The l3dt package
Data tables*

The LaTeX3 Project†

Released 2012/08/29

LaTeX3 implements a "data table" variable type, which is made up of a series of rows each of which contain a number of key–value pairs. Thus a data table is in effect an array of property lists. The rows of the table are stored in a fixed order, and are numbered consecutively from one. In the same way, the order of keys (columns) is recorded in a sequence-like manner, again indexed from one.

Within each row in a data table each entry must have a unique ⟨*key*⟩: if an entry is added to a row within a data table which already contains the ⟨*key*⟩ then the new entry will overwrite the existing one. The ⟨*keys*⟩ are compared on a string basis, using the same method as `\str_if_eq:nn`.

1 Creating and initialising data tables

`\dt_new:N` `\dt_new:N` ⟨*data table*⟩

Creates a new ⟨*data table*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*property lists*⟩ will initially contain no entries.
function

`\dt_clear:N` `\dt_clear:N` ⟨*data table*⟩
`\dt_gclear:N`

Clears all entries and keys from the ⟨*data table*⟩.
function

`\dt_clear_new:N` `\dt_clear_new:N` ⟨*data table*⟩
`\dt_gclear_new:N`

Ensures that the ⟨*data table*⟩ exists globally by applying `\dt_new:N` if necessary, then applies `\dt_(g)clear:N` to leave the table empty.
function

`\dt_set_eq:NN` `\dt_set_eq:NN` ⟨*data table1*⟩ ⟨*data table2*⟩
`\dt_gset_eq:NN`

Sets the content of ⟨*data table1*⟩ equal to that of ⟨*data table2*⟩.
function

*This file describes v4153, last revised 2012/08/29.

†E-mail: latex-team@latex-project.org

2 Adding data

\dt_add_key:Nn
\dt_gadd_key:Nn

`\dt_add_key:Nn` ⟨*dt*⟩ {⟨*key*⟩}

Adds the ⟨*key*⟩ to the list of those in the ⟨*data table*⟩. The ⟨*key*⟩ will be converted to a string using `\tl_to_str:n`, and thus category codes in the ⟨*key*⟩ are ignored. If the ⟨*key*⟩ is already present in the ⟨*data table*⟩ then no action is taken.

function

\dt_add_row:N
\dt_gadd_row:N

`\dt_add_row:N` ⟨*dt*⟩

Adds a new row to the ⟨*data table*⟩. This will initially contain no entries: all keys will be be blank.

function

\dt_put:Nnn
\dt_gput:Nnn

`\dt_put:Nnn` ⟨*dt*⟩ {⟨*key*⟩} {⟨*value*⟩}

Adds an entry to the current row of the ⟨*data table*⟩ which may be accessed using the ⟨*key*⟩ and which has ⟨*value*⟩. Both the ⟨*key*⟩ and ⟨*value*⟩ may contain any ⟨*balanced text*⟩. The ⟨*key*⟩ is stored after processing with `\tl_to_str:n`, meaning that category codes are ignored. If the ⟨*key*⟩ is already present in the current row of the ⟨*data table*⟩, the existing entry is overwritten by the new ⟨*value*⟩.

function

\dt_put:Nnnn
\dt_gput:Nnnn

`\dt_put:Nnnn` ⟨*dt*⟩ {⟨*row*⟩} {⟨*key*⟩} {⟨*value*⟩}

Adds an entry to the ⟨*row*⟩ of the ⟨*data table*⟩ which may be accessed using the ⟨*key*⟩ and which has ⟨*value*⟩. Both the ⟨*key*⟩ and ⟨*value*⟩ may contain any ⟨*balanced text*⟩. The ⟨*key*⟩ is stored after processing with `\tl_to_str:n`, meaning that category codes are ignored. If the ⟨*key*⟩ is already present in the ⟨*row*⟩ of the ⟨*data table*⟩, the existing entry is overwritten by the new ⟨*value*⟩. The ⟨*row*⟩ should be given as an ⟨*integer expression*⟩.

function

3 Removing data

\dt_remove:Nn
\dt_gremove:Nn

`\dt_remove:Nn` ⟨*dt*⟩ {⟨*key*⟩}

Deletes any entry from the current row of the ⟨*data table*⟩ with the ⟨*key*⟩. The ⟨*key*⟩ is compared after processing with `\tl_to_str:n`, meaning that category codes are ignored. Deleting of all entries from a row does not delete the row itself.

function

\dt_remove:Nnn
\dt_gremove:Nnn

`\dt_remove:Nnn` ⟨*dt*⟩ {⟨*row*⟩} {⟨*key*⟩}

Deletes any entry from the ⟨*row*⟩ of the ⟨*data table*⟩ with the ⟨*key*⟩. The ⟨*key*⟩ is compared after processing with `\tl_to_str:n`, meaning that category codes are ignored. The ⟨*row*⟩ may be given as an ⟨*integer expression*⟩. Deleting of all entries from a row does not delete the row itself.

function

\dt_remove_key:Nn
\dt_gremove_key:Nn

`\dt_remove_key:N` ⟨*data table*⟩ {⟨*key*⟩}

Removes the ⟨*key*⟩ from the ⟨*data table*⟩ if it is present. The ⟨*key*⟩ and any associated ⟨*value*⟩ will be removed from any row that it is found in.
function

\dt_remove_row:Nn
\dt_gremove_row:Nn

`\dt_remove_row:Nn` ⟨*data table*⟩ {⟨*row*⟩}

Removes the ⟨*row*⟩ (given as an ⟨*integer expressions*⟩) from the ⟨*data table*⟩. The remaining rows of the table will be renumbered such that they are sequential.
function

4 Recovering information

\dt_keys:N ⋆

`\dt_keys:N` ⟨*dt*⟩

Leaves the number of keys in the ⟨*data table*⟩ in the input stream as an ⟨*integer denotation*⟩.
XP

\dt_rows:N ⋆

`\dt_rows:N` ⟨*dt*⟩

Leaves the number of rows in the ⟨*data table*⟩ in the input stream as an ⟨*integer denotation*⟩.
XP

\dt_get:NnN

`\dt_get:NnnN` ⟨*dt*⟩ {⟨*key*⟩} ⟨*tl var*⟩

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from the current row in the ⟨*data table*⟩, and places this in the ⟨*token list variable*⟩. If the ⟨*key*⟩ is not found in the ⟨*row*⟩ of the ⟨*data table*⟩ then the ⟨*token list variable*⟩ will contain the special marker `\q_no_value`. The ⟨*token list variable*⟩ is set within the current TEX group. The ⟨*row*⟩ should be given as an ⟨*integer expression*⟩. See also `\dt_get:NnNTF`.
function

\dt_get:NnNTF

`\dt_get:NnnNTF` ⟨*dt*⟩ {⟨*key*⟩} ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from the current row in the ⟨*data table*⟩, and places this in the ⟨*token list variable*⟩. If the ⟨*key*⟩ is not found in the ⟨*row*⟩ of the ⟨*data table*⟩ then the ⟨*token list variable*⟩ will contain the special marker `\q_no_value`. The ⟨*token list variable*⟩ is set within the current TEX group. The ⟨*row*⟩ should be given as an ⟨*integer expression*⟩. Once the ⟨*token list variable*⟩ has been assigned either the ⟨*true code*⟩ or ⟨*false code*⟩ will be left in the input stream, depending on whether the ⟨*key*⟩ was found. See also `\dt_get:NnN`.
F

