnn` 248
`\__prg_break:` 176
`\__prg_break_point:` 169, 176
`\__seq_item:n` 41, 47
`\__sort_clist:NNn` 61, 62, 64, 65
`\__sort_level:` 30, 86, 86, 92
`\__sort_main:NNNnNn` 16, 16, 39, 45, 51, 57, 69
`\__sort_merge_blocks:` . . . 90, 95, 95, 112
`\__sort_merge_blocks_aux:` 111, 125, 125, 139, 149
`\__sort_merge_blocks_end:` 147, 151, 151, 159
`\__sort_quick_cleanup:w` . . 161, 182, 185
`\__sort_quick_end:nnTFNn` 181, 216, 216, 222, 229, 234, 237
`\__sort_quick_only_i:NnnnnNn` 186, 188, 192, 194
`\__sort_quick_only_i_end:nnnwnw` 195, 216, 219
`\__sort_quick_only_ii:NnnnnNn` 186, 188, 198, 200
`\__sort_quick_only_ii_end:nnnwnw` 201, 216, 226
`\__sort_quick_prepare:Nnnn` 161, 167, 174, 177
`\__sort_quick_prepare_end:NNNnw` 161, 169, 179
`\__sort_quick_single_end:nnnwnw` 189, 216, 217
`\__sort_quick_split:NnNn` 181, 186, 186, 221, 228, 234, 236
`\__sort_quick_split_end:nnnwnw` 207, 213, 216, 232
`\__sort_quick_split_i:NnnnnNn` 186, 200, 204, 206, 212
`\__sort_quick_split_ii:NnnnnNn` 186, 194, 206, 210, 212
`\__sort_toks:NNw` 41, 47, 53, 59, 73, 78, 78, 82
`\__sort_too_long_error:NNw` . 23, 242, 242

`\␣` 253

C

`\c__sort_max_length_int` . . . 7, 7, 22, 252
`\c_max_register_int` 251
`\c_minus_one` 108, 109, 110, 134, 135, 144, 145, 154, 155
`\c_one` 26, 29, 83, 118, 119
`\c_two` 91
`\c_zero` 19, 89
`\clist_gsort:Nn` 2, 61, 63
`\clist_if_empty:NF` 67
`\clist_map_break:` 70
`\clist_map_inline:Nn` 70
`\clist_sort:Nn` 2, 61, 61
`\cs_new:Npn` 78, 161, 174, 179, 185, 186, 192, 198, 204, 210, 216, 217, 219, 226, 232
`\cs_new_protected:Npn` 16, 65, 242
`\cs_new_protected_nopar:Npn` 37, 43, 49, 55, 61, 63, 86, 95, 115, 125, 131, 141, 151

\cs_set:Npn 28

E

\exp_after:wN 82, 83, 92, 112, 127, 128, 129
\exp_args:No 73
\exp_last_unbraced:Nf 72
\exp_not:f 163
\exp_not:N 34, 41, 47
\exp_stop_f: 182
\ExplFileDate 5
\ExplFileDescription 5
\ExplFileName 5
\ExplFileVersion 5

F

\fi: 24, 84, 93, 104, 113, 122, 138, 148, 158, 242, 244

G

\group_begin: 18
\group_end: 33

I

\if_int_compare:w 22, 80, 88, 99, 102, 120, 136, 146, 156
\int_const:Nn 7
\int_eval:n 251
\int_new:N 9, 10, 11, 12, 13, 14, 15
\int_use:N 83, 252

L

\l__sort_A_int 13, 13, 100, 105, 108, 128, 143, 145, 146
\l__sort_B_int 13, 14, 105, 109, 117, 119, 120, 133, 134, 143, 144, 153, 154, 156
\l__sort_begin_int . . . 11, 11, 97, 146, 156
\l__sort_block_int 10, 10, 29, 88, 91, 98, 101
\l__sort_C_int 13, 15, 106, 110, 117, 118, 129, 133, 135, 136, 153, 155
\l__sort_end_int 11, 12, 89, 97, 98, 99, 100, 101, 102, 103, 120
\l__sort_length_int 9, 9, 19, 22, 25, 26, 80, 88, 99, 102, 103, 106, 136
\luatex_if_engine:TF 8

P

\prg_do_nothing: 53, 59
\ProvidesExplPackage 4

Q

\q_mark 182, 217, 219, 222, 226, 229, 232, 234, 237
\q_stop 170, 179, 183, 185, 217, 218, 219, 224, 226, 230, 232, 240

S

\s__stop 183, 185
\seq_gsort:Nn *1*, 37, 43
\seq_map_break: 40, 46
\seq_map_inline:Nn 40, 46
\seq_sort:Nn *1*, 37, 37
\sort_compare:nn 28, 127
\sort_copy_block: 107, 115, 115, 123
\sort_ordered: 131, 131
\sort_reversed: 141, 141

T

\tex_advance:D 26, 98, 101, 108, 109, 110, 118, 119, 134, 135, 144, 145, 154, 155
\tex_multiply:D 91
\tex_the:D 81, 128, 129
\tex_toks:D 25, 81, 117, 128, 129, 133, 143, 153
\tl_gset:Nn 45, 57, 64
\tl_gsort:Nn *2*, 49, 55
\tl_if_blank:nF 165
\tl_map_break: 52, 58
\tl_map_inline:Nn 52, 58
\tl_set:Nn 39, 51, 62
\tl_sort:Nn *2*, 49, 49
\tl_sort:nN *2*, 161, 161
\token_to_str:N 245

U

\use:x 31
\use_i:nn 121, 137, 147, 157
\use_none:n 72