\dt_get:NnnN

\dt_get:NnnN ⟨*dt*⟩ {⟨*row*⟩} {⟨*key*⟩} ⟨*tl var*⟩

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from ⟨*row*⟩ in the ⟨*data table*⟩, and places this in the ⟨*token list variable*⟩. If the ⟨*key*⟩ is not found in the ⟨*row*⟩ of the ⟨*data table*⟩ then the ⟨*token list variable*⟩ will contain the special marker `\q_no_value`. The ⟨*token list variable*⟩ is set within the current TEX group. The ⟨*row*⟩ should be given as an ⟨*integer expression*⟩. See also `\dt_get:NnnNTF`.

function

\dt_get:NnnNTF

\dt_get:NnnNTF ⟨*dt*⟩ {⟨*row*⟩} {⟨*key*⟩} ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from ⟨*row*⟩ in the ⟨*data table*⟩, and places this in the ⟨*token list variable*⟩. If the ⟨*key*⟩ is not found in the ⟨*row*⟩ of the ⟨*data table*⟩ then the ⟨*token list variable*⟩ will contain the special marker `\q_no_value`. The ⟨*token list variable*⟩ is set within the current TEX group. The ⟨*row*⟩ should be given as an ⟨*integer expression*⟩. Once the ⟨*token list variable*⟩ has been assigned either the ⟨*true code*⟩ or ⟨*false code*⟩ will be left in the input stream, depending on whether the ⟨*key*⟩ was found. See also `\dt_get:NnnN`.

F

5 Mapping to data tables

\dt_map_variables:Nnn

\dt_map_variables:Nnn ⟨*data table*⟩ {⟨*key-variable mapping*⟩} {⟨*code*⟩}

Applies the ⟨*code*⟩ to each ⟨*row*⟩ of the ⟨*data table*⟩. The ⟨*keys*⟩ of the ⟨*data table*⟩ are mapped to variables by the ⟨*key–variable mapping*⟩, which should be a key–value list of the form

```
key-a = \l_a_tl ,
key-b = \l_b_tl
...
```

It is not necessary to map all of the ⟨*keys*⟩ in a ⟨*data table*⟩ to variables. If there is not ⟨*value*⟩ for a ⟨*key*⟩ in a row, the variable will contain the marker `\q_no_value`. Assignment of the ⟨*variables*⟩ is local to the current TEX group. The mapping to rows is ordered.

function

\g_dt_map_level_int

The nesting level of the data table mapping is available as `\g_dt_map_level_int`. Within a mapping, the `int` variable `\l_dt_map_`⟨*level*⟩`_row_int` is available so that the row number being operated on is available. Thus

```
\int_use:c { l_dt_map_ \int_use:N \g_dt_map_level_int _row_int }
```

will give the current row for the current mapping.

variable

\dt_map_break: ☆
\dt_map_break:n ☆

\dt_map_break:
\dt_map_break:n {⟨*tokens*⟩}

Used to terminate a \dt_map_... function before all entries in the ⟨*data table*⟩ have been processed. This will normally take place within a conditional statement, for example

```
\dt_map_variables:Nn \l_my_dt { a = \l_my_tl }
  {
    \str_if_eq:VnTF \l_my_tl { bingo }
      { \dt_map_break: }
      {
        % Do something useful
      }
  }
```

The :n variant will insert the ⟨*tokens*⟩ into the input stream after the mapping terminates. Use outside of a \dt_map_... scenario will lead low level TEX errors.

EXP

6 Data table conditionals

\dt_if_empty_p:N ⋆
\dt_if_empty:NTF ⋆

\dt_if_empty_p:N ⟨*dt*⟩
\dt_if_empty:NTF ⟨*dt*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*dt*⟩ is empty, containing no keys and no rows.

XP, pTF

\dt_if_in_p:Nn ⋆
\dt_if_in:NnTF ⋆

\dt_if_in_p:Nn ⟨*dt*⟩ {⟨*key*⟩}
\dt_if_in:NnTF ⟨*dt*⟩ {⟨*key*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*key*⟩ is present in the ⟨*data table*⟩ at all, *i.e.* if it is one of the columns of the table. This test will be true even if none of the rows contain an entry for the ⟨*key*⟩.

XP, pTF

\dt_if_in_row_p:Nnn ⋆
\dt_if_in_row:NnnTF ⋆

\dt_if_in_row_p:Nnn ⟨*dt*⟩ {⟨*row*⟩} {⟨*key*⟩}
\dt_if_in_row:NnnTF ⟨*dt*⟩ {⟨*row*⟩} {⟨*key*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*key*⟩ is present in the ⟨*row*⟩ of the ⟨*data table*⟩. The ⟨*row*⟩ may be given as an ⟨*integer expression*⟩.

XP, pTF

\dt_if_in_row_p:Nn ⋆
\dt_if_in_row:NnTF ⋆

\dt_if_in_row_p:Nn ⟨*dt*⟩ {⟨*key*⟩}
\dt_if_in_row:NnTF ⟨*dt*⟩ {⟨*key*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*key*⟩ is present in the current row of ⟨*data table*⟩.

XP, pTF

7 Variables

\c_empty_dt

A permanently empty data table.

variable

\l_tmpa_dt
\l_tmpb_dt
\g_tmpa_dt
\g_tmpb_dt

Scratch data tables for general use: these are never used by the kernel.

variable

8 l3dt implementation

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

8.1 Structures

The structure of a data table must allow each row (record) to contain only some of the keys, and for the keys to be removed after the table is initialised. It also needs to ensure that a unique match can be made to every item in the table. At the same time, it is desirable to keep all of the information about the table in a single TEX macro. This can be achieved by packing the data into a structure in which each key and row is numbered:

{⟨*rows*⟩}
{⟨*columns*⟩}
\q__dt ⟨key_1⟩ \q__dt ⟨key_2⟩ \q__dt ...
\q_nil
\q__dt_header
\q__dt_row
⟨row_1⟩
\q__dt ⟨key_1⟩ \q__dt {⟨$data_{1,1}$⟩}
\q__dt ⟨key_2⟩ \q__dt {⟨$data_{1,2}$⟩}
...
\q__dt
\q_nil
\q__dt_row
⟨row_2⟩
\q__dt ⟨key_1⟩ \q__dt {⟨$data_{2,1}$⟩}

```
\q__dt ⟨key_2⟩ \q__dt {⟨data_{2,2}⟩}
...
\q__dt
\q_nil
\q__dt_row
...
\q__dt_row
```

\q__dt
\q__dt_row
\q__dt_header

The quarks are set up.

```
\quark_new:N \q__dt
\quark_new:N \q__dt_row
\quark_new:N \q__dt_header
```

(*End definition for* `\q__dt`, `\q__dt_row`, *and* `\q__dt_header` *These variables are documented on page* ??.)

\c_empty_dt

A permanently-empty data table, which therefore contains only the minimum number of items necessary to comply with the structure above.

```
\tl_const:Nn \c_empty_dt
  {
    { 0 }
    { 0 }
    \q__dt
    \q_nil
    \q__dt_header
    \q__dt_row
  }
```

(*End definition for* `\c_empty_dt` *This variable is documented on page 6.*)

8.2 Allocation and initialisation

\dt_new:N

Internally, data tables are token lists, but an empty dt is not an empty tl.

```
\cs_new_protected:Npn \dt_new:N #1 { \cs_new_eq:NN #1 \c_empty_dt }
```

(*End definition for* `\dt_new:N` *This function is documented on page 1.*)

\dt_clear:N
\dt_gclear:N

The same idea for clearing.

```
\cs_new_protected:Npn \dt_clear:N #1  { \cs_set_eq:NN  #1 \c_empty_dt }
\cs_new_protected:Npn \dt_gclear:N #1 { \cs_gset_eq:NN #1 \c_empty_dt }
```

(*End definition for* `\dt_clear:N` *and* `\dt_gclear:N` *These functions are documented on page 1.*)

\dt_clear_new:N
\dt_gclear_new:N

Once again a simple copy from the token list functions.

```
\cs_new_protected:Npn \dt_clear_new:N #1
  { \cs_if_exist:NTF #1 { \dt_clear:N #1 } { \dt_new:N #1 } }
\cs_new_protected:Npn \dt_gclear_new:N #1
  { \cs_if_exist:NTF #1 { \dt_gclear:N #1 } { \dt_new:N #1 } }
```

(*End definition for* `\dt_clear_new:N` *and* `\dt_gclear_new:N` *These functions are documented on page 1.*)

\dt_set_eq:NN
\dt_gset_eq:NN

Once again, these are simply copies from the token list functions.

```
\cs_new_eq:NN \dt_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \dt_gset_eq:NN \tl_gset_eq:NN
```

(*End definition for* `\dt_set_eq:NN` *and* `\dt_gset_eq:NN` *These functions are documented on page 1.*)

\l_tmpa_dt
\l_tmpb_dt
\g_tmpa_dt
\g_tmpb_dt

Scratch tables.

```
\dt_new:N \l_tmpa_dt
\dt_new:N \l_tmpb_dt
\dt_new:N \g_tmpa_dt
\dt_new:N \g_tmpb_dt
```

(*End definition for* `\l_tmpa_dt` *and others. These variables are documented on page 6.*)

8.3 Splitting functions

__dt_split:nnnn
__dt_split:w

Two general auxiliaries. The `nnnn` function is used to apply the `T` branch if a match is found and the `F` branch otherwise. The `w` function is general purpose, and is used to define the matching parameter set.

```
\cs_new_protected:Npn \__dt_split:nnnn #1#2#3#4 { #3 #2 }
\cs_new_protected:Npn \__dt_split:w { }
```

(*End definition for* `\__dt_split:nnnn` *This function is documented on page 6.*)

__dt_split_header:NT
__dt_split_header:wn

Splits the header from the table, inserting the code required to then process the split table. The `\q_nil` is also removed from the end of the header, as it is essentially a distraction here.

```
\cs_new:Npn \__dt_split_header:NT #1#2
  { \exp_after:wN \__dt_split_header:wn #1 \q_stop {#2} }
\cs_new:Npn \__dt_split_header:wn #1 \q_nil \q__dt_header #2 \q_stop #3
  { #3 {#1} { \q__dt_header #2 } }
```

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

__dt_split_key:nnTF
__dt_split_key_aux:nnTF

Here, the split is made for a partial list within a row. The row is basically the same as a property list, so the split here is almost identical to that in `\prop_split_aux:NnTF`. The row-end data is set up such that it will not interfere with this process.

```
\cs_new_protected:Npn \__dt_split_key:nnTF #1#2
  { \exp_args:No \__dt_split_key_aux:nnTF { \tl_to_str:n {#2} } {#1} }
\cs_new_protected:Npn \__dt_split_key_aux:nnTF #1#2
  {
    \cs_set_protected:Npn \__dt_split:w
      ##1 \q__dt #1 \q__dt ##2##3##4 \q_mark ##5 \q_stop
      { \__dt_split:nnnn ##3 { { ##1 \q__dt } {##2} {##4} } }
    \__dt_split:w #2 \q_mark
      \q__dt #1 \q__dt { } { ? \use_ii:nn { } } \q_mark \q_stop
  }
```

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

__dt_split_key_list:NnTF
__dt_split_key_list_aux:NnTF

Finding a key in the header uses a similar approach to finding a key in a property list. Here, if the key is found there will always be at least one token between `\q__dt` and `\q__dt_header` due to the `\q_nil` which is part of a new table. The use of `##1##2##3` in `\__dt_split:w` here is to deal with the overall number of rows and keys. The set up here means that these will always be unbraced then rebraced: simply grabbing `##1##2` to include this and anything before the key of interest will give variable results depending on whether the match is to the very first key or not.

```
\cs_new_protected:Npn \__dt_split_key_list:NnTF #1#2
  { \exp_args:NNo \__dt_split_key_list_aux:NnTF #1 { \tl_to_str:n {#2} } }
\cs_new_protected:Npn \__dt_split_key_list_aux:NnTF #1#2
  {
    \cs_set_protected:Npn \__dt_split:w
      ##1##2##3 \q__dt #2 \q__dt ##4##5 \q__dt_header ##6 \q_mark ##7 \q_stop
      {
        \__dt_split:nnnn ##4
          { { {##1} {##2} ##3 \q__dt } { ##4##5 \q__dt_header ##6 } }
      }
    \exp_after:wN \__dt_split:w #1 \q_mark
      \q__dt #2 \q__dt { ? \use_ii:nn { } } \q__dt_header \q_mark \q_stop
  }
```

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

__dt_split_row:NnTF
__dt_split_row_aux:NnTF
__dt_split_row_aux:NfTF

The usual approach, here using the fact that each row start with row number and ends with `\q_nil` so there will always be at least one token to be absorbed as `##2`. The only odd thing to watch here is that the row number is evaluated so that higher-level functions in the main do not need to have an `f`-type variant.

```
\cs_new_protected:Npn \__dt_split_row:NnTF #1#2
  { \__dt_split_row_aux:NfTF #1 { \int_eval:n {#2} } }
\cs_new_protected:Npn \__dt_split_row_aux:NnTF #1#2
  {
    \cs_set_protected:Npn \__dt_split:w
      ##1 \q__dt_row #2 \q__dt ##2##3 \q__dt_row ##4 \q_mark ##5 \q_stop
      {
        \__dt_split:nnnn ##2
          { { ##1 \q__dt_row } { #2 \q__dt ##2##3 } {##4} }
      }
    \exp_after:wN \__dt_split:w #1 \q_mark
      \q__dt_row #2 \q__dt { ? \use_ii:nn { } } \q__dt_row \q_mark \q_stop
  }
\cs_generate_variant:Nn \__dt_split_row_aux:NnTF { Nf }
```

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

8.4 Adding and removing data

\dt_add_key:Nn
\dt_gadd_key:Nn
__dt_add_key:NNn
__dt_add_key:NNnnn
__dt_add_key:NNnnn

Here, there are two stages. If the key is already present in the list of known keys then no action is taken, and the split list is thrown away. On the other hand, if the key is not present then the header and body are separated and the key is added to the end of the

list of known keys (hence keys are ordered). The `\__dt_split_header:Nn` function will have removed the `\q_nil` from the header, and so it is put back in here.

```
\cs_new_protected_nopar:Npn \dt_add_key:Nn { \__dt_add_key:NNn \tl_set:Nx }
\cs_new_protected_nopar:Npn \dt_gadd_key:Nn { \__dt_add_key:NNn \tl_gset:Nx }
\cs_new_protected:Npn \__dt_add_key:NNn #1#2#3
  {
    \__dt_split_key_list:NnTF #2 {#3}
      { \use_none:nn }
      {
        \__dt_split_header:NT #2
          { \__dt_add_key:NNnnn #1 #2 {#3} }
      }
  }
\cs_new_protected:Npn \__dt_add_key:NNnnn #1#2#3#4#5
  { \__dt_add_key:NNnnnwnn #1 #2 #4 \q_stop {#3} {#5} }
\cs_new_protected:Npn \__dt_add_key:NNnnnwnn #1#2#3#4#5 \q_stop #6#7
  {
    #1 #2
      {
        {#3}
        { \int_eval:n { #4 + \c_one } }
        \exp_not:n {#5}
        \tl_to_str:n {#6}
        \exp_not:n { \q__dt \q_nil #7 }
      }
  }
```

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

`\dt_add_row:N`
`\dt_gadd_row:N`
`\__dt_add_row:NN`
`\__dt_add_row:NnN`
`\__dt_add_row:NfN`
`\__dt_add_row:nw`

Adding a row means incrementing the total number and adding the structure of an empty row. As finding the rows will get slow for large tables, this is only done once.

```
\cs_new_protected_nopar:Npn \dt_add_row:N { \__dt_add_row:NN \tl_set:Nx }
\cs_new_protected_nopar:Npn \dt_gadd_row:N { \__dt_add_row:NN \tl_gset:Nx }
\cs_new_protected:Npn \__dt_add_row:NN #1#2
  { \__dt_add_row:NfN #1 { \int_eval:n { \dt_rows:N #2 + \c_one } } #2 }
\cs_new_protected:Npn \__dt_add_row:NnN #1#2#3
  {
    #1 #3
      {
        {#2}
        \exp_after:wN \__dt_add_row:nw #3 \q_stop
        #2
        \exp_not:n { \q__dt \q_nil \q__dt_row }
      }
  }
\cs_generate_variant:Nn \__dt_add_row:NnN { Nf }
\cs_new:Npn \__dt_add_row:nw #1#2 \q_stop { \exp_not:n {#2} }
```

(*End definition for* `\dt_add_row:N` *and* `\dt_gadd_row:N` *These functions are documented on page 2.*)

`\dt_put:Nnn`
`\dt_gput:Nnn`

Adding to the current row is simply a special case of adding to an arbitrary row.

```
\cs_new_protected:Npn \dt_put:Nnn #1
  { \dt_put:Nnnn #1 { \dt_rows:N #1 } }
\cs_new_protected:Npn \dt_gput:Nnn #1
  { \dt_gput:Nnnn #1 { \dt_rows:N #1 } }
```

(*End definition for* `\dt_put:Nnn` *and* `\dt_gput:Nnn` *These functions are documented on page 2.*)

`\dt_put:Nnnn`
`\dt_gput:Nnnn`
`\__dt_put:NNNnnn`
`\__dt_put:NNnnnnn`
`\__dt_put_update:NNnnnnnnn`
`\__dt_put_add_to_row:NNnnnnn`
`\__dt_put_add_to_row_aux:w`

Adding to a row is a slightly complex procedure. The lead-off is the standard combination across the local and global routes.

```
\cs_new_protected_nopar:Npn \dt_put:Nnnn
  { \__dt_put:NNNnnn \dt_add_key:Nn \tl_set:Nx }
\cs_new_protected_nopar:Npn \dt_gput:Nnnn
  { \__dt_put:NNNnnn \dt_gadd_key:Nn \tl_gset:Nx }
```

Add the key to the list those known, if necessary, then check that the row requested makes sense.

```
\cs_new_protected:Npn \__dt_put:NNNnnn #1#2#3#4#5#6
  {
    #1 #3 {#5}
    \__dt_split_row:NnTF #3 {#4}
      { \__dt_put:NNnnnnn #2 #3 {#5} {#6} }
      {
        \__msg_kernel_error:nnxxx { dt } { unknown-row }
          { \token_to_str:N #3 } { \int_eval:n {#4} } { \dt_rows:N #3 }
      }
  }
```

At this stage, the arguments are

1. the set function `\tl_(g)set:Nx`,
2. the data table,
3. the key,
4. the value,
5. the data table before the row,
6. the extracted data table row,
7. the data table after the row.

Splitting on the key will then leave three further items in the input stack if the key is already present. So there is some care needed sending the parameters forward without running out of TEX arguments.

```
\cs_new_protected:Npn \__dt_put:NNnnnnn #1#2#3#4#5#6#7
  {
    \__dt_split_key:nnTF {#6} {#3}
      { \__dt_put_update:NNnnnnnnn #1 #2 {#3} {#4} {#5} {#7} }
      { \__dt_put_add_to_row:NNnnnnn #1 #2 {#3} {#4} {#5} {#6} {#7} }
  }
```

The arguments here are

1. the set function `\tl_(g)set:Nx`,
2. the data table,
3. the key,
4. the value,
5. the data table before the row,
6. the data table after the row,
7. the row before the key,
8. the current value for the key
9. the row after the key.

What happens here is a reconstruction of the table: everything except `#8` is needed. To try to keep things clear, there are a few more `\exp_not:n` here than formally required.

```
\cs_new_protected:Npn \__dt_put_update:NNnnnnnnn #1#2#3#4#5#6#7#8#9
  {
    #1 #2
      {
        \exp_not:n { #5 #7 }
        \tl_to_str:n {#3}
        \exp_not:n { \q__dt {#4} \q__dt #9 \q__dt_row #6 }
      }
  }
```

A slightly more complex case when adding an item. The arguments here are identical to those for `\__dt_put:NNnnnnnn`. The row has not been split, so the `\q_nil` there is removed and re-added to come after the new content.

```
\cs_new_protected:Npn \__dt_put_add_to_row:NNnnnnn #1#2#3#4#5#6#7
  {
    #1 #2
      {
        \exp_not:n {#5}
        \exp_not:o { \__dt_put_add_to_row_aux:w #6 }
        \tl_to_str:n {#3}
        \exp_not:n { \q__dt {#4} \q__dt \q_nil \q__dt_row #7 }
      }
  }
\cs_new:Npn \__dt_put_add_to_row_aux:w #1 \q_nil {#1}
```

(*End definition for* `\dt_put:Nnnn` *and* `\dt_gput:Nnnn` *These functions are documented on page* 2.)

`\dt_keys:N`
`\__dt_keys:nnw`
`\dt_rows:N`

The number of rows in a dt is the very first entry. Getting the number of keys is almost the same, except a custom auxiliary is needed.

```
\cs_new:Npn \dt_keys:N #1 { \exp_after:wN \__dt_keys:nnw #1 \q_stop }
```

```
\cs_new:Npn \__dt_keys:nnw #1#2#3 \q_stop {#2}
\cs_new:Npn \dt_rows:N #1
  { \exp_after:wN \use_i_delimit_by_q_stop:nw #1 \q_stop }
```

(*End definition for* `\dt_keys:N` *This function is documented on page 3.*)

8.5 Removing data

`\dt_remove:Nn`
`\dt_gremove:Nn`

Deleting to the current row is simply a special case of deleting to an arbitrary row.

```
\cs_new_protected:Npn \dt_remove:Nn #1
  { \dt_remove:Nnn #1 { \dt_rows:N #1 } }
\cs_new_protected:Npn \dt_gremove:Nn #1
  { \dt_gremove:Nnn #1 { \dt_rows:N #1 } }
```

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

`\dt_remove:Nnn`
`\dt_gremove:Nnn`
`\dt_remove_aux:NNnn`
`\dt_remove_aux:NNnnnn`
`\dt_remove_aux:NNnnnnn`

Deleting a single entry from a single row means first splitting by row, then splitting by key, and finally doing the assignment. If the row or the key are not present then the entire function does nothing at all.

```
\cs_new_protected_nopar:Npn \dt_remove:Nnn { \dt_remove_aux:NNnn \tl_set:Nn }
\cs_new_protected_nopar:Npn \dt_gremove:Nnn { \dt_remove_aux:NNnn \tl_gset:Nn }
\cs_new_protected:Npn \dt_remove_aux:NNnn #1#2#3#4
  {
    \__dt_split_row:NnTF #2 {#3}
      { \dt_remove_aux:NNnnnn #1 #2 {#4} }
      { }
  }
\cs_new_protected:Npn \dt_remove_aux:NNnnnn #1#2#3#4#5#6
  {
    \__dt_split_key:nnTF {#5} {#3}
      { \dt_remove_aux:NNnnnnn #1 #2 {#4} {#6} }
      { }
  }
\cs_new_protected:Npn \dt_remove_aux:NNnnnnn #1#2#3#4#5#6#7
  { #1 #2 { #3 #5 #7 #4 } }
```

(*End definition for* `\dt_remove:Nnn` *and* `\dt_gremove:Nnn` *These functions are documented on page 2.*)

`\dt_remove_key:Nn`
`\dt_gremove_key:Nn`
`\dt_remove_key_aux:NNn`
`\dt_remove_key_aux:nNNnn`
`\dt_remove_key_aux:w`

Deleting a key also removes from the table itself, so that there is no need to do any awkward checks when extracting data from the table. (It's likely that there will be more cases of accessing data than deleting rows). The deletion mapping ignores rows entirely and just pulls out matching key–value pairs, as this reduces the number of matches needed to a minimum.

```
\cs_new_protected_nopar:Npn \dt_remove_key:Nn
  { \dt_remove_key_aux:NNn \tl_set:Nx }
\cs_new_protected_nopar:Npn \dt_gremove_key:Nn
  { \dt_remove_key_aux:NNn \tl_gset:Nx }
\cs_new_protected:Npn \dt_remove_key_aux:NNn #1#2#3
  {
    \__dt_split_key_list:NnTF #2 {#3}
      { \exp_args:No \dt_remove_key_aux:nNNnn { \tl_to_str:n {#3} } #1 #2 }
```

```
      { }
  }
\cs_new_protected:Npn \dt_remove_key_aux:nNNnn #1#2#3#4#5
  { \dt_remove_key_aux:nNNnnwn {#1} #2 #3 #4 \q_stop {#5} }
\cs_new_protected:Npn \dt_remove_key_aux:nNNnnwn #1#2#3#4#5#6 \q_stop #7
  {
    \cs_set:Npn \dt_remove_key_aux:w ##1 \q__dt #1 \q__dt ##2 ##3
      {
        \exp_not:n {##1}
        \__quark_if_recursion_tail_break:nN {##3} \dt_map_break:
        \dt_remove_key_aux:w ##3
      }

    #2 #3
      {
        {#4}
        { \int_eval:n { #5 - \c_one } }
        \exp_not:n {#6}
        \dt_remove_key_aux:w #7 \q__dt #1 \q__dt { } \q_recursion_tail
          \__prg_break_point:Nn \dt_map_break: { }
      }
  }
\cs_new:Npn \dt_remove_key_aux:w { }
```

(*End definition for* `\dt_remove_key:Nn` *and* `\dt_gremove_key:Nn` *These functions are documented on page 3.*)

`\dt_remove_row:Nn`
`\dt_gremove_row:Nn`
`\dt_remove_row_aux:NNn`
`\dt_remove_row_aux:NNnnnn`
`\dt_remove_row_aux:nw`
`\dt_remove_row_loop:nw`

Removing a row is a slightly complex operation as there are two stages. The row itself is easy enough to remove, but then all later rows have to be renumbers.

```
\cs_new_protected_nopar:Npn \dt_remove_row:Nn
  { \dt_remove_row_aux:NNn \tl_set:Nx }
\cs_new_protected_nopar:Npn \dt_gremove_row:Nn
  { \dt_remove_row_aux:NNn \tl_gset:Nx }
\cs_new_protected:Npn \dt_remove_row_aux:NNn #1#2#3
  {
    \__dt_split_row:NnTF #2 {#3}
      { \dt_remove_row_aux:NNnnn #1 #2 }
      { }
  }
```

If the code gets here, then `#3` is the table before the removed row, `#4` is the removed row and `#5` is everything afterwards. The first stage is to work out the new number of rows, then include all of `#3` except the old number of rows. The removed row `#4` is thrown away, and then there is a loop to recalculate the row numbers for all of the later rows.

```
\cs_new_protected:Npn \dt_remove_row_aux:NNnnn #1#2#3#4#5
  {
    #1 #2
      {
        { \int_eval:n { \dt_rows:N #2 - \c_one } }
        \dt_remove_row_aux:nw #3 \q_stop
        \dt_remove_row_loop:nw #5 \q_recursion_tail \q__dt_row
```

```
          \__prg_break_point:Nn \dt_map_break: { }
      }
  }
\cs_new_eq:NN \dt_remove_row_aux:nw \__dt_add_row:nw
\cs_new:Npn \dt_remove_row_loop:nw #1#2 \q__dt_row
  {
    \__quark_if_recursion_tail_break:nN {#1} \dt_map_break:
    \int_eval:n { #1 - \c_one }
    \exp_not:n { #2 \q__dt_row }
    \dt_remove_row_loop:nw
  }
```

(*End definition for* `\dt_remove_row:Nn` *and* `\dt_gremove_row:Nn` *These functions are documented on page 3.*)

8.6 Accessing data in data tables

`\dt_get:NnnN`
`\dt_get_aux:nNnnn`
`\dt_get_aux:nNnnn`

Recovering a value from a row means doing two splits: first find the row, then find the key. Nothing exciting, just a question of tracking the returned items.

```
\cs_new_protected:Npn \dt_get:NnnN #1#2#3#4
  {
    \__dt_split_row:NnTF #1 {#2}
      { \dt_get_aux:nNnnn {#3} #4 }
      { \tl_set:Nn #4 { \q_no_value } }
  }
\cs_new_protected:Npn \dt_get_aux:nNnnn #1#2#3#4#5
  {
    \__dt_split_key:nnTF {#4} {#1}
      { \dt_get_aux:Nnnn #2 }
      { \tl_set:Nn #2 { \q_no_value } }
  }
\cs_new_protected:Npn \dt_get_aux:Nnnn #1#2#3#4 { \tl_set:Nn #1 {#3} }
```

(*End definition for* `\dt_get:NnnN` *This function is documented on page 4.*)

`\dt_get:NnnNTF`
`\__dt_get_true:nNnnn`
`\__dt_get_true:Nnnn`

The same idea as the standard method, but built as a conditional.

```
\prg_new_protected_conditional:Npnn \dt_get:NnnN #1#2#3#4 { T , F , TF }
  {
    \__dt_split_row:NnTF #1 {#2}
      { \__dt_get_true:nNnnn {#3} #4 }
      { \prg_return_false: }
  }
\cs_new_protected:Npn \__dt_get_true:nNnnn #1#2#3#4#5
  {
    \__dt_split_key:nnTF {#4} {#1}
      { \__dt_get_true:Nnnn #2 }
      { \prg_return_false: }
  }
\cs_new_protected:Npn \__dt_get_true:Nnnn #1#2#3#4
  {
    \tl_set:Nn #1 {#3}
```

```
    \prg_return_true:
  }
```

(*End definition for* `\dt_get:NnnN` *This function is documented on page 4.*)

`\dt_get:NnN` `\dt_get:NnNTF` Simple wrappers.

```
\cs_new_protected:Npn \dt_get:NnN #1
  { \dt_get:NnnN  #1 { \dt_rows:N #1 } }
\cs_new_protected:Npn \dt_get:NnNT #1
  { \dt_get:NnnNF  #1 { \dt_rows:N #1 } }
\cs_new_protected:Npn \dt_get:NnNF #1
  { \dt_get:NnnNF  #1 { \dt_rows:N #1 } }
\cs_new_protected:Npn \dt_get:NnNTF #1
  { \dt_get:NnnNTF #1 { \dt_rows:N #1 } }
```

(*End definition for* `\dt_get:NnN` *This function is documented on page 3.*)

8.7 Mapping to data tables

`\g_dt_map_level_int` Unlike other mappings, the mapping level here has to be available and so linked to the module.

```
\int_new:N \g_dt_map_level_int
```

(*End definition for* `\g_dt_map_level_int` *This variable is documented on page 4.*)

`\dt_map_variables:Nnn` `\__dt_map_variables_key:nn` `\__dt_map_variables:nnn` `\__dt_map_variables:nNNw` `\__dt_map_variables:nnw` Mapping across a data table is more complex than other cases as there are two "dimensions" to worry about: the rows and the keys. The first stage of the mapping is to convert the key–variable mapping into a sequence that can be used later. This is done with the assumption that any key without a variable can simply be dropped entirely. The header of the table is then split from the body.

```
\cs_new_protected:Npn \dt_map_variables:Nnn #1#2#3
  {
    \int_gincr:N \g_dt_map_level_int
    \seq_gclear_new:c { g_dt_map_ \int_use:N \g_dt_map_level_int _seq }
    \keyval_parse:NNn \use_none:n \__dt_map_variables_key:nn {#2}
    \__dt_split_header:NT #1 { \__dt_map_variables:nnn {#3} }
  }
\cs_new_protected:Npn \__dt_map_variables_key:nn #1#2
  {
    \seq_gput_right:cn { g_dt_map_ \int_use:N \g_dt_map_level_int _seq }
      { {#1} #2 }
  }
```

As `\__dt_split_header:NT` will leave a couple of tokens at the front of the body part of the split, there is a quick piece of tidying up to remove them.

```
\cs_new_protected:Npn \__dt_map_variables:nnn #1#2#3
  { \__dt_map_variables:nNNw {#1} #3 \q_stop }
\cs_new_protected:Npn \__dt_map_variables:nNNw
  #1 \q__dt_header \q__dt_row #2 \q_stop
  {
    \int_zero_new:c { l_dt_map_ \int_use:N \g_dt_map_level_int _row_int }
```

```
    \__dt_map_variables:nnw {#1} #2 { } \q_recursion_tail \q__dt_row
      \__prg_break_point:Nn \dt_map_break:
        { \int_gdecr:N \g_dt_map_level_int }
  }
\cs_new_protected:Npn \__dt_map_variables:nnw #1#2#3#4 \q__dt_row
  {
    \__quark_if_recursion_tail_break:nN {#3} \dt_map_break:
    \seq_map_inline:cn { g_dt_map_ \int_use:N \g_dt_map_level_int _seq }
      { \dt_get_aux:nNnnn ##1 { } {#3#4} { } }
    #1
    \int_incr:c { l_dt_map_ \int_use:N \g_dt_map_level_int _row_int }
    \__dt_map_variables:nnw {#1}
  }
```

(*End definition for* `\dt_map_variables:Nnn` *This function is documented on page 4.*)

`\dt_map_break:`
`\dt_map_break:n`

The break statements use the general `\__prg_map_break:Nn`.

```
\cs_new_nopar:Npn \dt_map_break:
  { \__prg_map_break:Nn \dt_map_break: { } }
\cs_new_nopar:Npn \dt_map_break:n
  { \__prg_map_break:Nn \dt_map_break: }
```

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

8.8 Data table conditionals

`\dt_if_empty_p:N`
`\dt_if_empty:NTF`

An empty data table has not only no rows but also no keys. (The number of rows can be tested using `\dt_rows:N` and an `int` test.)

```
\prg_new_conditional:Npnn \dt_if_empty:N #1 { T , F , TF , p }
  {
    \if_meaning:w #1 \c_empty_dt
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\dt_if_empty:N` *These functions are documented on page 5.*)

`\dt_if_in_p:Nn`
`\dt_if_in:NnTF`
`\__dt_if_in:nnn`
`\__dt_if_in:nwN`
`\__dt_if_in:n`

Expandably checking for the presence of a key in the table as a whole requires a mapping to the header. The idea is the usual recursion set up with a string-based comparison only after checking for the end of the loop.

```
\prg_new_conditional:Npnn \dt_if_in:Nn #1#2 { p , T , F , TF }
  { \__dt_split_header:NT #1 { \__dt_if_in:nnn {#2} } }
\cs_new:Npn \__dt_if_in:nnn #1#2#3
  {
    \exp_last_unbraced:Nno \__dt_if_in:nwN {#1} { \use_none:nn #2 }
      \q_recursion_tail \q__dt
      \__prg_break_point:
  }
\cs_new:Npn \__dt_if_in:nwN #1#2 \q__dt
```

```
  {
    \if_meaning:w \q_recursion_tail #2
      \exp_after:wN \__prg_break:n
    \else:
      \exp_after:wN \use_none:n
    \fi:
      { \prg_return_false: }
    \str_if_eq:nnTF {#1} {#2}
      { \__prg_break:n { \prg_return_true: } }
      { \__dt_if_in:nwN {#1} }
  }
```

(*End definition for* `\dt_if_in:Nn` *These functions are documented on page 5.*)

`\dt_if_in_row_p:Nnn`
`\dt_if_in_row:NnnTF`
`\__dt_if_in_row:nw`
`\__dt_if_in_row:nn`
`\__dt_if_in_row:nwn`
`\__dt_if_in_row:N`

Finding a key in a single row in an expandable way requires two mappings. To start of with, there is a search for the row. This uses for termination the fact that each row starts `\q__dt_row` and ends `\q_nil`, and always contains at least the row number as the first ⟨*balanced text*⟩. That can be replaced by the tail marker to terminate iteration: all that is then needed is the correct placement of the clean-up code.

```
\prg_new_conditional:Npnn \dt_if_in_row:Nnn #1#2#3 { p , T , F , TF }
  {
    \exp_last_unbraced:Nno \__dt_if_in_row:nw {#2} #1
      \q_recursion_tail \q_nil
      \__prg_break_point:
      { \tl_to_str:n {#3} }
  }
```

The row iteration does a numerical comparison to see if the target row has been found. That means that the row argument does not need to be converted to a number earlier.

```
\cs_new:Npn \__dt_if_in_row:nw #1#2 \q__dt_row #3#4 \q_nil
  {
    \if_meaning:w \q_recursion_tail #3
      \exp_after:wN \__prg_break:n
    \else:
      \exp_after:wN \use_none:n
    \fi:
      {
        \use_i:nn
        \prg_return_false:
      }
    \int_compare:nNnTF {#1} = {#3}
      { \__prg_break:n { \exp_args:Nno \__dt_if_in_row:nn {#4} } }
      { \__dt_if_in_row:nw {#1} }
  }
```

The second iteration is along the row. This is basically the same as `\prop_if_in:NnTF` with the `\q__dt` in place of `\q_prop`.

```
\cs_new:Npn \__dt_if_in_row:nn #1#2
  {
    \__dt_if_in_row:nwn {#2} #1 {#2} \q__dt { } \q_recursion_tail
      \__prg_break_point:
```

```
  }
\cs_new:Npn \__dt_if_in_row:nwn #1 \q__dt #2 \q__dt #3
  {
    \str_if_eq:xxTF {#1} {#2}
      { \__dt_if_in_row:N }
      { \__dt_if_in_row:nwn {#1} }
  }
\cs_new:Npn \__dt_if_in_row:N #1
  {
    \if_meaning:w \q__dt #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
    \__prg_break:
  }
```

(*End definition for* `\dt_if_in_row:Nnn` *These functions are documented on page 5.*)

`\dt_if_in_row_p:Nn`
`\dt_if_in_row:NnTF`

Simple wrappers.

```
\cs_new:Npn \dt_if_in_row_p:Nn #1
  { \dt_if_in_row_p:Nnn #1 { \dt_rows:N #1 } }
\cs_new:Npn \dt_if_in_row:NnT  #1
  { \dt_if_in_row:NnnT  #1 { \dt_rows:N #1 } }
\cs_new:Npn \dt_if_in_row:NnF  #1
  { \dt_if_in_row:NnnF  #1 { \dt_rows:N #1 } }
\cs_new:Npn \dt_if_in_row:NnTF #1
  { \dt_if_in_row:NnnTF #1 { \dt_rows:N #1 } }
```

(*End definition for* `\dt_if_in_row:Nn` *These functions are documented on page 5.*)

8.9 Messages

```
\__msg_kernel_new:nnnn { dt } { unknown-row }
  { Data~table~#1~does~not~contain~a~row~'#2'. }
  {
    Data~table~#1~contains~#3~rows.~These~must~be~accessed~by~number:~row~
    #2~is~not~present~in~the~table.
  }
```

⟨/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

`\__dt_add_key:NNn` 75, 75, 76, 77
`\__dt_add_key:NNnnn` 75, 75, 83, 86
`\__dt_add_key:NNnnnwnn` 87, 88

__dt_add_row:NN 99, 99, 100, 101
__dt_add_row:NfN 99, 102
__dt_add_row:NnN 99, 103, 113
__dt_add_row:nw 99, 108, 114, 234
__dt_get_true:Nnnn 255, 264, 267
__dt_get_true:nNnnn 255, 258, 261
__dt_if_in:n 324
__dt_if_in:nnn 324, 325, 326
__dt_if_in:nwN 324, 328, 332, 342
__dt_if_in_row:N 344, 374, 377
__dt_if_in_row:nn 344, 363, 366
__dt_if_in_row:nw 344, 346, 351, 364
__dt_if_in_row:nwn . . . 344, 368, 371, 375
__dt_keys:nnw 159, 159, 160
__dt_map_variables:nNNw . . 281, 294, 295
__dt_map_variables:nnn . . 281, 286, 293
__dt_map_variables:nnw 281, 299, 303, 310
__dt_map_variables_key:nn 281, 285, 288
__dt_put:NNNnnn 119, 120, 122, 123
__dt_put:NNnnnnn 119, 127, 133
__dt_put_add_to_row:NNnnnnn 119, 137, 148
__dt_put_add_to_row_aux:w 119, 153, 158
__dt_put_update:NNnnnnnnn 119, 136, 139
__dt_split:nnnn 32, 32, 44, 55, 68
__dt_split:w 32, 33, 42, 45, 52, 58, 65, 71
__dt_split_header:NT 34, 34, 82, 286, 325
__dt_split_header:wn 34, 35, 36
__dt_split_key:nnTF 38, 38, 135, 177, 250, 263
__dt_split_key_aux:nnTF 38, 39, 40
__dt_split_key_list:NnTF 48, 48, 79, 189
__dt_split_key_list_aux:NnTF 48, 49, 50
__dt_split_row:NnTF 61, 61, 126, 171, 220, 244, 257
__dt_split_row_aux:NfTF 61, 62
__dt_split_row_aux:NnTF 61, 63, 74
__msg_kernel_error:nnxxx 129
__msg_kernel_new:nnnn 394
__prg_break: 384
__prg_break:n 335, 341, 354, 363
__prg_break_point: 330, 348, 369
__prg_break_point:Nn 210, 231, 300
__prg_map_break:Nn 313, 315
__quark_if_recursion_tail_break:nN 200, 237, 305

C

\c_empty_dt *6*, 10, 10, 19, 20, 21, 318
\c_one 93, 102, 207, 228, 238
\cs_generate_variant:Nn 74, 113
\cs_gset_eq:NN 21
\cs_if_exist:NTF 23, 25
\cs_new:Npn 34, 36, 114, 158, 159, 160, 161, 213, 235, 326, 332, 351, 366, 371, 377, 386, 388, 390, 392
\cs_new_eq:NN 19, 26, 27, 234
\cs_new_nopar:Npn 312, 314
\cs_new_protected:Npn 19, 20, 21, 22, 24, 32, 33, 38, 40, 48, 50, 61, 63, 77, 86, 88, 101, 103, 115, 117, 123, 133, 139, 148, 163, 165, 169, 175, 181, 187, 193, 195, 218, 224, 242, 248, 254, 261, 267, 272, 274, 276, 278, 281, 288, 293, 295, 303
\cs_new_protected_nopar:Npn 75, 76, 99, 100, 119, 121, 167, 168, 183, 185, 214, 216
\cs_set:Npn 197
\cs_set_eq:NN 20
\cs_set_protected:Npn 42, 52, 65

D

\dt_add_key:Nn *2*, 75, 75, 120
\dt_add_row:N *2*, 99, 99
\dt_clear:N *1*, 20, 20, 23
\dt_clear_new:N *1*, 22, 22
\dt_gadd_key:Nn *2*, 75, 76, 122
\dt_gadd_row:N *2*, 99, 100
\dt_gclear:N *1*, 20, 21, 25
\dt_gclear_new:N *1*, 22, 24
\dt_get:NnN *3*, 272, 272
\dt_get:NnNF 276
\dt_get:NnnN *4*, 242, 242, 255, 273
\dt_get:NnnNF 275, 277
\dt_get:NnnNTF *4*, 255, 279
\dt_get:NnNT 274
\dt_get:NnNTF *3*, 272, 278
\dt_get_aux:Nnnn 251, 254
\dt_get_aux:nNnnn 242, 242, 245, 248, 307
\dt_gput:Nnn *2*, 115, 117
\dt_gput:Nnnn *2*, 118, 119, 121
\dt_gremove:Nn *2*, 163, 165
\dt_gremove:Nnn *2*, 166, 167, 168
\dt_gremove_key:Nn *3*, 183, 185
\dt_gremove_row:Nn *3*, 214, 216
\dt_gset_eq:NN *1*, 26, 27
\dt_if_empty:N 316
\dt_if_empty:NTF *5*, 316
\dt_if_empty_p:N *5*, 316

\dt_if_in:Nn 324
\dt_if_in:NnTF 5, 324
\dt_if_in_p:Nn 5, 324
\dt_if_in_row:NnF 390
\dt_if_in_row:Nnn 344
\dt_if_in_row:NnnF 391
\dt_if_in_row:NnnT 389
\dt_if_in_row:NnnTF 5, 344, 393
\dt_if_in_row:NnT 388
\dt_if_in_row:NnTF 5, 386, 392
\dt_if_in_row_p:Nn 5, 386, 386
\dt_if_in_row_p:Nnn 5, 344, 387
\dt_keys:N 3, 159, 159
\dt_map_break: 5, 200, 210, 231, 237, 300, 305, 312, 312, 313, 315
\dt_map_break:n 312, 314
\dt_map_variables:Nnn 4, 281, 281
\dt_new:N . 1, 19, 19, 23, 25, 28, 29, 30, 31
\dt_put:Nnn 2, 115, 115
\dt_put:Nnnn 2, 116, 119, 119
\dt_remove:Nn 2, 163, 163
\dt_remove:Nnn 2, 164, 167, 167
\dt_remove_aux:NNnn . . . 167, 167, 168, 169
\dt_remove_aux:NNnnnn 167, 172, 175
\dt_remove_aux:NNnnnnn . . . 167, 178, 181
\dt_remove_key:Nn 3, 183, 183
\dt_remove_key_aux:NNn 183, 184, 186, 187
\dt_remove_key_aux:nNNnn . . 183, 190, 193
\dt_remove_key_aux:nNNnnwn 194, 195
\dt_remove_key_aux:w 183, 197, 201, 209, 213
\dt_remove_row:Nn 3, 214, 214
\dt_remove_row_aux:NNn 214, 215, 217, 218
\dt_remove_row_aux:NNnnn 221, 224
\dt_remove_row_aux:NNnnnn 214
\dt_remove_row_aux:nw 214, 229, 234
\dt_remove_row_loop:nw 214, 230, 235, 240
\dt_rows:N 3, 102, 116, 118, 130, 159, 161, 164, 166, 228, 273, 275, 277, 279, 387, 389, 391, 393
\dt_set_eq:NN 1, 26, 26

E

\else: 320, 336, 355, 381
\exp_after:wN 35, 58, 71, 108, 159, 162, 335, 337, 354, 356
\exp_args:NNo 49
\exp_args:Nno 363
\exp_args:No 39, 190
\exp_last_unbraced:Nno 328, 346
\exp_not:n 94, 96, 110, 114, 143, 145, 152, 155, 199, 208, 239
\exp_not:o 153
\ExplFileDate 5
\ExplFileDescription 5
\ExplFileName 5
\ExplFileVersion 5

F

\fi: 322, 338, 357, 383

G

\g_dt_map_level_int 4, 280, 280, 283, 284, 290, 298, 301, 306, 309
\g_tmpa_dt 6, 28, 30
\g_tmpb_dt 6, 28, 31

I

\if_meaning:w 318, 334, 353, 379
\int_compare:nNnTF 362
\int_eval:n . 62, 93, 102, 130, 207, 228, 238
\int_gdecr:N 301
\int_gincr:N 283
\int_incr:c 309
\int_new:N 280
\int_use:N 284, 290, 298, 306, 309
\int_zero_new:c 298

K

\keyval_parse:NNn 285

L

\l_tmpa_dt 6, 28, 28
\l_tmpb_dt 6, 28, 29

P

\prg_new_conditional:Npnn . 316, 324, 344
\prg_new_protected_conditional:Npnn 255
\prg_return_false: 259, 265, 321, 339, 360, 382
\prg_return_true: 270, 319, 341, 380
\ProvidesExplPackage 4

Q

\q__dt 7, 7, 14, 43, 44, 46, 53, 56, 59, 66, 69, 72, 96, 110, 145, 155, 197, 209, 329, 332, 368, 371, 379
\q__dt_header 7, 9, 16, 36, 37, 53, 56, 59, 296
\q__dt_row 7, 8, 17, 66, 69, 72, 110, 145, 155, 230, 235, 239, 296, 299, 303, 351
\q_mark . . . 43, 45, 46, 53, 58, 59, 66, 71, 72

\q_nil . . 15, 36, 96, 110, 155, 158, 347, 351
\q_no_value 246, 252
\q_recursion_tail 209, 230, 299, 329, 334, 347, 353, 368
\q_stop 35, 36, 43, 46, 53, 59, 66, 72, 87, 88, 108, 114, 159, 160, 162, 194, 195, 229, 294, 296
\quark_new:N 7, 8, 9

S

\seq_gclear_new:c 284
\seq_gput_right:cn 290
\seq_map_inline:cn 306
\str_if_eq:nnTF 340
\str_if_eq:xxTF 373

T

\tl_const:Nn 10
\tl_gset:Nn 168
\tl_gset:Nx 76, 100, 122, 186, 217
\tl_gset_eq:NN 27
\tl_set:Nn 167, 246, 252, 254, 269
\tl_set:Nx 75, 99, 120, 184, 215
\tl_set_eq:NN 26
\tl_to_str:n . 39, 49, 95, 144, 154, 190, 349
\token_to_str:N 130

U

\use_i:nn 359
\use_i_delimit_by_q_stop:nw 162
\use_ii:nn 46, 59, 72
\use_none:n 285, 337, 356
\use_none:nn 80, 328